Яцек Галовиц

C++17 STL

СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

Packt›

ПИТЕР®

C++17 STL Cookbook

Over 90 recipes that leverage the powerful features of the standard library in C++17

Jacek Galowicz

Packt›

BIRMINGHAM - MUMBAI

C++17 STL

СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

Яцек Галовиц

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

2018

ББК 32.973.2-018.1
УДК 004.43
Г16

Галовиц Я.

Г16 C++17 STL. Стандартная библиотека шаблонов. — СПб.: Питер, 2018. — 432 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0680-6

C++ — объектно-ориентированный язык программирования, без которого сегодня немыслима промышленная разработка ПО. В этой замечательной книге описана работа с контейнерами, алгоритмами, вспомогательными классами, лямбда-выражениями и другими интересными инструментами, которыми богат современный C++. Освоив материал, вы сможете коренным образом пересмотреть привычный подход к программированию. Преимущество издания — в подробном описании стандартной библиотеки шаблонов C++, STL. Ее свежая версия была выпущена в 2017 году. В книге вы найдете более 90 максимально реалистичных примеров, которые демонстрируют всю мощь STL. Многие из них станут базовыми кирпичиками для решения более универсальных задач. Вооружившись этой книгой, вы сможете эффективно использовать C++17 для создания высококачественного и высокопроизводительного ПО, применимого в различных отраслях.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1787120495 англ.

ISBN 978-5-4461-0680-6

Краткое содержание

Оглавление

Предисловие

Книга по C++, которую вы держите в руках, научит вас максимально эффективно использовать C++17. В ней представлены примеры написания кода, основанные на языке C++ и его стандартной библиотеке шаблонов (Standard Template Library, STL). Поскольку в рассматриваемых здесь примерах STL применяется по максимуму, об этой библиотеке стоит сказать подробнее.

C++ — мощный язык. Он позволяет скрывать сложные решения за простыми высокоуровневыми интерфейсами, но в то же время дает возможность писать низкоуровневый код, для которого важны высокая производительность и незначительное потребление ресурсов. Комитет ISO по стандартизации C++ старательно работает над улучшением стандарта C++.

Сегодняшний C++ (сам язык и библиотека шаблонов) предоставляет средства для работы со сложными структурами данных и алгоритмами, предлагает возможность управления ресурсами с помощью автоматических указателей, а также поддерживает лямбда-выражения, константные выражения, переносимые средства управления потоками для параллельного (конкурентного) программирования, регулярные выражения, генераторы случайных чисел, исключения, шаблоны с переменным количеством аргументов (эта часть языка C++, отвечающая за шаблонные типы, является полной по Тьюрингу!), определенные пользователями литералы, переносимые средства работы с файловой системой и многое другое. Такое количество возможностей делает C++ универсальным языком, который идеально подходит для реализации высококачественного и высокопроизводительного программного обеспечения, применимого в различных отраслях.

Однако многие разработчики C++ охотно изучают сам язык, а библиотеку STL переводят на задний план. Применение языка C++ без поддержки стандартной библиотеки зачастую приводит к тому, что программы выглядят так, будто написаны с использованием классов, а не с учетом современных подходов. Это печально, ведь подобное применение языка не дает задействовать всю его мощь.

В четвертом издании своей книги *The C++ Programming Language* («Язык программирования C++»), включающем сведения в том числе о C++11, Бьярн Страуструп (Bjarne Stroustrup) пишет: «Пожалуйста, помните, что эти возможности

языка и стандартной библиотеки призваны поддержать приемы программирования, позволяющие разрабатывать качественное ПО. Для решения какой-то конкретной задачи их нужно использовать в комбинации друг с другом — как кирпичики из одного набора, — а не отдельно и изолированно друг от друга».

Именно этому посвящена настоящая книга и все приведенные в ней примеры. Они максимально приближены к реальной жизни, причем без ориентации на какие-либо внешние библиотеки — только на STL. Данное обстоятельство значительно упрощает работу, избавляя от утомительной настройки среды. Я очень надеюсь, что представленные здесь примеры привнесут вдохновение в вашу работу с языком C++. Возможно, вы обнаружите, что некоторые из них являются отличными стандартными кирпичиками для решения задач более высокого уровня.

Темы, рассмотренные в этой книге

Глава 1 *«Новые возможности C++17»* посвящена новому функционалу, привнесенному стандартом C++17 в сам язык C++. В следующих главах мы уже сконцентрируемся на функционале STL.

Глава 2 *«Контейнеры STL»* содержит объяснение, как были улучшены разнообразные контейнеры структур данных, предоставляемые STL. После рассмотрения всей коллекции контейнеров мы более детально изучим новые дополнения к ним.

Глава 3 *«Итераторы»* дает описание итераторов, которые представляют собой очень важную абстракцию и служат «клеем» между алгоритмами STL и контейнерами в моменты их совместного использования. Мы разберем всю концепцию итераторов с нуля, чтобы узнать, как лучше всего применять их в наших программах.

Глава 4 *«Лямбда-выражения»* посвящена лямбда-выражениям, позволяющим задействовать некоторые интересные приемы программирования. Лямбда-выражения, чьему появлению в стандарте C++11 способствовали чистые функциональные языки программирования, в C++14 и C++17 получили новые возможности.

Глава 5 *«Основы работы с алгоритмами STL»* знакомит со стандартными алгоритмами STL и способами их применения. Эти алгоритмы легки в использовании, имеют высокую производительность, хорошо протестированы и универсальны. Научившись работать с ними, можно концентрироваться на решениях конкретных задач, а не тратить время на изобретение велосипеда.

Глава 6 *«Сложные случаи использования алгоритмов STL»* показывает, как объединять простые алгоритмы STL для создания более сложных без необходимости писать один и тот же код несколько раз. В рамках этой главы мы будем строго придерживаться принципов STL.

Глава 7 *«Строки, классы потоков и регулярные выражения»* содержит подробный обзор классов STL, необходимых для работы со строками, обобщенными

потоками ввода/вывода и регулярными выражениями. Мы детально разберем каждый из этих элементов STL.

Глава 8 *«Вспомогательные классы»* рассказывает, как с помощью STL генерировать случайные числа, измерять время, управлять динамической памятью, изящно сообщать об ошибках, а также о многом другом. Мы рассмотрим очень полезные и переносимые вспомогательные классы, предоставляемые STL для решения подобных задач, и разберем новые классы, появившиеся в C++17.

Глава 9 *«Параллелизм и конкуренция»* демонстрирует существующие расширения C++, позволяющие реализовать параллелизм и конкуренцию, — эти темы стали очень важны с тех пор, как мы вступили в эру многоядерных процессоров. В C++11 и C++17 появились возможности, значительно облегчающие реализацию программ, работающих на нескольких ядрах и выполняющих задачи одновременно. В рамках данной главы мы и рассмотрим упомянутые концепции.

Глава 10 *«Файловая система»* показывает: несмотря на то что в STL всегда предоставлялась поддержка чтения отдельных файлов и управления ими, в C++17 появилось много новых, не зависящих от операционной системы способов работы с файловыми путями и просмотра каталогов. В рамках главы мы научимся пользоваться этим инструментарием.

Что вам нужно для работы с книгой

Все примеры этой книги максимально просты и автономны. Их нетрудно скомпилировать и запустить, но, в зависимости от того, какими операционной системой и компилятором вы пользуетесь, эти действия могут различаться. Рассмотрим, как скомпилировать и запустить любой из примеров и на что нужно обратить особое внимание.

Компиляция и запуск примеров

Весь код из этой книги был разработан и протестирован в операционных системах Linux и MacOS с использованием компиляторов GNU C++, **g++**, LLVM C++, **clang++**.

Можно сгенерировать пример из командной оболочки с помощью g++ следующей командой:

```
$ g++ -std=c++1z -o recipe_app recipe_code.cpp
```

Аналогичная команда с использованием clang++:

```
$ clang++ -std=c++1z –o recipe_app recipe_code.cpp
```

Оба примера командной строки предполагают, что `recipe_code.cpp` является текстовым файлом, содержащим ваш код C++. После компиляции программы исполняемый бинарный файл получит имя `recipe_app`, его можно будет запустить следующим образом:

```
$ ./recipe_app
```

Во многих примерах мы считываем все содержимое файлов через стандартный ввод. В таких случаях мы используем стандартные каналы UNIX и команду `cat`, чтобы направить содержимое файла в наше приложение:

```
$ cat file.txt | ./recipe_app
```

Это работает в ОС Linux и MacOS. В оболочке Microsoft Windows команда выглядит так:

```
> recipe_app.exe < file.txt
```

Если вы запускаете программы не из оболочки, а из интегрированной среды разработки Microsoft Visual Studio, то вам нужно открыть вкладку Configuration properties ▸ Debugging (Свойства конфигурации ▸ Отладка) и добавить фрагмент `< file.txt` в командную строку приложения, которое запускает Visual Studio.

Предупреждения для первопроходцев

Возможно, некоторые примеры из этой книги у вас не будут компилироваться. Все зависит от того, какая часть C++17 STL реализована в вашем дистрибутиве STL.

При написании книги приходилось добавлять префикс пути `experimental/` к заголовкам `<execution_policy>` и `<filesystem>`. Кроме того, в каталоге `experimental/` вашего дистрибутива STL могут находиться дополнительные заголовочные файлы, например `algorithm`, `numeric` и т. д., в зависимости от того, насколько новым и стабильным является дистрибутив.

Это верно и для пространств имен, поддерживающих новейшие возможности. Элементы библиотеки, включенные в экспериментальную часть STL, обычно экспортируются внутрь пространства имен `std::experimental` (а не `std`).

Для кого предназначено издание

Эта книга не для вас, если раньше вы не писали программы на C++ и не компилировали их. Однако при условии, что вы уже имеете базовые сведения об этом языке, она идеально подойдет в качестве руководства по C++.

Кроме того, книга будет вам полезна, если вы узнали себя в одном из следующих описаний:

- вы изучили основы C++, но теперь не знаете, что делать дальше, поскольку разница между вашими знаниями и знаниями профессионала — разработчика на C++ все еще велика;
- вы хорошо знаете C++, но плохо ориентируетесь в STL;
- вы знаете C++ по одному из старых стандартов, C++98, C++11 или C++14. Как бы давно вы ни использовали C++ в последний раз, книга познакомит вас со множеством новых возможностей STL.

Разделы

В издании вы найдете несколько заголовков, встречающихся чаще других («Как это делается» и «Как это работает»). В этих разделах даны четкие инструкции, как работать с примером.

Как это делается

В этом подразделе описываются шаги, необходимые для выполнения примера.

Как это работает

Здесь обычно содержится подробное объяснение действий из предыдущего подраздела.

Дополнительная информация

В этот подраздел включены дополнительные сведения о примере, позволяющие читателю более детально ознакомиться с темой.

Условные обозначения

В книге информация разного рода оформлена различными способами. Рассмотрим несколько вариантов оформления и их значение.

Код в тексте, имена баз данных, каталогов и файлов, а также расширения файлов, пути к файлам, ненастоящие (dummy) URL, пользовательский ввод и имена пользователей Twitter выделяются так: «Следующий шаг — редактирование файла `build.properties`».

Блок кода выглядит следующим образом:

```
my_wrapper<T1, T2, T3> make_wrapper(T1 t1, T2 t2, T3 t3)
{
    return {t1, t2, t3};
}
```

Новые термины и *важные слова* выделены курсивом. Слова, которые вы видите на экране, например меню или диалоговые окна, выглядят в тексте так: «После завершения нажмите кнопку Activate (Активизировать)».

Предупреждения и важные примечания оформлены так.

Советы и приемы оформлены таким образом.

Загрузка примеров кода

Файлы с примерами кода для книги можно скачать по адресу https://github.com/PacktPublishing/Cpp17-STL-Cookbook. Для этого выполните следующие шаги.

1. Перейдите по указанному выше адресу.
2. Нажмите кнопку Clone or Download (Клонировать или скачать).
3. На открывшейся панели выберите ссылку Download ZIP (Скачать Zip).

После загрузки файла убедитесь, что распаковали или извлекли каталог с помощью последней версии одной из следующих программ:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Мы предлагаем и другие пакеты с кодом из нашего обширного каталога книг и видеороликов, доступного на https://github.com/PacktPublishing/. Обратите на них внимание!

1 Новые возможности C++17

В этой главе:

- применение структурированных привязок (декомпозиции) для распаковки набора возвращаемых значений;
- ограничение области видимости переменных в выражениях `if` и `switch`;
- новые правила инициализатора с фигурными скобками;
- разрешение конструктору автоматически вывести полученный тип класса шаблона;
- упрощение принятия решений во время компиляции с помощью `constexpr-if`;
- подключение библиотек, перечисленных в заголовочных файлах, с использованием встраиваемых переменных;
- реализация вспомогательных функций с помощью выражений свертки.

Введение

Функциональность языка C++ значительно расширилась с выходом C++11, C++14 и недавней версии C++17. На текущий момент он совсем не похож на себя образца десятилетней давности. Стандарт C++ упорядочивает не только язык, но и STL.

В этой книге на большом количестве примеров показаны наилучшие способы использования возможностей STL. Но для начала в текущей главе мы сконцентрируемся на самых важных особенностях языка. Изучив их, вы сможете писать легко читаемый, удобный в сопровождении и выразительный код.

Мы рассмотрим, как получить доступ к отдельным элементам пар, кортежей и структур с помощью структурированных привязок и ограничить область видимости переменных благодаря новым возможностям по инициализации переменных внутри выражений `if` и `switch`. Синтаксические двусмысленности, появившиеся в C++11 из-за нового синтаксиса инициализатора с фигурными скобками, кото-

рый выглядит так же, как синтаксис списков инициализаторов, были исправлены в *новых правилах инициализатора с фигурными скобками*. Точный *тип* экземпляра шаблонного класса может быть *определен* по аргументам, переданным его конструктору, а если разные специализации шаблонного класса выполняются в разном коде, то это легко выразить с помощью `constexpr-if`. Обработка переменного количества параметров в шаблонных функциях значительно упростилась благодаря новым *выражениям свертки*. Наконец, стало гораздо удобнее определять доступные глобально статические объекты в библиотеках, указанных в заголовочных файлах, благодаря новой возможности объявлять встраиваемые переменные, что ранее было выполнимо только для функций.

Отдельные примеры данной главы могут оказаться более интересными для тех, кто реализует библиотеки, нежели для тех, кто пишет приложения. Для полноты картины мы рассмотрим несколько свойств, но вам не обязательно разбираться со всеми примерами главы прямо сейчас, чтобы понять остальной материал этой книги.

Применяем структурированные привязки (декомпозицию) для распаковки набора возвращаемых значений

В C++17 появилась новая возможность, объединяющая синтаксический сахар и автоматическое определение типа, — *структурированные привязки*. Эта функция помогает присваивать отдельные значения пар, кортежей и структур отдельным переменным. В других языках программирования этот механизм называется *распаковкой*.

Как это делается

Применение декомпозиции для присвоения значений нескольким переменным на основе одной упакованной структуры всегда выполняется за один шаг. Сначала рассмотрим, как это делалось до появления C++17. Затем взглянем на несколько примеров, в которых показаны способы воплощения этого в C++17.

1. Получаем доступ к отдельным значениям `std::pair`. Представьте, что у нас есть математическая функция `divide_remainder`, которая принимает в качестве параметров *делимое* и *делитель* и возвращает частное и остаток в `std::pair`.

   ```
   std::pair<int, int> divide_remainder(int dividend, int divisor);
   ```

   Рассмотрим следующий способ получения доступа к отдельным значениям полученной пары.

   ```
   const auto result (divide_remainder(16, 3));
   std::cout << "16 / 3 is "
             << result.first << " with a remainder of "
             << result.second << '\n';
   ```

Вместо выполнения действий, показанных во фрагменте выше, мы теперь можем присвоить отдельные значения конкретным переменным с говорящими именами, что более удобочитаемо:

```
auto [fraction, remainder] = divide_remainder(16, 3);
std::cout << "16 / 3 is "
          << fraction << " with a remainder of "
          << remainder << '\n';
```

2. Структурированные привязки работают и для `std::tuple`. Рассмотрим следующий пример функции, которая возвращает информацию о ценах на акции:

```
std::tuple<std::string,
           std::chrono::system_clock::time_point, unsigned>
stock_info(const std::string &name);
```

Присваивание результата ее работы отдельным переменным выглядит так же, как и в предыдущем примере:

```
const auto [name, valid_time, price] = stock_info("INTC");
```

3. Декомпозицию можно применять и для пользовательских структур. В качестве примера создадим следующую структуру.

```
struct employee {
    unsigned id;
    std::string name;
    std::string role;
    unsigned salary;
};
```

Теперь можно получить доступ к ее членам с помощью декомпозиции. Мы даже можем сделать это в цикле, если предполагается наличие целого вектора таких структур:

```
int main()
{
    std::vector<employee> employees {
        /* Инициализируется в другом месте */};
    for (const auto &[id, name, role, salary] : employees) {
        std::cout << "Name: "   << name
                  << "Role: "   << role
                  << "Salary: " << salary << '\n';
    }
    }
```

Как это работает

Структурированные привязки всегда применяются по одному шаблону:

```
auto [var1, var2, ...] = <выражение пары, кортежа, структуры или массива>;
```

- Количество переменных `var1`, `var2`... должно точно совпадать с количеством переменных в выражении, в отношении которого выполняется присваивание.

- Элементом `<выражение пары, кортежа, структуры или массива>` должен быть один из следующих объектов:
  - `std::pair`;
  - `std::tuple`;
  - структура. Все члены должны быть *нестатическими* и определенными в *одном базовом классе*. Первый объявленный член присваивается первой переменной, второй член — второй переменной и т. д.;
  - массив фиксированного размера.
- Тип может иметь модификаторы `auto`, `const auto`, `const auto&` и даже `auto&&`.

При необходимости пользуйтесь ссылками, а не создавайте копии. Это важно не только с точки зрения производительности.

Если в квадратных скобках вы укажете *слишком мало* или *слишком много* переменных, то компилятор выдаст ошибку.

```
std::tuple<int, float, long> tup {1, 2.0, 3};
auto [a, b] = tup; // Не работает
```

В этом примере мы пытаемся поместить кортеж с тремя переменными всего в две переменные. Компилятор незамедлительно сообщает нам об ошибке:

```
error: type 'std::tuple<int, float, long>' decomposes into 3 elements, but
only 2 names were provided
auto [a, b] = tup;
```

Дополнительная информация

С помощью структурированных привязок вы точно так же можете получить доступ к большей части основных структур данных библиотеки STL. Рассмотрим, например, цикл, который выводит все элементы контейнера `std::map`:

```
std::map<std::string, size_t> animal_population {
    {"humans",   7000000000},
    {"chickens", 17863376000},
    {"camels",   24246291},
    {"sheep",    1086881528},
    /* … */
};

for (const auto &[species, count] : animal_population) {
    std::cout << "There are " << count << " " << species
              << " on this planet.\n";
}
```

Пример работает потому, что в момент итерации по контейнеру `std::map` мы получаем узлы `std::pair<const key_type, value_type>` на каждом шаге этого

процесса. Именно эти узлы распаковываются с помощью структурированных привязок (`key_type` представляет собой строку с именем `species`, а `value_type` — переменную `count` типа `size_t`), что позволяет получить к ним доступ по отдельности в теле цикла.

До появления C++17 аналогичного эффекта можно было достичь с помощью `std::tie`:

```
int remainder;
std::tie(std::ignore, remainder) = divide_remainder(16, 5);
std::cout << "16 % 5 is " << remainder << '\n';
```

Здесь показано, как распаковать полученную пару в две переменные. Применение контейнера `std::tie` не так удобно, как использование декомпозиции, ведь нам надо *заранее* объявить все переменные, которые мы хотим связать. С другой стороны, пример демонстрирует преимущество `std::tie` *перед* структурированными привязками: значение `std::ignore` играет роль переменной-пустышки. В данном случае частное нас не интересует и мы отбрасываем его, связав с `std::ignore`.

Когда мы применяем декомпозицию, у нас нет переменных-пустышек tie, поэтому нужно привязывать все значения к именованным переменным. Это может оказаться неэффективным, если позже не задействовать некоторые переменные, но тем не менее компилятор может оптимизировать неиспользованное связывание.

Раньше функцию `divide_remainder` можно было реализовать следующим образом, используя выходные параметры:

```
bool divide_remainder(int dividend, int divisor,
                      int &fraction, int &remainder);
```

Получить к ним доступ можно так:

```
int fraction, remainder;
const bool success {divide_remainder(16, 3, fraction, remainder)};
if (success) {
    std::cout << "16 / 3 is " << fraction << " with a remainder of "
              << remainder << '\n';
}
```

Многие все еще предпочитают делать именно так, а не возвращать пары, кортежи и структуры. При этом они приводят следующие аргументы: код работает *быстрее*, поскольку мы не создаем промежуточные копии этих значений. Но для современных компиляторов это *неверно* — они изначально оптимизированы так, что подобные копии не создаются.

Помимо того что аналогичной возможности нет в языке C, возврат сложных структур в качестве выходных параметров долгое время считался медленным, поскольку объект сначала нужно инициализировать

в возвращающей функции, а затем скопировать в переменную, которая должна будет содержать возвращаемое значение на вызывающей стороне. Современные компиляторы поддерживают оптимизацию возвращаемых значений (return value optimization, RVO), что позволяет избежать создания промежуточных копий.

Ограничиваем область видимости переменных в выражениях if и switch

Максимальное ограничение области видимости переменных считается хорошим тоном. Иногда, однако, переменная должна получить какое-то значение, а потом нужно его проверить на соответствие тому или иному условию, чтобы продолжить выполнение программы. Для этих целей в C++17 была введена инициализация переменных в выражениях `if` и `switch`.

Как это делается

В данном примере мы воспользуемся новым синтаксисом в обоих контекстах, чтобы увидеть, насколько это улучшит код.

- Выражение `if`. Допустим, нужно найти символ в таблице символов с помощью метода `find` контейнера `std::map`:

```
if (auto itr (character_map.find(c)); itr != character_map.end()) {
    // *itr корректен. Сделаем с ним что-нибудь.
} else {
    // itr является конечным итератором. Не разыменовываем.
}
// здесь itr недоступен
```

- Выражение `switch`. Так выглядит код получения символа из пользовательского ввода и его одновременная проверка в выражении `switch` для дальнейшего управления персонажем компьютерной игры:

```
switch (char c (getchar()); c) {
    case 'a': move_left();  break;
    case 's': move_back();  break;
    case 'w': move_fwd();   break;
    case 'd': move_right(); break;
    case 'q': quit_game();  break;

    case '0'...'9': select_tool('0' - c); break;

    default:
        std::cout << "invalid input: " << c << '\n';
}
```

Как это работает

Выражения `if` и `switch` с инициализаторами по сути являются синтаксическим сахаром. Два следующих фрагмента кода эквивалентны:

До C++17:

```
{
    auto var (init_value);
    if (condition) {
        // Ветвь A. К переменной var можно получить доступ
    } else {
        // Ветвь B. К переменной var можно получить доступ
    }
    // К переменной var все еще можно получить доступ
}
```

Начиная с C++17:

```
if (auto var (init_value); condition) {
    // Ветвь A. К переменной var можно получить доступ
} else {
    // Ветвь B. К переменной var можно получить доступ
}
// К переменной var больше нельзя получить доступ
```

То же верно и для выражений `switch`.

До C++17:

```
{
    auto var (init_value);
    switch (var) {
    case 1: ...
    case 2: ...
    ...
    }
    // К переменной var все еще можно получить доступ
}
```

Начиная с C++17:

```
switch (auto var (init_value); var) {
case 1: ...
case 2: ...
  ...
}
// К переменной var больше нельзя получить доступ
```

Благодаря описанному механизму область видимости переменной остается минимальной. До C++17 этого можно было добиться только с помощью дополнительных фигурных скобок, как показано в соответствующих примерах. Короткие жизненные циклы уменьшают количество переменных в области видимости, что позволяет поддерживать чистоту кода и облегчает рефакторинг.

Дополнительная информация

Еще один интересный вариант — ограниченная область видимости критических секций. Рассмотрим следующий пример:

```
if (std::lock_guard<std::mutex> lg {my_mutex}; some_condition) {
    // Делаем что-нибудь
}
```

Сначала создается `std::lock_guard`. Этот класс принимает мьютекс в качестве аргумента конструктора. Он *запирает* мьютекс в конструкторе, а затем, когда выходит из области видимости, *отпирает* его в деструкторе. Таким образом, невозможно *забыть* отпереть мьютекс. До появления C++17 требовалась дополнительная пара скобок, чтобы определить область, где мьютекс снова откроется.

Не менее интересный пример — это область видимости слабых указателей. Рассмотрим следующий фрагмент кода:

```
if (auto shared_pointer (weak_pointer.lock()); shared_pointer != nullptr) {
    // Да, общий объект еще существует
} else {
    // К указателю shared_pointer можно получить доступ, но он является нулевым
}
// К shared_pointer больше нельзя получить доступ
```

Это еще один пример с бесполезной переменной `shared_pointer`. Она попадает в текущую область видимости, несмотря на то что потенциально является бесполезной за пределами условного блока `if` или дополнительных скобок!

Выражения `if` с инициализаторами особенно хороши при работе с *устаревшими* API, имеющими выходные параметры:

```
if (DWORD exit_code; GetExitCodeProcess(process_handle, &exit_code)) {
    std::cout << "Exit code of process was: " << exit_code << '\n';
}
// Бесполезная переменная exit_code не попадает за пределы условия if
```

`GetExitCodeProcess` — функция API ядра Windows. Она возвращает код для заданного дескриптора процесса, но только в том случае, если данный дескриптор корректен. После того как мы покинем этот условный блок, переменная станет бесполезной, поэтому она не нужна в нашей области видимости.

Возможность инициализировать переменные внутри блоков `if`, очевидно, очень полезна во многих ситуациях, особенно при работе с устаревшими API, которые используют выходные параметры.

Всегда ограничивайте области видимости с помощью инициализации в выражениях if и switch. Это позволит сделать код более компактным, простым для чтения, а в случае рефакторинга его будет проще перемещать.

Новые правила инициализатора с фигурными скобками

В C++11 появился новый синтаксис инициализатора с фигурными скобками {}. Он предназначен как для *агрегатной* инициализации, так и для вызова обычного конструктора. К сожалению, когда вы объединяли данный синтаксис с типом переменных auto, был высок шанс выразить не то, что вам нужно. В C++17 появился улучшенный набор правил инициализатора. В следующем примере вы увидите, как грамотно инициализировать переменные в C++17 и какой синтаксис при этом использовать.

Как это делается

Переменные инициализируются в один прием. При использовании синтаксиса инициализатора могут возникнуть две разные ситуации.

1. Применение синтаксиса инициализатора с фигурными скобками *без* выведения типа auto:

```
// Три идентичных способа инициализировать переменную типа int:
int x1 = 1;
int x2  {1};
int x3  (1);
std::vector<int> v1   {1, 2, 3};
// Вектор, содержащий три переменные типа int: 1, 2, 3
std::vector<int> v2 = {1, 2, 3}; // Такой же вектор
std::vector<int> v3   (10, 20);
// Вектор, содержащий десять переменных типа int,
// каждая из которых имеет значение 20
```

2. Использование синтаксиса инициализатора с фигурными скобками *с* выведением типа auto:

```
auto v   {1};         // v имеет тип int
auto w   {1, 2};      // ошибка: при автоматическом выведении типа
                      // непосредственная инициализация разрешена
                      // только одиночными элементами! (нововведение)
auto x = {1};         // x имеет тип std::initializer_list<int>
auto y = {1, 2};      // y имеет тип std::initializer_list<int>
auto z = {1, 2, 3.0}; // ошибка: нельзя вывести тип элемента
```

Как это работает

Отдельно от механизма выведения типа auto оператор {} ведет себя предсказуемо, по крайней мере при инициализации обычных типов. При инициализации контейнеров наподобие std::vector, std::list и т. д. инициализатор с фигурными скобками будет соответствовать конструктору std::initializer_list этого класса-

контейнера. При этом он не может соответствовать неагрегированным конструкторам (таковыми являются обычные конструкторы, в отличие от тех, что принимают список инициализаторов).

`std::vector`, например, предоставляет конкретный неагрегированный конструктор, заносящий в некоторое количество элементов одно и то же значение: `std::vector<int> v (N, value)`. При записи `std::vector<int> v {N, value}` выбирается конструктор `initializer_list`, инициализирующий вектор с двумя элементами: `N` и `value`. Об этом следует помнить.

Есть интересное различие между оператором `{}` и вызовом конструктора с помощью обычных скобок `()`. В первом случае не выполняется неявных преобразований типа: `int x (1.2);` и `int x = 1.2;` инициализируют переменную `x` значением `1`, округлив в нижнюю сторону число с плавающей точкой и преобразовав его к типу `int`. А вот выражение `int x {1.2};` не скомпилируется, поскольку должно *точно* соответствовать типу конструктора.

Кто-то может поспорить о том, какой стиль инициализации является лучшим. Любители стиля с фигурными скобками говорят, что последние делают процесс явным, переменная инициализируется при вызове конструктора и эта строка кода ничего не инициализирует повторно. Более того, при использовании фигурных скобок {} будет выбран единственный подходящий конструктор, в то время как в момент применения обычных скобок () — ближайший похожий конструктор, а также выполнится преобразование типов.

Дополнительное правило, включенное в C++17, касается инициализации с выведением типа `auto`: несмотря на то что в C++11 тип переменной `auto x {123};` (`std::initializer_list<int>` с одним элементом) будет определен корректно, скорее всего, это не тот тип, который нужен. В C++17 та же переменная будет типа `int`.

Основные правила:

- в конструкции `auto var_name {one_element};` переменная `var_name` будет иметь тот же тип, что и `one_element`;
- конструкция `auto var_name {element1, element2, ...};` недействительна и не будет скомпилирована;
- конструкция `auto var_name = {element1, element2, ...};` будет иметь тип `std::initializer_list<T>`, где `T` — тип всех элементов списка.

В C++17 гораздо сложнее случайно определить список инициализаторов.

Попытка скомпилировать эти примеры в разных компиляторах в режиме C++11 или C++14 покажет, что одни компиляторы автоматически выводят тип auto x {123}; как int, а другие — как std::initializer_list<int>. Подобный код может вызвать проблемы с переносимостью!

Разрешаем конструктору автоматически выводить полученный тип класса шаблона

Многие классы C++ обычно специализируются по типам, о чем легко догадаться по типам переменных, которые пользователь задействует при вызовах конструктора. Тем не менее до C++17 эти возможности не были стандартизированы. C++17 позволяет компилятору *автоматически* вывести типы шаблонов из вызовов конструктора.

Как это делается

Данную особенность очень удобно проиллюстрировать на примере создания экземпляров типа `std::pair` и `std::tuple`. Это можно сделать за один шаг:

```
std::pair  my_pair  (123, "abc");       // std::pair<int, const char*>
std::tuple my_tuple (123, 12.3, "abc"); // std::tuple<int, double, const char*>
```

Как это работает

Определим класс-пример, где автоматическое выведение типа шаблона будет выполняться на основе переданных значений:

```
template <typename T1, typename T2, typename T3>
class my_wrapper {
    T1 t1;
    T2 t2;
    T3 t3;

public:
    explicit my_wrapper(T1 t1_, T2 t2_, T3 t3_)
        : t1{t1_}, t2{t2_}, t3{t3_}
    {}

    /* … */
};
```

О'кей, это всего лишь еще один класс шаблона. Вот как мы раньше создавали его объект (инстанцировали шаблон):

```
my_wrapper<int, double, const char *> wrapper {123, 1.23, "abc"};
```

Теперь же можно опустить специализацию шаблона:

```
my_wrapper wrapper {123, 1.23, "abc"};
```

До появления C++17 это было возможно только при *реализации вспомогательной функции*:

```
my_wrapper<T1, T2, T3> make_wrapper(T1 t1, T2 t2, T3 t3)
{
    return {t1, t2, t3};
}
```

Используя подобные вспомогательные функции, можно было добиться такого же эффекта:

```
auto wrapper (make_wrapper(123, 1.23, "abc"));
```

STL предоставляет множество аналогичных инструментов: std::make_shared, std::make_unique, std::make_tuple и т. д. В C++17 эти функции могут считаться устаревшими. Но, конечно, они все еще будут работать для обеспечения обратной совместимости.

Дополнительная информация

Из данного примера мы узнали о *неявном выведении типа шаблона*. Однако в некоторых случаях на этот способ нельзя полагаться. Рассмотрим следующий класс-пример:

```
template <typename T>
struct sum {
    T value;

    template <typename ... Ts>
    sum(Ts&& ... values) : value{(values + ...)} {}
};
```

Эта структура, `sum`, принимает произвольное количество параметров и суммирует их с помощью выражений свертки (пример, связанный с выражениями свертки, мы рассмотрим далее в этой главе). Полученная сумма сохраняется в переменную-член `value`. Теперь вопрос заключается в том, что за тип — `T`? Если мы не хотим указывать его явно, то ему следует зависеть от типов значений, переданных в конструктор. В случае передачи объектов-строк тип должен быть `std::string`. При передаче целых чисел тип должен быть `int`. Если мы передадим целые числа, числа с плавающей точкой и числа с удвоенной точностью, то компилятору следует определить, какой тип подходит всем значениям без потери точности. Для этого мы предоставляем *явные правила выведения типов*:

```
template <typename ... Ts>
sum(Ts&& ... ts) -> sum<std::common_type_t<Ts...>>;
```

Согласно этим правилам компилятор может использовать типаж `std::common_type_t`, который способен определить, какой тип данных подходит всем значениям. Посмотрим, как его применить:

```
sum s          {1u, 2.0, 3, 4.0f};
sum string_sum {std::string{"abc"}, "def"};
std::cout << s.value          << '\n'
          << string_sum.value << '\n';
```

В первой строке мы создаем объект типа `sum` на основе аргументов конструктора, имеющих типы `unsigned`, `double`, `int` и `float`. Типаж `std::common_type_t` возвращает тип `double`, поэтому мы получаем объект типа `sum<double>`. Во второй строке

мы предоставляем экземпляр типа `std::string` и строку в стиле C. В соответствии с нашими правилами компилятор создает экземпляр типа `sum<std::string>`.

При запуске этот код выведет значение `10` как результат сложения чисел и `abcdef` в качестве результата *объединения* строк.

Упрощаем принятие решений во время компиляции с помощью constexpr-if

В коде, содержащем шаблоны, зачастую необходимо по-разному выполнять определенные действия в зависимости от типа, для которого конкретный шаблон был специализирован. В C++17 появились выражения `constexpr-if`, позволяющие *значительно* упростить написание кода в таких ситуациях.

Как это делается

В этом примере мы реализуем небольшой вспомогательный шаблонный класс. Он может работать с разными типами, поскольку способен выбирать различные пути выполнения кода в зависимости от типа, для которого мы конкретизируем шаблон.

1. Напишем обобщенную часть кода. В нашем примере рассматривается простой класс, который добавляет значение типа `U` к элементу типа `T` с помощью функции `add`:

```
template <typename T>
class addable
{
    T val;
public:
    addable(T v) : val{v} {}
    template <typename U>
    T add(U x) const {
        return val + x;
    }
};
```

2. Представим, что тип `T` — это `std::vector<что-то>`, а тип `U` — просто `int`. Каков смысл выражения «добавить целое число к вектору»? Допустим, нужно добавить данное число к каждому элементу вектора. Это делается в цикле:

```
template <typename U>
T add(U x)
{
    auto copy (val); // Получаем копию элемента вектора
    for (auto &n : copy) {
        n += x;
    }
    return copy;
       }
```

3. Следующий и последний шаг заключается в том, чтобы *объединить* оба варианта. Если `T` — это вектор, состоящий из элементов типа `U`, то выполняем *цикл*. В противном случае выполняем *обычное* сложение.

```
template <typename U>
T add(U x) const {
    if constexpr (std::is_same_v<T, std::vector<U>>) {
        auto copy (val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    } else {
        return val + x;
    }
}
```

4. Теперь класс можно использовать. Посмотрим, насколько хорошо он может работать с разными типами, такими как `int`, `float`, `std::vector<int>` и `std::vector<string>`:

```
addable<int>{1}.add(2);               // результат - 3
addable<float>{1.0}.add(2);           // результат - 3.0
addable<std::string>{"aa"}.add("bb"); // результат - "aabb"

std::vector<int> v {1, 2, 3};
addable<std::vector<int>>{v}.add(10);
    // is std::vector<int>{11, 12, 13}

std::vector<std::string> sv {"a", "b", "c"};
addable<std::vector<std::string>>{sv}.add(std::string{"z"});
    // is {"az", "bz", "cz"}
```

Как это работает

Новая конструкция `constexpr-if` работает точно так же, как и обычные конструкции `if-else`. Разница между ними заключается в том, что значение условного выражения определяется *во время компиляции*. Весь код завершения, который компилятор сгенерирует из нашей программы, не будет содержать дополнительных ветвлений, относящихся к условиям `constexpr-if`. Кто-то может сказать, что эти механизмы работают так же, как и макросы препроцессора `#if` и `#else`, предназначенные для подстановки текста, но в данном случае всему коду даже не нужно быть синтаксически правильным. Ветвления конструкции `constexpr-if` должны быть *синтаксически правильными*, но неиспользованные ветви не обязаны быть *семантически корректными*.

Чтобы определить, должен ли код добавлять значение `x` к вектору, задействуем типаж `std::is_same`. Выражение `std::is_same<A, B>::value` вычисляется в логическое значение `true`, если `A` и `B` имеют один и тот же тип. В нашем примере применяется условие `std::is_same<T, std::vector<U>>::value`, которое имеет значение

`true`, если пользователь конкретизировал шаблон для класса `T = std::vector<X>` и пробует вызвать функцию `add` с параметром типа `U = X`.

В одном блоке `constexpr-if-else` может оказаться несколько условий (обратите внимание, что `a` и `b` должны зависеть от параметров шаблона, а не только от констант времени компиляции):

```
if constexpr (a) {
    // что-нибудь делаем
} else if constexpr (b) {
    // делаем что-нибудь еще
} else {
    // делаем нечто совсем другое
}
```

С помощью C++17 гораздо легче как выразить, так и прочитать код, получающийся при метапрограммировании.

Дополнительная информация

Для того чтобы убедиться, каким прекрасным новшеством являются конструкции `constexpr-if` для C++, взглянем, как решалась та же самая задача *до* C++17:

```
template <typename T>
class addable
{
    T val;

public:
    addable(T v) : val{v} {}
    template <typename U>
    std::enable_if_t<!std::is_same<T, std::vector<U>>::value, T>
    add(U x) const { return val + x; }

    template <typename U>
    std::enable_if_t<std::is_same<T, std::vector<U>>::value,
                     std::vector<U>>
    add(U x) const {
        auto copy (val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    }
};
```

Без конструкций `constexpr-if` этот класс работает для всех необходимых нам типов, но кажется очень сложным. Как же он работает?

Сами реализации *двух разных* функций `add` выглядят просто. Все усложняет объявление возвращаемого типа — выражение наподобие `std::enable_if_t<усло-`

вие, тип> обращается в тип, если выполняется условие. В противном случае выражение std::enable_if_t ни во что не обращается. Обычно такое положение дел считается ошибкой. Далее мы рассмотрим, почему в нашем случае это не так.

Для второй функции add то же условие используется *противоположным* образом. Следовательно, условие может иметь значение true только для одной из двух реализаций в любой момент времени.

Когда компилятор видит разные шаблонные функции с одинаковым именем и должен выбрать одну из них, в ход вступает важный принцип: он обозначается аббревиатурой *SFINAE*, которая расшифровывается как *Substitution Failure is not an Error* («Сбой при подстановке — не ошибка»). В данном случае это значит, что компилятор не генерирует ошибку, если возвращаемое значение одной из функций нельзя вывести на основе неверного шаблонного выражения (то есть std::enable_if, когда условие имеет значение false). Он просто продолжит работу и попробует обработать другие реализации функции. Вот и весь секрет.

Столько возни! Радует, что после выхода C++17 делать это стало гораздо проще.

Подключаем библиотеки с помощью встраиваемых переменных

Несмотря на то что в C++ всегда была возможность определить отдельные функции как *встраиваемые*, C++17 дополнительно позволяет определять встраиваемые *переменные*. Это значительно упрощает реализацию библиотек, *размещенных в заголовочных файлах*, для чего раньше приходилось искать обходные пути.

Как это делается

В этом примере мы создаем класс-пример, который может служить членом типичной библиотеки, размещенной в заголовочном файле. Мы хотим предоставить доступ к статическому полю класса через глобально доступный элемент класса и сделать это с помощью ключевого слова inline, что до появления C++17 было невозможно.

1. Класс process_monitor должен содержать статический член и быть доступным глобально сам по себе, что приведет (при включении его в несколько единиц трансляции) к появлению символов, определенных дважды:

```
// foo_lib.hpp
class process_monitor {
public:
    static const std::string standard_string
        {"some static globally available string"};
};
process_monitor global_process_monitor;
```

2. Теперь при попытке включить данный код в несколько файлов с расширением `.cpp`, а затем скомпилировать и связать их произойдет сбой на этапе связывания. Чтобы это исправить, добавим ключевое слово `inline`:

```
// foo_lib.hpp
class process_monitor {
public:
    static const inline std::string standard_string
    {"some static globally available string"};
};
inline process_monitor global_process_monitor;
```

Вуаля! Все работает!

Как это работает

Программы, написанные на C++, зачастую состоят из нескольких исходных файлов C++ (они имеют расширения `.cpp` или `.cc`). Они отдельно компилируются в модули/объектные файлы (обычно с расширениями `.o`). На последнем этапе все эти модули/объектные файлы компонуются в один исполняемый файл или разделяемую/статическую библиотеку.

На этапе связывания ошибкой считается ситуация, когда компоновщик встречает вхождение одного конкретного символа *несколько раз*. Предположим, у нас есть функция с сигнатурой `int foo();`. Если в двух модулях определены одинаковые функции, то какую из них считать правильной? Компоновщик не может просто подбросить монетку. Точнее, может, но вряд ли хоть один программист сочтет такое поведение приемлемым.

Традиционный способ создания функций, доступных глобально, состоит в *объявлении* их в заголовочном файле, впоследствии включенном в любой модуль C++, в котором их нужно вызвать. Эти функции будут определяться в отдельных файлах модулей. Далее они связываются с теми модулями, которые должны использовать эти функции. Данный принцип также называется *правилом одного определения* (one definition rule, ODR). Взгляните на рис. 1.1, чтобы лучше понять это правило.

Однако будь это единственный способ решения задачи, нельзя было бы создавать библиотеки, размещенные в заголовочных файлах. Такие библиотеки очень удобны, поскольку их можно включить в любой файл программы C++ с помощью директивы `#include`, и они мгновенно станут доступны. Для использования же библиотек, размещенных не в заголовочных файлах, программист также должен адаптировать сценарии сборки так, чтобы компоновщик связал модули библиотек и файлы своих модулей. Это неудобно, особенно для библиотек, содержащих только очень короткие функции.

В таких случаях можно применить ключевое слово `inline` — оно позволяет в порядке исключения разрешить *повторяющиеся* определения одного символа в разных модулях. Если компоновщик находит несколько символов с одинаковой

сигнатурой, но они объявлены встраиваемыми, то он выберет первый и будет считать, что остальные символы имеют такое же определение. На программиста возложена *ответственность* за то, чтобы все одинаковые встраиваемые символы были определены абсолютно идентично.

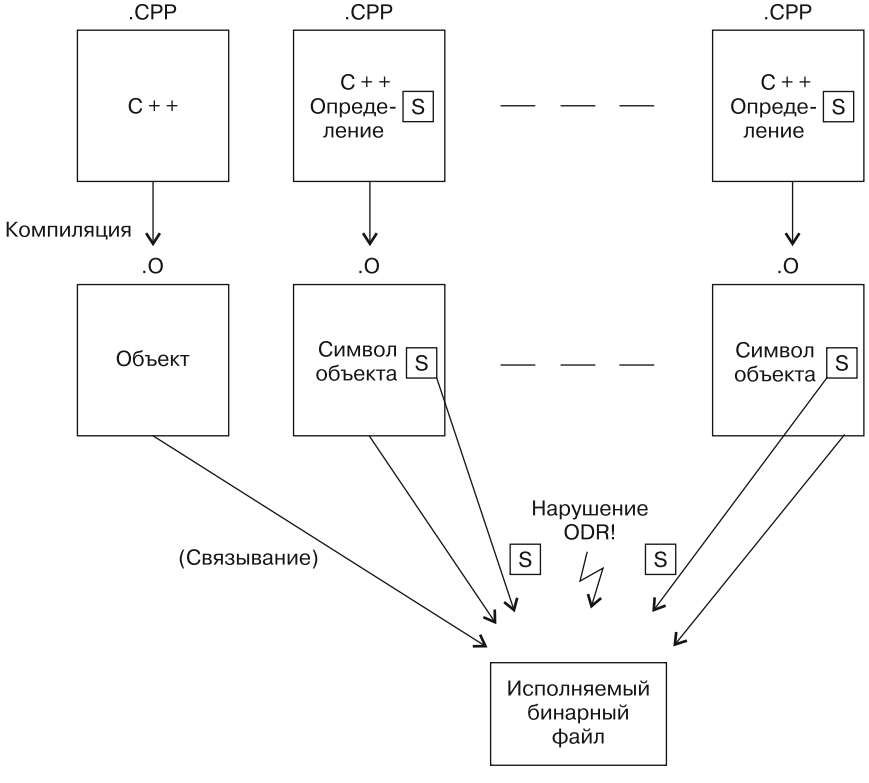

Рис. 1.1

Что касается нашего примера, компоновщик найдет символ `process_monitor::standard_string` в каждом модуле, который включает файл `foo_lib.hpp`. Без ключевого слова `inline` он не будет знать, какой символ выбрать, так что прекратит работу и сообщит об ошибке. Это же верно и для символа `global_process_monitor`. Как же выбрать правильный символ?

При объявлении обоих символов с помощью ключевого слова `inline` компоновщик просто примет первое вхождение символа и *отбросит* остальные.

До появления C++17 единственным явным способом сделать это было предоставление символа с помощью дополнительного файла модуля C++, что заставляло пользователей библиотеки включать данный файл на этапе компоновки.

Ключевое слово `inline` по традиции выполняет и *другую* задачу. Оно указывает компилятору, что он может *избавиться* от вызова функции, взяв ее реализацию и поместив в то место, из которого функция вызывается. Таким образом,

вызывающий код содержит на один вызов функции меньше — считается, что такой код работает быстрее. Если функция очень короткая, то полученный ассемблерный код также будет короче (предполагается, что количество инструкций, которые выполняют вызов функции, сохранение и восстановление стека и т. д., превышает количество строк с полезной нагрузкой). Если же встраиваемая функция очень длинная, то размер бинарного файла увеличится, а это не ускоряет работу программы. Поэтому компилятор будет использовать ключевое слово `inline` как подсказку и может избавиться от вызовов функций, встраивая их тело. Он даже может встроить отдельные функции, которые программист *не объявлял* встраиваемыми.

Дополнительная информация

Одним из способов решения такой задачи до появления C++17 было создание функции `static`, которая возвращает ссылку на объект `static`:

```
class foo {
public:
    static std::string& standard_string() {
        static std::string s {"some standard string"};
        return s;
    }
};
```

Подобным образом вы можете совершенно легально включить заголовочный файл в несколько модулей и при этом получать доступ к одному и тому же экземпляру отовсюду. Однако объект *не* создается *немедленно* при старте программы — это происходит только при первом вызове функции-геттера. В некоторых случаях это может оказаться проблемой. Представьте, будто нужно, чтобы конструктор статического объекта, доступного глобально, при *запуске программы* выполнял некую важную операцию (в точности как наш класс-пример), но мы не получаем желаемого из-за вызова геттера ближе к концу программы.

Проблему можно решить еще одним способом: сделав класс `foo` шаблонным и воспользовавшись преимуществами шаблонов.

В C++17 оба варианта становятся неактуальны.

Реализуем вспомогательные функции с помощью выражений свертки

Начиная с C++11, в языке появились пакеты параметров для шаблонов с переменным количеством аргументов. Такие пакеты позволяют реализовывать функции, принимающие переменное количество параметров. Иногда эти параметры объединяются в одно выражение, чтобы на его основе можно было получить результат работы функции. Решение этой задачи значительно упростилось с выходом C++17, где появились выражения свертки.

Как это делается

Реализуем функцию, которая принимает переменное количество параметров и возвращает их сумму.

1. Сначала определим ее сигнатуру:

```
template <typename ... Ts>
auto sum(Ts ... ts);
```

2. Теперь у нас есть пакет параметров `ts`, функция должна распаковать все параметры и просуммировать их с помощью выражения свертки. Допустим, мы хотим воспользоваться каким-нибудь оператором (в нашем случае `+`) вместе с `...`, чтобы применить его ко всем значениям пакета параметров. Для этого нужно взять выражение в скобки:

```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ...);
}
```

3. Теперь можно вызвать функцию следующим образом:

```
int the_sum {sum(1, 2, 3, 4, 5)}; // Значение: 15
```

4. Она работает не только с целочисленными типами; можно вызвать ее для любого типа, реализующего оператор `+`, например `std::string`:

```
std::string a {"Hello "};
std::string b {"World"};
std::cout << sum(a, b) << '\n'; // Вывод: Hello World
```

Как это работает

Только что мы написали код, в котором с помощью простой рекурсии бинарный оператор (`+`) применяется к заданным параметрам. Как правило, это называется *сверткой*. В C++17 появились *выражения свертки*, которые помогают выразить ту же идею и при этом писать меньше кода.

Подобное выражение называется *унарной сверткой*. C++17 позволяет применять к пакетам параметров свертки следующие бинарные операторы: `+`, `–`, `*`, `/`, `%`, `^`, `&`, `|`, `=`, `<`, `>`, `<<`, `>>`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`, `==`, `!=`, `<=`, `>=`, `&&`, `||`, `,`, `.*`, `->*`.

Кстати, в нашем примере кода неважно, какую использовать конструкцию, `(ts + …)` или `(… + ts);`. Они обе работают так, как нужно. Однако между ними есть разница, которая может иметь значение в других случаях: если многоточие ... находится с *правой* стороны оператора, то такое выражение называется *правой сверткой*. Если же оно находится с *левой* стороны, то это *левая свертка*.

В нашем примере с суммой левая унарная свертка разворачивается в конструкцию `1 + (2 + (3 + (4 + 5)))`, а правая унарная свертка развернется в `(((1 + 2) + 3) + 4) + 5`. В зависимости от того, какой оператор используется, могут проявиться нюансы. При добавлении новых чисел ничего не меняется.

Дополнительная информация

Если кто-то вызовет функцию `sum()` и *не передаст* в нее аргументы, то пакет параметров произвольной длины не будет содержать значений, которые могут быть свернуты. Для большинства операторов такая ситуация считается ошибкой (но для некоторых — нет, вы увидите это чуть позже). Далее нужно решить, генерировать ошибку или же вернуть конкретное значение. Очевидным решением будет вернуть значение `0`.

Это делается так:

```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ... + 0);
}
```

Таким образом, вызов `sum()` возвращает значение `0`, а вызов `sum(1, 2, 3)` — значение `(1 + (2 + (3 + 0)))`. Подобные свертки с начальным значением называются *бинарными*.

Кроме того, обе конструкции, `(ts + ... + 0)` и `(0 + ... + ts)`, работают как полагается, но такая бинарная свертка становится *правой* или *левой* соответственно. Взгляните на рис. 1.2.

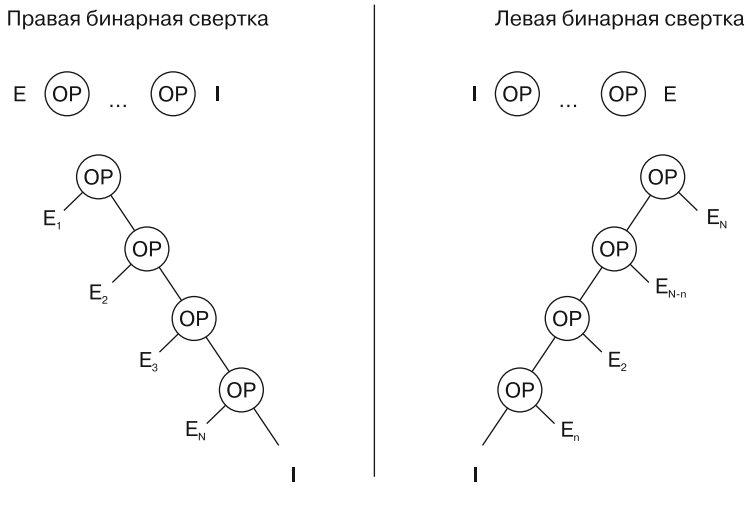

Рис. 1.2

При использовании бинарных сверток для решения такой задачи, когда аргументы отсутствуют, очень важны *нейтральные* элементы — в нашем случае сложение любого числа с нулем ничего не меняет, что делает `0` нейтральным элементом. Поэтому можно добавить `0` к любому выражению свертки с помощью

операторов + или –. Если пакет параметров пуст, это приведет к возврату функцией значения 0. С математической точки зрения это правильно. С точки зрения реализации нужно определить, что именно является правильным в зависимости от наших требований.

Тот же принцип применяется и к умножению. Здесь нейтральным элементом станет 1:

```
template <typename ... Ts>
auto product(Ts ... ts)
{
    return (ts * ... * 1);
}
```

Результат вызова product(2, 3) равен 6, а результат вызова product() без параметров равен 1.

В логических операторах И (&&) и ИЛИ (||) появились *встроенные* нейтральные элементы. Свертка пустого пакета параметров с оператором && заменяется на true, а свертка пустого пакета с оператором || — на false.

Еще один оператор, для которого определено значение по умолчанию, когда он используется для пустых пакетов параметров, — это оператор «запятая» (,), заменяемый на void().

Давайте взглянем на другие вспомогательные функции, которые можно реализовать с помощью этих механизмов.

Соотнесение диапазонов и отдельных элементов

Как насчет функции, которая определяет, содержит ли диапазон *хотя бы одно* из значений, передаваемых в пакете параметров с переменной длиной:

```
template <typename R, typename ... Ts>
auto matches(const R& range, Ts ... ts)
{
    return (std::count(std::begin(range), std::end(range), ts) + ...);
}
```

Вспомогательная функция использует функцию std::count из библиотеки STL. Она принимает три параметра: первые два представляют собой *начальный* и *конечный* итераторы того или иного итерабельного промежутка, а третий параметр — это *значение*, с которым будут сравниваться все элементы промежутка. Метод std::count возвращает количество всех элементов внутри диапазона, равных третьему параметру.

В нашем выражении свертки мы всегда передаем в функцию std::count *начальный* и *конечный* итераторы одного диапазона параметров. Однако в качестве третьего параметра мы всякий раз отправляем один параметр из пакета.

В конечном счете функция складывает все результаты и возвращает их вызывающей стороне.

Ее можно использовать следующим образом:

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

matches(v,          2, 5);          // возвращает 2
matches(v,          100, 200);      // возвращает 0
matches("abcdefg", 'x', 'y', 'z'); // возвращает 0
matches("abcdefg", 'a', 'd', 'f'); // возвращает 3
```

Как видите, вспомогательная функция `matches` довольно гибкая — ее можно вызвать для векторов или даже строк. Она также будет работать для списка инициализаторов, контейнеров `std::list`, `std::array`, `std::set` и прочих!

Проверка успешности вставки нескольких элементов в множество

Напишем вспомогательную функцию, которая добавляет произвольное количество параметров в контейнер `std::set` и возвращает булево значение, показывающее, *успешно* ли прошла операция:

```
template <typename T, typename ... Ts>
bool insert_all(T &set, Ts ... ts)
{
    return (set.insert(ts).second && ...);
}
```

Как же это работает? Функция `insert` контейнера `std::set` имеет следующую сигнатуру:

```
std::pair<iterator, bool> insert(const value_type& value);
```

Документация гласит, что при попытке вставить элемент функция `insert` вернет пару из `iterator` и переменной `bool`. Если вставка пройдет успешно, значение переменной будет равно `true`. Итератор же в этом случае укажет на *новый элемент* множества, а в противном случае — на *существующий* элемент, который *помешал* вставке.

Наша вспомогательная функция после вставки обращается к полю `.second`. Оно содержит переменную `bool`, которая показывает, была ли вставка успешной. Если все полученные пары имеют значение `true`, то все вставки прошли успешно. Свертка объединяет все результаты вставки с помощью оператора `&&` и возвращает результат.

Контейнер можно использовать следующим образом:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 5, 6); // Возвращает true
insert_all(my_set, 7, 8, 2); // Возвращает false, поскольку 2 уже присутствует
```

Обратите внимание: если мы попробуем вставить, например, три элемента, но в процессе окажется, что второй элемент вставить нельзя, свертка `&& ...` досрочно прекратит работать и оставшиеся элементы не будут добавлены:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 2, 5); // Возвращает false
// теперь множество содержит значения {1, 2, 3, 4}, без 5!
```

Проверка попадания всех параметров в заданный диапазон

Поскольку можно убедиться, что *одна* из переменных находится в конкретном диапазоне, можно сделать то же самое для *нескольких* переменных с помощью выражений свертки:

```
template <typename T, typename ... Ts>
bool within(T min, T max, Ts ...ts)
{
    return ((min <= ts && ts <= max) && ...);
}
```

Выражение `(min <= ts && ts <= max)` определяет, находится ли каждый элемент пакета параметров в диапазоне между `min` и `max` (*включая* `min` и `max`). Мы выбрали оператор `&&`, чтобы свести все результаты булева типа к одному, который имеет значение `true` только в том случае, если все отдельные результаты имеют такое же значение.

Это работает следующим образом:

```
within( 10,  20,  1, 15, 30);    // --> false
within( 10,  20,  11, 12, 13);   // --> true
within(5.0, 5.5,  5.1, 5.2, 5.3) // --> true
```

Что интересно: эта функция очень гибкая, поскольку единственным требованием, которое она предъявляет к типам, служит *возможность сравнения* экземпляров с помощью оператора `<=`. Это требование выполняется, например, типом `std::string`:

```
std::string aaa {"aaa"};
std::string bcd {"bcd"};
std::string def {"def"};
std::string zzz {"zzz"};

within(aaa, zzz,  bcd, def); // --> true
within(aaa, def,  bcd, zzz); // --> false
```

Отправка нескольких элементов в вектор

Кроме того, вы можете написать вспомогательную функцию, которая не обобщает никаких результатов, но обрабатывает несколько действий одного вида. Такими действиями могут быть вставки элементов в контейнер `std::vector`, поскольку они

не возвращают никаких результатов (функция `std::vector::insert()` сообщает об ошибке, генерируя исключения):

```
template <typename T, typename ... Ts>
void insert_all(std::vector<T> &vec, Ts ... ts)
{
    (vec.push_back(ts), ...);
}

int main()
{
    std::vector<int> v {1, 2, 3};
    insert_all(v, 4, 5, 6);
}
```

Обратите внимание: мы используем оператор «запятая» (`,`), чтобы распаковать пакет параметров в отдельные вызовы `vec.push_back(...)`, не выполняя свертку для самого результата. Эта функция также хорошо работает в отношении *пустого* пакета параметров, поскольку оператор «запятая» имеет неявный нейтральный элемент, `void()`, который означает *«ничего не делать»*.

2 Контейнеры STL

В этой главе:

- использование идиомы erase-remove для контейнера `std::vector`;
- удаление элементов из неотсортированного контейнера `std::vector` за время *O(1)*;
- получение доступа к экземплярам класса `std::vector` быстрым или безопасным способом;
- поддержка экземпляров класса `std::vector` в отсортированном состоянии;
- вставка элементов в контейнер `std::map`: эффективно и в соответствии с условиями;
- исследование новой семантики подсказок для вставки элементов с помощью метода `std::map::insert`;
- эффективное изменение ключей элементов `std::map`;
- применение контейнера `std::unordered_map` для пользовательских типов;
- отбор повторно встречающихся слов из пользовательского ввода и вывод их на экран в алфавитном порядке с помощью контейнера `std::set`;
- реализация простого ОПН-калькулятора с использованием контейнера `std::stack`;
- подсчет частоты встречаемости слов с применением контейнера `std::map`;
- реализация вспомогательного инструмента для поиска очень длинных предложений в текстах с помощью `std::multimap`;
- реализация личного списка текущих дел с помощью `std::priority_queue`.

Введение

В стандартной библиотеке C++ появилось большое количество стандартных контейнеров. Контейнер всегда содержит набор данных или объектов. Достоинство контейнеров в том, что их можно применять практически для всех объектов, поэтому нужно только выбрать правильные контейнеры для конкретного приложения. STL предоставляет стеки, автоматически увеличивающиеся векторы,

ассоциативные массивы и т. д. Таким образом, можно сконцентрироваться на нашем приложении и не изобретать велосипед. В целом каждому программисту C++ не повредит знакомство со всеми контейнерами.

Все контейнеры, предоставляемые STL, можно разделить на такие категории, которые подробнее рассматриваются в следующем разделе:

- непрерывные хранилища;
- списки;
- деревья поиска;
- хеш-таблицы;
- адаптеры контейнеров.

Рассмотрим более подробно каждый из пунктов.

Непрерывные хранилища

Самый простой способ хранения объектов — поместить их рядом друг с другом в одном большом фрагменте памяти. Произвольный доступ к такому фрагменту выполняется за время *O(1)*.

Это проще всего сделать так: воспользоваться контейнером `std::array` (он представляет собой обертку для обычных массивов в стиле C). Вам практически всегда следует выбирать их вместо обычных массивов, поскольку это не потребует никаких усилий, но работать станет комфортнее и безопаснее. Как и в случае с обычными массивами, массивы STL имеют *фиксированный* размер, определяемый при создании.

Контейнер `std::vector` вступает в дело, когда вам нужно хранилище, похожее на массив, но с изменяемой длиной. Он использует память из кучи для хранения объектов. Если при добавлении элемента в вектор происходит превышение размера вектора, то все элементы автоматически перемещаются в более крупный фрагмент вновь выделенной памяти, старый же фрагмент удаляется. Более того, если новый элемент помещается между двумя старыми, то может даже перемещать существующие элементы в памяти. При удалении элемента из середины вектора класс `vector` автоматически сдвинет оставшиеся элементы так, чтобы закрыть получившуюся дыру.

Добавление (или удаление) в начало (или конец) контейнера `std::vector` сразу многих объектов может повлечь выполнение большого количества операций выделения памяти для получения пространства, а также потенциально затратных операций по перемещению объектов. В таких ситуациях стоит воспользоваться контейнером `std::deque` (deque («дек») расшифровывается как *double-ended queue* (двусторонняя очередь)). Объекты хранятся в непрерывных фрагментах памяти, которые не зависят друг от друга. Это позволяет быстро увеличивать размер дека, поскольку объекты, расположенные в таких фрагментах памяти, не будут перемещаться, когда выделяется память для нового фрагмента и размещается в начале или конце контейнера.

Хранение списков

Контейнер `std::list` представляет собой классический двухсвязный список. Ни больше ни меньше. Если вам нужно выполнять обход списка только в одном направлении, то больше подойдет контейнер `std::forward_list` — он быстрее работает и занимает меньше места, поскольку в нем хранятся лишь указатели на следующий элемент. Пройти список можно только последовательно, за время *O(n)*. Вставку и удаление элементов в заданную позицию можно выполнить за время *O(1)*.

Деревья поиска

Если для объектов характерен естественный порядок, такой, что их можно отсортировать с использованием математического отношения `<`, то их можно поддерживать в этом же порядке с помощью *деревьев поиска*. Из названия следует: конкретный элемент дерева легко находится благодаря ключу поиска, что дает возможность выполнять эту операцию за время *O(log(n))*.

В STL есть несколько видов таких деревьев, самым простым является `std::set` — в нем хранятся *уникальные* сортируемые объекты.

Контейнер `std::map` отличается тем, что данные в нем хранятся *парами*. Пара состоит из *ключа* и *значения*. Дерево поиска использует ключ для сортировки элементов, таким образом можно задействовать `std::map` в качестве *ассоциативного контейнера*. Как и в случае с контейнером `std::set`, все ключи в дереве должны быть в единственном экземпляре.

Контейнеры `std::multiset` и `std::multimap` являются частными случаями контейнеров, показанных выше, для которых отсутствует требование *уникальности* ключей.

Хеш-таблицы

Деревья поиска не единственный способ реализации ассоциативных контейнеров. В *хеш-таблицах* элементы можно найти за время *O(1)*, но они игнорируют естественный порядок элементов, поэтому не так просто пройти по упорядоченному списку. Пользователь может изменять *размеры* хеш-таблицы, можно даже выбрать отдельно хеш-функцию, что немаловажно, поскольку от этого зависят производительность и потребление памяти.

Контейнеры `std::unordered_set` и `std::unordered_map` весьма похожи на `std::set` и `std::map`, что можно утверждать на основании их взаимозаменяемости.

Как и реализации поисковых деревьев, у обоих контейнеров есть «коллеги» `std::unordered_multiset` и `std::unordered_multimap`, которые не имеют ограничения на уникальность объектов/ключей, поэтому можно хранить несколько элементов для одного ключа.

Адаптеры контейнеров

Массивы, списки, деревья и хеш-таблицы не единственный способ хранения данных и получения к ним доступа. Еще есть стеки, очереди и т. д. Аналогично тому, как более сложные структуры могут быть реализованы с использованием более примитивных, в STL такие структуры создаются благодаря следующим классам — *адаптерам* контейнеров: `std::stack`, `std::queue` и `std::priority_queue`.

Приятная новость: если нам нужна подобная структура данных, можно просто выбрать адаптер. Далее, понимая, что нас не устраивает производительность, мы можем просто изменить параметр шаблона, чтобы адаптер выбрал другую реализацию контейнера. На практике это значит, например, возможность замены находящейся в основе экземпляра `std::stack` реализации с `std::vector` на `std::deque`.

Используем идиому erase-remove для контейнера std::vector

Многие программисты-новички узнают о существовании класса `std::vector`, который в основном работает как *автоматически увеличивающийся массив*, и останавливаются на этом. Далее они лишь ищут в документации сведения о том, как решать конкретные задачи, например *удалять* элементы. Такого рода использование контейнеров STL не задействует и сотой доли их возможностей в создании *чистого*, *удобного в сопровождении* и *быстрого* кода.

Текущий раздел посвящен вопросу удаления элементов из середины объекта вектора. Если элемент исчезает из вектора и при этом он находился *между* другими элементами, то все элементы, стоящие справа от него, должны *сместиться* на одну позицию *влево* (в результате время выполнения операции составляет $O(n)$). Многие начинающие программисты будут решать задачу с помощью *цикла*, ведь это так просто. К сожалению, при этом они, скорее всего, упустят потенциальные возможности оптимизации. В конечном счете цикл, написанный вручную, не *быстрее* и не *читабельнее*, чем способ решения задачи средствами STL, которые мы еще рассмотрим далее.

Как это делается

В этом примере мы наполним объект класса `std::vector` целыми числами, а затем выбросим из него конкретные элементы. При этом мы воспользуемся *корректным* способом удаления нескольких элементов из вектора.

1. Конечно, прежде чем начинать что-то делать, включим некоторые заголовочные файлы:

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

2. Далее мы объявляем, что будем использовать пространство имен `std`, чтобы поменьше печатать:

```
using namespace std;
```

3. Теперь создадим вектор, состоящий из целых чисел, и заполним его некими числами:

```
int main()
{
    vector<int> v {1, 2, 3, 2, 5, 2, 6, 2, 4, 8};
```

4. Удалим некоторые элементы. Что именно? В нашем примере много значений `2`, удалим их:

```
    const auto new_end (remove(begin(v), end(v), 2));
```

5. Что интересно, это был лишь первый шаг. Вектор все еще имеет прежний размер. Мы укоротим его по-настоящему, написав такую строку:

```
    v.erase(new_end, end(v));
```

6. Остановимся и выведем на экран содержимое вектора, а затем продолжим:

```
    for (auto i : v) {
        cout << i << ", ";
    }
    cout << '\n';
```

7. Теперь удалим целый *класс* элементов, а не конкретные *значения*. Для этого сначала определим функцию-предикат, которая принимает число в качестве параметра и возвращает значение `true`, если переданное число *нечетное*:

```
    const auto odd ([](int i) { return i % 2 != 0; });
```

8. Используем функцию `remove_if`, передавая ей значения с помощью функции-предиката. Вместо удаления элементов за два шага теперь делаем это за один:

```
    v.erase(remove_if(begin(v), end(v), odd), end(v));
```

9. Мы удалили все нечетные элементы, но *емкость* вектора все еще соответствует старым десяти элементам. На последнем шаге мы сократим эту емкость до *реального* размера вектора. Обратите внимание: это может привести к тому, что код обслуживания вектора выделит новый фрагмент памяти, который будет иметь соответствующий размер, и переместит все элементы из старого фрагмента в новый:

```
    v.shrink_to_fit();
```

10. Теперь выведем на экран содержимое вектора после второго раунда удаления элементов и закончим с примером:

```
    for (auto i : v) {
        cout << i << ", ";
    }
    cout << '\n';
}
```

11. Скомпилировав и запустив программу, вы увидите следующие строки, показывающие результат выполнения операций по удалению элементов:

```
$ ./main
1, 3, 5, 6, 4, 8,
6, 4, 8,
```

Как это работает

Из данного примера стало очевидно: при удалении элементов из середины вектора их нужно сначала *убрать*, а затем *стереть*. По крайней мере функции, которые мы использовали, имеют соответствующие имена. Это может запутать, но рассмотрим их подробнее, чтобы разобраться в том, что происходит на каждом шаге.

Код, удаляющий все значения `2` из вектора, выглядел так:

```
const auto new_end (remove(begin(v), end(v), 2));
v.erase(new_end, end(v));
```

Функции `std::begin` и `std::end` принимают в качестве параметра экземпляр вектора и соответственно возвращают итераторы, которые указывают на *первый* элемент и на позицию, находящуюся *после последнего* элемента, как показано на рис. 2.1.

Рис. 2.1

После того как мы передадим их и значение `2` функции `std::remove`, она переместит все значения, кроме 2, вперед точно так же, как если бы мы сделали это вручную с помощью цикла. С первого взгляда рисунок может показаться непонятным. На шаге 2 все еще можно найти значение `2`, а сам вектор должен был стать короче, поскольку содержал четыре таких значения. Вместо этого значе-

ния 4 и 8, присутствовавшие в оригинальном массиве, встречаются дважды. Что происходит?

Рассмотрим только элементы, находящиеся в рамках диапазона от итератора begin, показанного на рис. 2.1, до итератора new_end. Элемент, на который указывает итератор new_end, является *первым элементом за пределами* диапазона, поэтому он не включается. Сконцентрировавшись на заданном участке (в нем содержатся элементы от 1 до 8 включительно), мы понимаем, что *перед нами корректный* диапазон, из которого удалены все значения 2.

Здесь вступает в дело функция erase: мы должны указать вектору, что все элементы между итераторами new_end и end больше к нему не относятся. Вектор легко с этим справится, поскольку может просто перевести конечный итератор в позицию, обозначенную new_end. Обратите внимание: итератор new_end является возвращаемым значением вызова std::remove, следовательно, можно просто им воспользоваться.

Заметьте: вектор выполнил больше манипуляций, нежели просто передвинул внутренний указатель. Если бы вектор состоял из более сложных объектов, то ему пришлось бы вызвать деструкторы для всех удаляемых элементов.

В конечном счете вектор выглядит так, как показано в шаге 3: считается, что его размер *уменьшился*. Старые элементы, лежащие вне диапазона, *все еще* находятся *в памяти*.

Чтобы вектор занимал ровно столько памяти, сколько ему нужно, в конце работы мы вызываем метод shrink_to_fit. Во время этого вызова выделяется ровно необходимый объем памяти, все элементы перемещаются и освобождается более крупный фрагмент памяти, который уже не нужен.

В шаге 8 мы определили функцию-*предикат* и использовали ее вместе с функцией std::remove_if. Это работает, поскольку независимо от того, какой итератор вернет функция, его можно будет безопасно применить в функции вектора erase. Если мы *не найдем нечетных элементов*, то функция std::remove_if не выполнит никаких действий и вернет конечный итератор. Далее вызов наподобие v.erase(end, end); также ни к чему не приведет.

Дополнительная информация

Функция std::remove работает и для других контейнеров. При ее вызове для std::array обратите внимание, что массив не поддерживает вызов функции erase из второго шага, поскольку вы не можете изменять его размер автоматически. Несмотря на то что функция std::remove, по сути, лишь перемещает элементы и не удаляет их, она пригодна для структур данных наподобие массивов, которые не могут изменять размер. При работе с массивом можно переписать значения после конечного итератора некоторыми граничными значениями, например '\0' для строк.

Удаляем элементы из неотсортированного объекта класса std::vector за время O(1)

Удаление элементов из середины вектора занимает *O(n)* времени. При этом образовавшийся промежуток должен быть заполнен: все элементы, стоящие после него, перемещаются на одну позицию влево.

Такое перемещение с сохранением порядка может оказаться затратным по времени, если перемещаемые элементы сложны и/или велики и содержат много объектов. Если порядок сохранять не требуется, можно оптимизировать процесс, как показано в этом разделе.

Как это делается

В этом примере мы заполним экземпляр класса `std::vector` некими числами, а затем реализуем функцию быстрого удаления, которая удаляет любой элемент из вектора за время *O(1)*.

1. Сначала включим необходимые заголовочные файлы:

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

2. Далее определим функцию `main`, где создадим вектор, заполненный числами:

```
int main()
{
    std::vector<int> v {123, 456, 789, 100, 200};
```

3. Очередной шаг заключается в том, чтобы удалить значение с индексом `2` (отсчет начинается с нуля, следовательно, это будет третье число, `789`). Функция, которую мы будем использовать для данной задачи, еще не реализована. Мы сделаем это спустя несколько шагов. Затем выведем содержимое вектора:

```
    quick_remove_at(v, 2);
    for (int i : v) {
       std::cout << i << ", ";
    }
    std::cout << '\n';
```

4. Теперь удалим еще один элемент. Это будет значение `123`. Предположим, что не знаем его индекс. Как следствие, нужно вызвать функцию `std::find`, которая принимает диапазон (наш вектор) и значение, а затем ищет позицию значения. После этого она возвращает *итератор*, указывающий на значение `123`. Мы используем его для той же функции `quick_remove_at`, но на сей раз применим *перегруженную* версию *предыдущей*, которая принимает в качестве параметров *итераторы*. Данная функция также не реализована.

```
quick_remove_at(v, std::find(std::begin(v), std::end(v), 123));
for (int i : v) {
```

```
        std::cout << i << ", ";
    }
    std::cout << '\n';
}
```

5. Нам осталось реализовать только две функции `quick_remove_at`. Этим и займемся. (Обратите внимание: они должны быть как минимум объявлены до функции `main`. Так что просто определим их там.)

   Обе функции принимают ссылку на вектор, содержащий значения *некоторого* типа (в нашем случае это `int`), так что мы не решаем за пользователя, какой тип вектора он задействует. Для нас это будет вектор, содержащий значения типа `T`. Первая функция `quick_remove_at` принимает значения *индекса*, которые представляют собой *числа*, вследствие чего ее интерфейс выглядит следующим образом:

```
template <typename T>
void quick_remove_at(std::vector<T> &v, std::size_t idx)
{
```

6. Сейчас перейдем к самому главному — быстрому удалению элементов. При этом нужно постараться не перемещать слишком большого количества оставшихся элементов. Во-первых, просто возьмем значение последнего элемента вектора и запишем его на место элемента, который должен быть удален. Во-вторых, отбросим последний элемент. Вот и все, только два шага. Мы добавим в этот код небольшую проверку безопасности. Если значение индекса очевидным образом выходит за пределы вектора, мы не станем ничего делать. В противном случае код будет давать сбой для пустого вектора.

```
    if (idx < v.size()) {
        v[idx] = std::move(v.back());
        v.pop_back();
    }
}
```

7. Другая реализация метода `quick_remove_at` работает аналогично. Вместо того чтобы принимать численный индекс, она принимает итератор для вектора `std::vector<T>`. Получить такой обобщенный тип несложно, поскольку для контейнеров STL уже определены подобные типы.

```
template <typename T>
void quick_remove_at(std::vector<T> &v,
                     typename std::vector<T>::iterator it)
{
```

8. Теперь получим доступ к значению, на которое указывает итератор. Как и в другой функции, перепишем его с помощью последнего элемента вектора. Поскольку мы работаем не с числовым индексом, а с итератором, следует выполнять более аккуратную проверку на безопасность. Если итератор указывает на специальную конечную позицию, то разыменовывать его нельзя.

```
    if (it != std::end(v)) {
```

9. Внутри этого блока `if` делаем то же самое, что и раньше: переписываем значение удаляемого элемента с последней позиции, а затем отбрасываем последний элемент вектора:

```
            *it = std::move(v.back());
            v.pop_back();
        }
    }
```

10. Вот и все. Компиляция и запуск программы дадут следующий результат:

```
$ ./main
123, 456, 200, 100,
100, 456, 200,
```

Как это работает

Функция `quick_remove_at` довольно быстро удаляет элементы и затрагивает не слишком много других элементов. Это относительно хитроумно. По сути, она ставит *последний* элемент вектора на место *удаляемого* элемента. Несмотря на то что последний элемент *не связан* с выбранным элементом, он находится *в особой позиции*: удаление последнего элемента *занимает мало времени*! Вам нужно лишь сократить размер вектора на одну позицию, и на этом все. На данном шаге элементы не перемещаются. Представить происходящее поможет рис. 2.2.

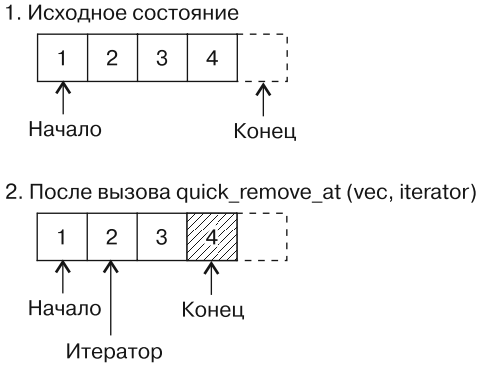

Рис. 2.2

Оба шага в коде примера выглядят следующим образом:

```
v.at(idx) = std::move(v.back());
v.pop_back();
```

В версии с итераторами шаги выглядят примерно так же:

```
*it = std::move(v.back());
v.pop_back();
```

По логике, мы *меняем местами* выбранный и последний элементы. Но в коде мы не делаем этого, а лишь *перемещаем* последний элемент на место выбранного.

Почему? Если бы мы действительно меняли местами элементы, нам пришлось бы сохранить выбранный элемент во *временной* переменной, переместить последний элемент на место выбранного, а затем сохранить временное значение в последней позиции. Это выглядит *бессмысленно*, поскольку мы все равно будем *удалять* последний элемент.

Ладно, значит, менять элементы местами бесполезно, и односторонняя перезапись нам больше подходит. Убедившись в этом, мы можем решить, что такое действие можно выполнить просто с помощью операции `*it = v.back();`, верно? Да, совершенно верно. Но представьте хранение на каждой позиции очень больших строк или даже другого вектора или ассоциативного массива — в такой ситуации эта небольшая операция присваивания приведет к очень затратной операции копирования. Вызов `std::move`, вставленный посередине (между `=` и `v.back()`), ускоряет выполнение кода. В случае со строками элемент вектора указывает на большую строку в куче. Нам не нужно ее копировать. Вместо этого при *перемещении* строки адрес исходной строки меняется на *адрес места назначения*. Исходный элемент остается прежним, но сам по себе он бесполезен, что нас устраивает, поскольку мы все равно его удаляем.

Получаем доступ к экземплярам класса std::vector быстрым или безопасным способом

Контейнер `std::vector`, возможно, является наиболее широко используемым контейнером STL, поскольку хранит данные аналогично массивам, но работать с ним гораздо удобнее. Однако неверное обращение к элементам вектора может быть опасным. Если вектор содержит 100 элементов, а наш код пытается получить доступ к элементу с индексом 123, то это, очевидно, плохо. Такая программа в лучшем случае даст сбой, поскольку подобное поведение явно указывает на ошибку в коде! Если программа не дает сбой, то мы наверняка заметим ее *странное* поведение, что куда хуже, чем аварийно завершившая программа. Опытный программист может добавить проверки перед непосредственным обращением к элементу вектора по индексу. Такие проверки не повышают читабельность кода, и многие разработчики не знают, что контейнер `std::vector` уже поддерживает подобные встроенные проверки!

Как это делается

В этом примере мы попытаемся двумя способами получить доступ к элементу контейнера `std::vector`, а затем рассмотрим, как можно применить их для написания более безопасных программ, не снижая читаемости кода.

1. Включим все необходимые заголовочные файлы и заполним вектор тысячей значений `123`, чтобы нам было с чем работать:

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

using namespace std;
```

```
int main()
{
    const size_t container_size {1000};
    vector<int> v (container_size, 123);
```

2. Теперь обратимся к элементу, лежащему за пределами вектора, с помощью оператора `[]`:

```
    cout << "Out of range element value: "
         << v[container_size + 10] << '\n';
```

3. Далее обратимся к элементу, лежащему за пределами вектора, с помощью функции `at`:

```
    cout << "Out of range element value: "
         << v.at(container_size + 10) << '\n';
}
```

4. Запустим программу и посмотрим, что произойдет. Сообщение об ошибке характерно для GCC. Прочие компиляторы отправят другие, но аналогичные сообщения. Первая операция чтения будет выполнена успешно, но странным образом. Она не заставит программу дать сбой, но мы получим *значение*, отличное от `123`. Мы не можем увидеть результат второго обращения, поскольку оно намеренно сгенерировало сбой. Второй подход позволяет гораздо раньше выявить случайные выходы за границы контейнера.

```
Out of range element value: -726629391
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 1010) >= _Nm (which is 1000)
Aborted (core dumped)
```

Как это работает

Контейнер `std::vector` предоставляет оператор `[]` и функцию `at`, и они, по сути, делают одинаковую работу. Однако функция выполняет дополнительные проверки границ и генерирует *исключение*, если мы вышли за границы вектора. Такое свойство очень полезно в ситуациях вроде нашей, но работа программы при этом несколько замедляется.

Оператор `[]` полезен при выполнении вычислений, для которых нужно очень быстро обращаться к проиндексированным элементам вектора. В любой другой ситуации функция `at` помогает определять ошибки, при этом вы почти не теряете в производительности.

Широко практикуется использование функции at по умолчанию. Если полученный код слишком медленный, но при этом безошибочный, то вместо данной функции можно задействовать оператор [] в тех местах, где важна высокая производительность.

Дополнительная информация

Конечно, можно *обработать* ситуацию выполнения доступа к элементу, лежащему за пределами вектора, вместо того чтобы *прерывать* работу программы. Для ее обработки нужно *перехватить* исключение, которое в нашем случае будет сгенерировано функцией `at`. Сделать это нетрудно. Мы окружим вызов функции `at` блоком `try` и определим код обработки ошибки в блоке `catch`:

```
try {
    std::cout << "Out of range element value: "
              << v.at(container_size + 10) << '\n';
} catch (const std::out_of_range &e) {
     std::cout << "Ooops, out of range access detected: "
               << e.what() << '\n';
}
```

Кстати говоря, контейнер std::array также предоставляет функцию at.

Сохраняем сортировку экземпляров класса std::vector

Массивы и векторы не сортируются самостоятельно. Если нам потребуется такая возможность, мы всегда можем воспользоваться структурами данных, которые предоставляют ее автоматически. Контейнер `std::vector` идеально подходит для нашего случая, ведь добавлять в него новые элементы *в порядке сортировки* несложно и удобно.

Как это делается

В этом примере мы заполним контейнер `std::vector` случайными словами, отсортируем их, а затем вставим дополнительные слова с учетом сортировки.

1. Сначала включим все необходимые заголовочные файлы:

   ```
   #include <iostream>
   #include <vector>
   #include <string>
   #include <algorithm>
   #include <iterator>
   #include <cassert>
   ```

2. Кроме того, объявим пространство имен `std`, чтобы не писать префиксы `std::`:

   ```
   using namespace std;
   ```

3. Далее напишем небольшую функцию `main`, в которой вектор заполняется случайными строками:

```
int main()
{
    vector<string> v {"some", "random", "words",
                      "without", "order", "aaa",
                      "yyy"};
```

4. Затем *отсортируем* вектор. Для этого воспользуемся некоторыми утверждениями и функцией `is_sorted` из STL, показывающей, что изначально вектор *не был* отсортирован, а теперь все его элементы *расположены по порядку*:

```
    assert(false == is_sorted(begin(v), end(v)));
    sort(begin(v), end(v));
    assert(true == is_sorted(begin(v), end(v)));
```

5. Наконец, добавим случайные слова в отсортированный вектор с помощью новой функции `insert_sorted`, которую будем реализовывать далее. Эти слова сразу нужно помещать в правильную позицию, поэтому вектор останется отсортированным:

```
    insert_sorted(v, "foobar");
    insert_sorted(v, "zzz");
```

6. Теперь реализуем функцию `insert_sorted` и расположим ее перед функцией `main`:

```
void insert_sorted(vector<string> &v, const string &word)
{
    const auto insert_pos (lower_bound(begin(v), end(v), word));
    v.insert(insert_pos, word);
}
```

7. Теперь вернемся в функцию `main` — туда, где мы остановились, — и продолжим работу, выведя содержимое вектора и увидев, что процедура вставки отработала:

```
    for (const auto &w : v) {
        cout << w << " ";
    }
    cout << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат:

```
aaa foobar order random some without words yyy zzz
```

Как это работает

Вся программа построена вокруг функции `insert_sorted`, решающей задачу, которой посвящен этот раздел. Для каждой новой строки эта функция определяет позицию в отсортированном векторе (куда нужно произвести вставку), позволя-

ющую *сохранить* порядок строк в векторе. Однако мы предполагаем, что вектор был отсортирован заранее. Иначе этот подход не сработает.

Позиция определяется с помощью функции STL `lower_bound`, принимающей три аргумента. Первые два из них указывают на *начало* и *конец* диапазона. В нашем случае таковым является вектор слов. Третий аргумент — вставляемое слово. Функция находит первый элемент диапазона, который *больше ее третьего параметра или равен ему*, и возвращает итератор, указывающий на него.

Определив правильную позицию, мы передаем ее методу `insert` контейнера `std::vector`, который принимает всего два аргумента. Первый аргумент — итератор, указывающий на позицию в векторе, в которую будет вставлен второй параметр. Очень удобно, что можно использовать итератор, возвращаемый функцией `lower_bound`. Второй аргумент — это, конечно же, вставляемый элемент.

Дополнительная информация

Функция `insert_sorted` довольно универсальна. Если мы обобщим типы ее параметров, то она будет работать и с другими типами содержимого контейнеров, и даже для других контейнеров, таких как `std::set`, `std::deque`, `std::list` и т. д.! (Обратите внимание: контейнер `set` имеет собственную функцию-член `lower_bound`, которая делает то же самое, что и функция `std::lower_bound`, но более эффективно, поскольку создана специально для множеств.)

```
template <typename C, typename T>
void insert_sorted(C &v, const T &item)
{
    const auto insert_pos (lower_bound(begin(v), end(v), item));
    v.insert(insert_pos, item);
}
```

При попытке изменить тип контейнера, приведенный в примере, с `std::vector` на что-то еще нужно учитывать следующий факт: не все контейнеры поддерживают функцию `std::sort`. Этот алгоритм предполагает использование контейнеров с произвольным доступом. Таковым, например, не является контейнер `std::list`.

Вставляем элементы в контейнер std::map эффективно и в соответствии с условиями

Иногда нужно заполнить ассоциативный массив парами «ключ — значение», и при выполнении данной задачи можно столкнуться с двумя ситуациями.

1. Заданный ключ не существует. В этом случае создается *новая* пара «ключ — значение».
2. Заданный ключ уже существует. В такой ситуации берем *существующий* элемент и *модифицируем* его.

Конечно, мы могли бы просто воспользоваться методами `insert` или `emplace` контейнера `map` и проверить успешность их выполнения. Или же нужно модифицировать существующий элемент. В обоих вышеперечисленных случаях функции `insert` и `emplace` создают элемент, который мы пытаемся вставить, а во втором случае только что созданный элемент отбрасывается. В обоих случаях совершается бесполезный вызов конструктора.

Начиная с C++17, в STL появилась функция `try_emplace`, позволяющая создавать элементы только при условии, что мы выполняем именно вставку. Реализуем программу, принимающую список миллиардеров и создающую ассоциативный массив, в котором указывается количество миллиардеров в каждой стране. Наш пример не содержит элементов, на создание которых тратится много времени, и если мы окажемся в подобной ситуации в рамках реального проекта, то сможем без труда разрешить ее с помощью `try_emplace`.

Как это делается

В этом примере мы реализуем приложение, которое создает ассоциативный массив на основе списка миллиардеров. В нем отображаются названия стран и ссылки на самого богатого человека, проживающего в данной стране, а также счетчик, указывающий количество миллиардеров.

1. Как всегда, включим некоторые заголовочные файлы; объявим также об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <functional>
#include <list>
#include <map>

using namespace std;
```

2. Определим структуру, которая представляет миллиардеров из нашего списка:

```
struct billionaire {
    string name;
    double dollars;
    string country;
};
```

3. В функции `main` сначала определяем список миллиардеров. В мире существует множество миллиардеров, поэтому создадим краткий список, включающий самых богатых людей из отдельных стран. Этот список заранее упорядочен. Он основан на списке *The World's Billionaires* («Миллиардеры мира»), создаваемом журналом *Forbes* и находящемся по адресу https://www.forbes.com/billionaires/list/.

```
int main()
{
    list<billionaire> billionaires {
        {"Bill Gates", 86.0, "USA"},
        {"Warren Buffet", 75.6, "USA"},
        {"Jeff Bezos", 72.8, "USA"},
```

```
        {"Amancio Ortega", 71.3, "Spain"},
        {"Mark Zuckerberg", 56.0, "USA"},
        {"Carlos Slim", 54.5, "Mexico"},
        // ...
        {"Bernard Arnault", 41.5, "France"},
        // ...
        {"Liliane Bettencourt", 39.5, "France"},
        // ...
        {"Wang Jianlin", 31.3, "China"},
        {"Li Ka-shing", 31.2, "Hong Kong"}
        // ...
    };
```

4. Теперь определим ассоциативный массив. В нем строка, содержащая страну, соотносится с парой. Пара включает неизменяемую копию первого миллиардера из каждой страны из нашего списка. Эти люди являются самыми богатыми жителями своей страны. Другая переменная, входящая в пару, — счетчик, который будет увеличиваться на каждого последующего миллиардера в стране:

```
    map<string, pair<const billionaire, size_t>> m;
```

5. Теперь пройдем по списку и попробуем для каждой страны заменить соответствующее значение новой парой. Пара включает ссылку на миллиардера, которого мы просматриваем в данный момент, и значение счетчика, равное `1`:

```
    for (const auto &b : billionaires) {
        auto [iterator, success] = m.try_emplace(b.country, b, 1);
```

6. При успешном выполнении данного шага не нужно больше ничего делать. Пара, для которой мы предоставили аргументы конструктора `b, 1`, была создана и вставлена в ассоциативный массив. Если вставка завершилась *неудачно*, поскольку страна-ключ уже существует, то пара не создавалась. Очень большой размер нашей структуры, описывающей миллиардера, сэкономит время, которое пришлось бы потратить на ее копирование.

   Однако в случае неудачи нам все еще нужно увеличить счетчик для заданной страны:

```
        if (!success) {
            iterator->second.second += 1;
        }
    }
```

7. На этом все. Теперь можно вывести на экран точное количество миллиардеров, живущих в стране, а также имя самого богатого человека для каждой из них:

```
    for (const auto & [key, value] : m) {
        const auto &[b, count] = value;
        cout << b.country << " : " << count
             << " billionaires. Richest is "
             << b.name << " with " << b.dollars
             << " B$\n";
    }
}
```

8. Компиляция и запуск программы дадут следующий результат. (Конечно, он будет неполным, поскольку мы ограничили наш входной ассоциативный массив.)

```
$ ./efficient_insert_or_modify
China : 1 billionaires. Richest is Wang Jianlin with 31.3 B$
France : 2 billionaires. Richest is Bernard Arnault with 41.5 B$
Hong Kong : 1 billionaires. Richest is Li Ka-shing with 31.2 B$
Mexico : 1 billionaires. Richest is Carlos Slim with 54.5 B$
Spain : 1 billionaires. Richest is Amancio Ortega with 71.3 B$
USA : 4 billionaires. Richest is Bill Gates with 86 B$
```

Как это работает

Весь пример строится на функции `try_emplace` контейнера `std::map`, которая появилась в C++17. Она имеет следующую сигнатуру:

```
std::pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
```

Вставляемый ключ — параметр `k`, а связанное значение создается на основе набора параметров `args`. При успешной вставке элемента функция вернет *итератор*, указывающий на новый узел ассоциативного массива, *объединенный в пару* со значением `true` логического типа. В противном случае булева переменная в паре будет иметь значение `false`, а итератор станет указывать на элемент, с которым пересекается вставляемый элемент.

В нашей ситуации описанная характеристика очень полезна: когда мы видим миллиардера из конкретной страны в первый раз, это говорит о том, что ее еще нет в ассоциативном массиве. В таком случае нам следует добавить эту страну и установить значение счетчика, равное `1`. Если мы *уже встречали* миллиардера из этой страны, то нужно получить ссылку на существующий счетчик, чтобы увеличить его. Именно это и происходит на шаге 6:

```
if (!success) {
    iterator->second.second += 1;
}
```

Обратите внимание: функции insert и emplace контейнера std::map работают одинаково. Основное различие между ними заключается в том, что функция try_emplace не будет создавать объект, связанный с ключом, если последний уже существует. Это повышает производительность в ситуациях, когда создание объектов занимает много времени.

Дополнительная информация

Наша программа продолжит работать, если мы сменим тип контейнера с `std::map` на `std::unordered_map`. Таким образом мы просто переключимся с одной реализа-

ции на другую, которая имеет иные характеристики производительности. В этом примере есть единственное отличие: ассоциативный массив больше не будет выводиться в алфавитном порядке, поскольку подобные массивы на основе хешей не упорядочивают свои объекты так, как это делается для деревьев поиска.

Исследуем новую семантику подсказок для вставки элементов с помощью метода std::map::insert

Поиск элементов в контейнере `std::map` занимает время *O(log(n))*. То же касается вставки новых элементов, поскольку позицию, на которую нужно добавить новый элемент, тоже необходимо найти. Простая вставка *M* новых элементов займет время *O(M * log(n))*.

Чтобы операция была более эффективной, функция вставки контейнера `std::map` принимает необязательный параметр, представляющий собой *подсказку для вставки*. Это, по сути, итератор, который указывает на место, близкое к будущей позиции вставляемого элемента. Если подсказка корректна, то амортизированное время вставки составит *O(1)*.

Как это делается

В этом примере мы вставим несколько элементов в контейнер `std::map` и используем подсказки для вставки, чтобы снизить количество операций поиска.

1. Мы преобразуем строки в числа, поэтому включим заголовочные файлы для `std::map` и `std::string`:

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

2. Далее создаем ассоциативный массив, в котором содержатся некие символы:

```
int main()
{
    std::map<std::string, size_t> m {{"b", 1}, {"c", 2}, {"d", 3}};
```

3. Теперь вставим несколько элементов и для каждого из них используем подсказку для вставки. Поскольку изначально ее у нас нет, выполним первую операцию вставки, указывая на конечный итератор ассоциативного массива:

```
    auto insert_it (std::end(m));
```

4. Вставим элементы в порядке, обратном алфавитному, используя имеющуюся у нас подсказку для вставки, а затем повторно инициализируем ее значением,

которое возвращает функция `insert`. Следующий элемент будет вставлен прямо *перед* подсказкой:

```
for (const auto &s : {"z", "y", "x", "w"}) {
    insert_it = m.insert(insert_it, {s, 1});
}
```

5. Чтобы продемонстрировать, как *не нужно* решать задачу, вставим строку, которая окажется с левого края ассоциативного массива, но зададим для нее *неправильную* подсказку, указывающую на крайнюю правую позицию ассоциативного массива — `end`:

```
m.insert(std::end(m), {"a", 1});
```

6. Наконец, просто выведем на экран полученный результат.

```
for (const auto & [key, value] : m) {
    std::cout << "\"" << key << "\": " << value << ", ";
}
std::cout << '\n';
}
```

7. Скомпилировав и запустив программу, мы получим следующий результат. Очевидно, ошибочная подсказка при вставке ничего не испортила, поскольку порядок ассоциативного массива все еще правильный:

```
"a": 1, "b": 1, "c": 2, "d": 3, "w": 1, "x": 1, "y": 1, "z": 1,
```

Как это работает

Единственное отличие от обычной вставки в этом примере заключается в том, что мы используем дополнительный итератор-подсказку. Кроме того, мы говорили о *правильных* и *неправильных* подсказках.

Правильная подсказка будет указывать на существующий элемент, чье значение *превышает* значение вставляемого элемента, чтобы новый элемент занял позицию прямо *перед* ней. Если с подсказкой, предоставленной пользователем, это невозможно сделать, то функция `insert` откатится к неоптимизированной версии, которая выполнится за время $O(log(n))$.

Во время первой вставки у нас есть итератор, указывающий на конечный элемент ассоциативного массива, поскольку у нас нет более удачной стартовой позиции. После вставки в дерево символа `z` нам станет известно, что прямо перед ним будет вставлен символ `y`, и это вполне сгодится на роль правильной подсказки. То же применимо и к символу `x`, если мы будем вставлять его в дерево после добавления символа `y`, и т. д. Именно поэтому вы можете использовать итератор, который был возвращен *последней* операцией вставки, для выполнения *следующей* вставки.

Важно знать, что до появления C++11 подсказки для вставки считались правильными, если указывали на позицию, которая стоит перед вновь вставленным элементом.

Дополнительная информация

Что интересно, неправильная подсказка не нарушает порядок элементов в ассоциативном массиве. Как же тогда это вообще работает и что же означает выражение «амортизированное время вставки равно *O(1)*»?

Контейнер `std::map` обычно реализуется с применением бинарного дерева поиска. При вставке в данное дерево новый ключ сравнивается с ключами других узлов, начиная с вершины. Если ключ меньше или больше, чем ключ текущего узла, то алгоритм поиска смещается влево или вправо и опускается вниз на один узел. Выполняя такие операции, поисковый алгоритм остановится в точке, где глубина текущего дерева максимальна, и поместит туда новый узел и его ключ. Вполне возможно, что данный шаг нарушит баланс дерева, поэтому после вставки будет выполнен алгоритм перебалансировки.

Когда мы вставляем в дерево элементы, которые имеют ключи, являющиеся непосредственными соседями друг друга (например, целое число `1` выступает соседом целого числа `2`, поскольку нельзя поместить между ними ни одно целое число), они *часто* могут оказаться в дереве рядом друг с другом. Это нетрудно проверить, если это верно для определенного ключа и соответствующей подсказки. В таком случае в алгоритме вставки будет пропущен этап поиска, что позволит сэкономить существенное количество времени. Тем не менее после этого все равно может быть запущен алгоритм перебалансировки. Поскольку такую оптимизацию можно выполнять *часто* (хоть и *не всегда*), это приводит к *повышению* средней производительности. После выполнения нескольких вставок *итоговая* сложность алгоритма снижается, вот и получается *амортизированная сложность* (рис. 2.3).

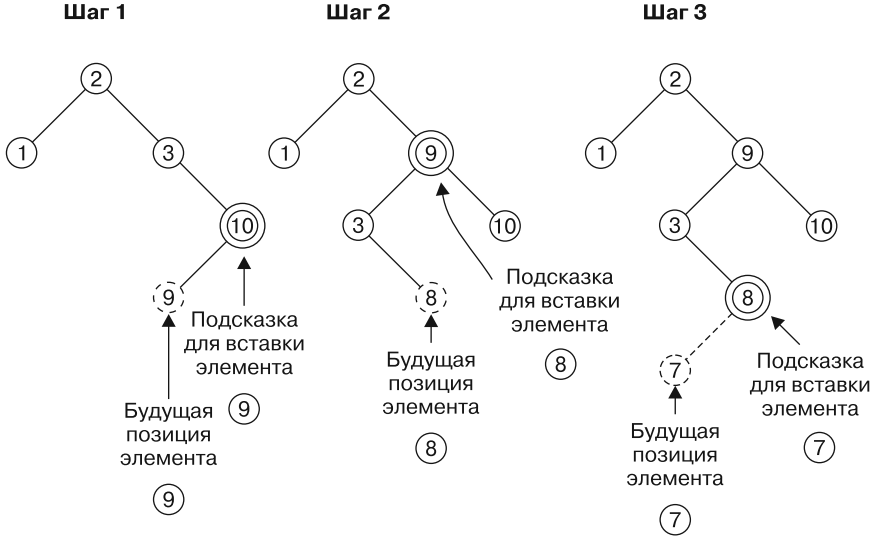

Рис. 2.3

Если же подсказка ошибочна, то функция вставки попросту *проигнорирует* ее и начнет работу с выполнения алгоритма поиска. В итоге она отработает корректно, но, очевидно, *медленнее*.

Эффективно изменяем ключи элементов std::map

Поскольку структура данных `std::map` соотносит ключи со значениями таким образом, что ключи всегда уникальны и отсортированы, очень важно исключить для пользователей возможность изменить ключи узлов, которые уже вставлены в контейнер. Чтобы пользователи не могли изменять ключи элементов идеально отсортированных узлов ассоциативного массива, к типу ключа добавляется слово `const`.

Такого рода ограничение разумно, поскольку пользователю будет сложнее неправильно задействовать контейнер `std::map`. Но что же делать, если нам вдруг действительно понадобится изменить ключи некоторых элементов ассоциативного массива?

До появления C++17 нам приходилось удалять элементы, для которых нужно изменить ключ, а затем вставлять их снова. Недостаток такого подхода состоит в выполнении бесполезных выделений и высвобождений памяти, что плохо с точки зрения производительности.

Начиная с C++17 можно удалить и снова вставить элементы ассоциативного массива без повторного выделения памяти. Далее мы увидим, как это работает.

Как это делается

В этом примере мы реализуем небольшое приложение, которое упорядочивает список водителей, участвующих в вымышленной гонке, в структуре `std::map`. По мере того как водители обходят друг друга во время гонки, нужно изменять ключи, показывающие их текущее место. Мы будем делать это в стиле C++17.

1. Начнем с того, что включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <map>

using namespace std;
```

2. Мы выведем на экран места всех водителей до и после изменения контейнера `map`, поэтому реализуем вспомогательную функцию, посвященную именно этому:

```
template <typename M>
void print(const M &m)
{
```

```
    cout << "Race placement:\n";
    for (const auto &[placement, driver] : m) {
        cout << placement << ": " << driver << '\n';
    }
}
```

3. В функции `main` создадим и инициализируем ассоциативный массив, в котором целые числа, указывающие текущее место гонщика, будут соотноситься со строками, содержащими его имя. Кроме того, выведем содержимое ассоциативного массива на экран, поскольку на следующих шагах изменим его.

```
int main()
{
    map<int, string> race_placement {
        {1, "Mario"}, {2, "Luigi"}, {3, "Bowser"},
        {4, "Peach"}, {5, "Yoshi"}, {6, "Koopa"},
        {7, "Toad"}, {8, "Donkey Kong Jr."}
    };
    print(race_placement);
```

4. Предположим, что на одном из кругов гонки у Боузера (Bowser) произошла небольшая авария и он откатился на последнее место, а Донки Конгу — младшему (Donkey Kong Jr.) представился шанс перескочить с последнего места на третье. В данном случае сначала нужно извлечь указанные элементы из ассоциативного массива, поскольку это единственный способ манипулировать их ключами. Функция `extract` — новая возможность C++17. Она удаляет элементы из массива, притом не вызывая побочных эффектов, связанных с выделением памяти. Создадим также новую область видимости для данной задачи.

```
    {
        auto a (race_placement.extract(3));
        auto b (race_placement.extract(8));
```

5. Теперь поменяем местами ключи Боузера и Донки Конга — младшего. Несмотря на то что ключи элементов ассоциативного массива обычно неизменяемы (поскольку объявлены с модификатором `const`), можно изменить ключи извлеченных элементов, полученных с помощью метода `extract`.

```
        swap(a.key(), b.key());
```

6. Метод `insert` контейнера `std::map` в C++17 получил новую перегруженную версию, которая принимает дескрипторы извлеченных узлов, чтобы вставить их в контейнер, не вызывая выделения памяти:

```
        race_placement.insert(move(a));
        race_placement.insert(move(b));
    }
```

7. Выходим из области видимости, и на этом все. Выводим на экран новые места гонщиков и позволяем приложению завершиться:

```
    print(race_placement);
}
```

8. Компиляция и запуск программы дадут следующий результат. Сначала мы видим изначальную расстановку гонщиков, а затем новую, которую получили после изменения позиций Боузера и Донки Конга — младшего:

```
$ ./mapnode_key_modification
Race placement:
1: Mario
2: Luigi
3: Bowser
4: Peach
5: Yoshi
6: Koopa
7: Toad
8: Donkey Kong Jr.
Race placement:
1: Mario
2: Luigi
3: Donkey Kong Jr.
4: Peach
5: Yoshi
6: Koopa
7: Toad
8: Bowser
```

Как это работает

В C++17 контейнер `std::map` получил новую функцию-член `extract`. Она поставляется в двух версиях:

```
node_type extract(const_iterator position);
node_type extract(const key_type& x);
```

В этом примере мы используем вторую версию, которая принимает ключ, а затем находит и извлекает соответствующий ему элемент ассоциативного массива. Первая версия функции принимает итератор, что подразумевает ее *более быструю* работу, поскольку ей не нужно искать элемент.

Если мы попробуем извлечь с помощью второго метода (выполняющего поиск по ключу) элемент, которого не существует, то получим пустой экземпляр типа `node_type`. Метод-член `empty()` возвращает булево значение, которое указывает, пуст ли экземпляр `node_type`. Вызов любого метода для пустого экземпляра приводит к неопределенному поведению.

После извлечения узлов можно изменить их ключи с помощью метода `key()`, который предоставляет неконстантный доступ к ключам, несмотря на то что они обычно являются неизменяемыми.

Обратите внимание: для повторного добавления узлов в ассоциативный массив нужно *передать* их в функцию `insert`. Это логично, поскольку функция `extract` стремится избежать создания ненужных копий и выделения памяти. Еще одно примечание: хотя мы перемещаем экземпляр типа `node_type`, на самом деле никакие значения из контейнера не перемещаются.

Дополнительная информация

Элементы ассоциативного массива, которые были извлечены с помощью метода `extract`, обычно довольно гибкие. Можно извлечь элементы из одного экземпляра типа `map` и вставить их в любой другой экземпляр типа `map` или даже `multimap`. Это верно и для контейнеров `unordered_map` и `unordered_multimap`, а также `set`/`multiset` и, соответственно, `unordered_set`/`unordered_multiset`.

Для того чтобы можно было перемещать элементы из одной структуры ассоциативного массива или множества в другую, типы ключей, значений и средство выделения памяти должны быть идентичны. Обратите внимание: даже при совпадении этих типов мы не можем перемещать элементы из `map` в `unordered_map` или из `set` в `unordered_set`.

Применяем контейнер std::unordered_map для пользовательских типов

Использование контейнера `std::unordered_map` вместо `std::map` подразумевает дополнительную степень свободы при выборе типа ключа. Контейнер `std::map` требует наличия между ключами естественного порядка. Таким образом элементы можно отсортировать. Но если мы хотим, например, использовать в качестве ключа математический вектор? Мы не можем сказать, что вектор (0, 1) *меньше* или *больше* вектора (1, 0). Они просто указывают в разных направлениях. Это верно для контейнера `std::unordered_map`, поскольку мы станем различать элементы не по их величине, а по *значениям их хеша*. Единственное, что нужно сделать, — реализовать *хеш-функцию* для нашего собственного типа, а также оператор `==`, который будет проверять идентичность двух объектов. В данном разделе мы рассмотрим пример такой реализации.

Как это делается

В примере мы определим простую структуру `coord`, которая по умолчанию не имеет хеш-функции, поэтому нам потребуется установить ее самостоятельно. Затем задействуем ее, задав соотношение значений `coord` с числами.

1. Сначала включим все необходимое, чтобы вывести на экран и использовать контейнер `std::unordered_map`:

```
#include <iostream>
#include <unordered_map>
```

2. Далее определим собственную структуру, которую не так-то просто хешировать с помощью существующих хеш-функций:

```
struct coord {
    int x;
    int y;
};
```

3. Нам нужна не только хеш-функция, которая позволит использовать структуру в качестве ключа для ассоциативного массива, основанного на хеше, но и реализация оператора сравнения:

```
bool operator==(const coord &l, const coord &r)
{
    return l.x == r.x && l.y == r.y;
}
```

4. Чтобы расширить возможности хеширования, предоставляемые STL, добавим в пространство имен `std` свою специализацию шаблонной структуры `std::hash`. Оно содержит такие же псевдонимы (задаваемые с помощью `using`), как и другие специализации типа `hash`.

```
namespace std
{
template <>
struct hash<coord>
{
    using argument_type = coord;
    using result_type   = size_t;
```

5. Основная часть данной структуры — определение функции `operator()`. Мы просто складываем значения численных членов структуры `coord`. Эта техника хеширования не самая лучшая, но для демонстрации подойдет. Хорошая хеш-функция пытается максимально равномерно распределить значения в рамках допустимого диапазона, чтобы сократить количество *хеш-столкновений*.

```
    result_type operator()(const argument_type &c) const
    {
        return static_cast<result_type>(c.x)
             + static_cast<result_type>(c.y);
    }
};
}
```

6. Теперь можно создать новый экземпляр контейнера `std::unordered_map`, который принимает в качестве ключа структуру `coord` и соотносит ее с некоторыми значениями. Поскольку этот раздел посвящен использованию собственных типов для контейнера `std::unordered_map`, наш пример почти завершен. Создадим ассоциативный массив, основанный на хеше, с нашим собственным типом, заполним его какими-нибудь элементами и выведем содержимое на экран:

```
int main()
{
    std::unordered_map<coord, int> m {{{0, 0}, 1}, {{0, 1}, 2},
                                      {{2, 1}, 3}};
    for (const auto & [key, value] : m) {
        std::cout << "{(" << key.x << ", " << key.y
                  << "): " << value << "} ";
    }
    std::cout << '\n';
}
```

7. Компиляция и запуск программы дадут следующий результат:

```
$ ./custom_type_unordered_map
{(2, 1): 3} {(0, 1): 2} {(0, 0): 1}
```

Как это работает

Обычно при создании экземпляра ассоциативного массива, основанного на хеше, например `std::unordered_map`, мы пишем следующую команду:

```
std::unordered_map<key_type, value_type> my_unordered_map;
```

Не вполне очевиден тот факт, что при создании компилятором контейнера уточненного типа `std::unordered_map` за кулисами творится настоящее волшебство. Поэтому взглянем на полное определение шаблонного типа:

```
template<
    class Key,
    class T,
    class Hash      = std::hash<Key>,
    class KeyEqual  = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

В качестве первых двух типов шаблона мы выбрали `coord` и `int`, здесь все просто и очевидно. Три остальных типа шаблона являются необязательными, так что автоматически заполняются существующими стандартными шаблонными классами, которые сами принимают типы шаблонов. В случае если последние аргументы заполняются значениями по умолчанию, в качестве типов шаблонов передаются те типы, которые мы указали в первых двух аргументах.

Для нас сейчас особый интерес представляет шаблонный параметр `class Hash`: когда мы не указываем явно что-то другое, он будет специализирован как `std::hash<key_type>`. STL уже содержит специализации `std::hash<std::string>`, `std::hash<int>`, `std::hash<unique_ptr>` и многие другие. Эти классы знают, как работать с подобными конкретными типами, что позволяет вычислять оптимальные хеш-значения.

Однако STL пока не знает, как рассчитывать хеш-значение на основе нашей структуры `coord`. Мы лишь определили еще одну специализацию, которая знает, как это делается. Компилятор теперь может пройти по списку всех известных специализаций для контейнера `std::hash` и найти нашу реализацию, что позволит соотнести ее с типом, который мы выбрали для ключа.

Если бы мы не добавили новую специализацию `std::hash<coord>`, а назвали бы имеющуюся вместо этого `my_hash_type`, то нам пришлось бы использовать следующую строку для создания объекта:

```
std::unordered_map<coord, value_type, my_hash_type> my_unordered_map;
```

Очевидно, что таким образом придется набирать больше кода. Кроме того, при этом подходе код тяжелее читать, в отличие от ситуации, когда компилятор самостоятельно находит правильную реализацию хеш-функции.

Отсеиваем повторяющиеся слова из пользовательского ввода и выводим их на экран в алфавитном порядке с помощью контейнера std::set

Контейнер `std::set` довольно странный. Принцип его работы похож на принцип работы контейнера `std::map`. Однако `std::set` содержит только ключи в качестве значений, а не пары «ключ — значение». Поэтому он не очень подходит для соотношения значения одного типа со значениями другого. Поскольку способы применения этого контейнера не вполне очевидны, многие разработчики даже не знают о его существовании. Они часто начинают реализовывать похожие механизмы самостоятельно, хотя в некоторых ситуациях контейнер `std::set` оказался бы отличным подспорьем.

В этом разделе будет показано, как применить контейнер `std::set`, на примере, в котором мы получаем потенциально большое количество разных элементов. Мы *отфильтруем* их и выведем на экран *уникальные* элементы.

Как это делается

В этом примере мы считаем поток слов из стандартного средства ввода. Все *уникальные* слова будут помещены в экземпляр класса `std::set`. Таким образом мы сможем перечислить все уникальные слова из потока[1].

1. Мы используем несколько разных типов STL, для чего включим некоторые заголовочные файлы:

```
#include <iostream>
#include <set>
#include <string>
#include <iterator>
```

2. Чтобы сэкономить немного времени на наборе текста, объявим об использовании пространства имен `std`:

```
using namespace std;
```

3. Теперь мы готовы писать саму программу, начинающуюся с функции `main`. В ней создается экземпляр класса `std::set`, в котором будут храниться строки:

```
int main()
{
    set<string> s;
```

4. Далее получим данные от пользователя. Просто считаем их из стандартного потока ввода с помощью удобного итератора `istream_iterator`:

```
    istream_iterator<string> it {cin};
    istream_iterator<string> end;
```

[1] Все это будет проделано средствами стандарта C++98, из нового в этом разделе только синтаксис.

5. Имея начальный и конечный итераторы, которые представляют данные, введенные пользователем, можем просто заполнить множество на основе этих данных с помощью `std::inserter`:

```
copy(it, end, inserter(s, s.end()));
```

6. На этом, в общем-то, все. Чтобы увидеть, какие *уникальные* слова мы получили из стандартного ввода, просто выведем на экран содержимое нашего множества:

```
    for (const auto word : s) {
        cout << word << ", ";
    }
    cout << '\n';
}
```

7. Скомпилируем и запустим программу с нашими входными данными. Взглянем на полученный результат, из которого были удалены все дубликаты, а уникальные слова теперь отсортированы по алфавиту:

```
$ echo "a a a b c foo bar foobar foo bar bar" | ./program
a, b, bar, c, foo, foobar,
```

Как это работает

В данной программе можно отметить два интересных момента. Первый из них состоит в том, что мы применяем итератор `std::istream_iterator`, чтобы получить доступ к данным, введенным пользователем. Второй момент: для записи этих данных в наш контейнер `std::set` мы задействуем алгоритм `std::copy`, который обернули в экземпляр класса `std::inserter`! Может показаться удивительным то, что всего одна строка кода выполняет всю работу по *токенизации* входных данных, *помещению* их во множество, *отсортированное* по алфавиту, и *отсечению* дубликатов.

std::istream_iterator

Этот класс очень интересен в тех случаях, когда мы хотим обрабатывать большие объемы *однотипных* данных, получаемые из потока, чему и посвящен наш пример: мы анализируем все входные данные слово за словом и помещаем их в множество в виде экземпляров класса `std::string`.

Итератор `std::istream_iterator` принимает один шаблонный параметр с необходимым нам типом. Мы выбрали тип `std::string`, поскольку ожидаем, что будем работать со словами, но это могут быть, например, и числа с плавающей точкой. По сути, здесь годится любой тип, для которого можно записать `cin >> var;`. Конструктор принимает экземпляр класса `istream`. Стандартный поток ввода представляется глобальным объектом потока ввода `std::cin`, который вполне подходит для нашего случая.

```
istream_iterator<string> it {cin};
```

К созданному итератору потока ввода применимы две операции. Во-первых, при разыменовании (`*it`) он возвращает текущий введенный символ. Поскольку мы

указали, что итератор соотнесен с типом `std::string` с помощью шаблонного параметра, этот символ будет содержать одно слово. Во-вторых, при инкременте (`++it`) итератор переходит на следующее слово, к которому мы также можем получить доступ путем разыменования.

Но погодите, нужно быть аккуратными после выполнения операции инкремента и до разыменования. При *пустом* стандартном потоке ввода итератор нельзя разыменовывать. Вместо этого следует завершить цикл, в котором мы разыменовываем итератор, для получения каждого слова. Условие остановки, позволяющее узнать, что итератор стал некорректным, — сравнение с конечным итератором. Если сравнение `it == end` выполнилось, то мы дошли до последнего введенного слова.

Конечный итератор — это экземпляр `std::istream_iterator`, созданный с помощью стандартного конструктора без параметров. Он нужен для сравнения в условии остановки, проверяемом на каждой итерации цикла:

```
istream_iterator<string> end;
```

Как только `std::cin` окажется пустым, итератор `it` *обнаружит* это и наше сравнение с конечным оператором вернет результат `true`.

std::inserter

Мы использовали пару итераторов `it` и `end` как *итераторы для работы с входными данными* в вызове `std::copy`. Третий параметр должен быть итератором для работы с выходными данными. Здесь мы не можем просто взять итератор `s.begin()` или `s.end()`. Для пустого множества они будут одинаковыми, так что нам даже нельзя *разыменовать* их независимо от того, делаем мы это для чтения (из) или присваивания (в).

Тут вступает в дело `std::inserter`. Это функция, возвращающая итератор `std::insert_iterator`, который ведет себя как итератор, но при этом делает что-то отличное от того, что делают обычные итераторы. Выполнение операции инкремента для данного итератора ничего не даст. Когда мы его разыменовываем и присваиваем значение, он берет контейнер, прикрепленный к нему, и *добавляет* в него заданное значение как *новый* элемент!

При создании экземпляра `std::insert_iterator` с помощью `std::inserter` вам понадобятся два параметра:

```
auto insert_it = inserter(s, s.end());
```

Здесь `s` — наше множество, а `s.end()` — итератор, указывающий на место, куда должен быть вставлен новый элемент. Для пустого множества, с которого и начинается наш пример, этот итератор эквивалентен `s.begin()`. В других структурах данных наподобие векторов или списков второй параметр критически важен при определении того, куда именно итератор вставки должен добавить новые элементы.

Собираем все воедино

В конце концов все волшебство происходит во время вызова метода `std::copy`:

```
copy(input_iterator_begin, input_iterator_end, insert_iterator);
```

Данный вызов получает следующий токен слова из потока `std::cin` с помощью входного итератора и помещает его в контейнер `std::set`. После этого он инкрементирует оба итератора и проверяет, равен ли входной итератор конечному. Если это не так, то в потоке ввода еще остаются слова, так что все *повторяется*.

Повторяющиеся слова отбрасываются автоматически. При наличии в множестве конкретного слова повторное его добавление *эффекта не возымеет*. Этим контейнер `std::set` отличается от `std::multiset`, куда можно вставить повторяющиеся записи.

Реализуем простой ОПН-калькулятор с использованием контейнера std::stack

Класс `std::stack` — класс-адаптер, который позволяет помещать *в себя* объекты, словно в реальную стопку объектов, а также получать их. В текущем разделе на основе этой структуры данных мы создадим калькулятор, применяющий обратную польскую нотацию, ОПН (reverse polish notation, RPN).

ОПН — это нотация, которая может служить для записи математических выражений таким образом, чтобы их было проще анализировать. В ОПН выражение `1 + 2` выглядит как `1 2 +`. Сначала идут операнды, а затем — оператор. Еще один пример: выражение `(1 + 2) * 3` в ОПН выглядит как `1 2 + 3 *`, оно уже показывает, почему подобные выражения проще анализировать, и нам не нужны скобки, чтобы определить подвыражения (рис. 2.4).

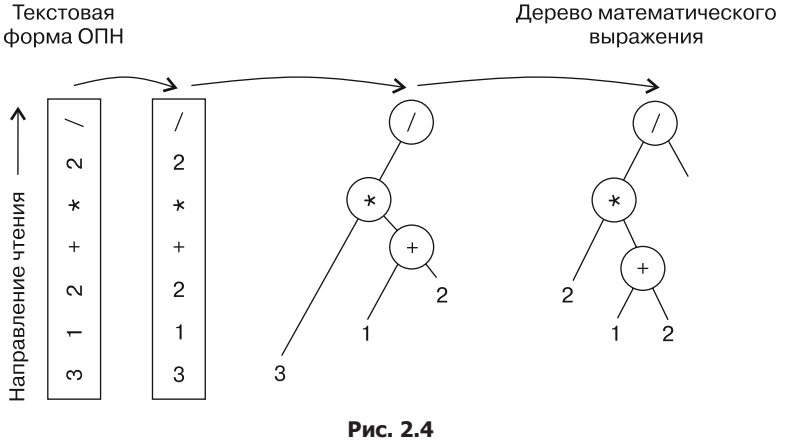

Рис. 2.4

Как это делается

В этом примере мы считаем математическое выражение, записанное в формате ОПН, из стандартного потока ввода, а затем передадим его в функцию, которая вычислит его значение. Наконец, выведем на экран численный результат.

1. Мы будем использовать множество вспомогательных объектов из STL, поэтому сначала разместим несколько директив включения:

```
#include <iostream>
#include <stack>
#include <iterator>
#include <map>
#include <sstream>
#include <cassert>
#include <vector>
#include <stdexcept>
#include <cmath>
```

2. Мы также объявляем, что используем пространство имен `std`, чтобы сэкономить немного времени, которое ушло бы на набор текста:

```
using namespace std;
```

3. Далее немедленно начнем реализовывать наш анализатор ОПН. Он станет принимать пару итераторов, указывающих на начало и конец математического выражения, передаваемого в качестве строки, которое будет проработано токен за токеном:

```
template <typename IT>
double evaluate_rpn(IT it, IT end)
{
```

4. Пока мы итерируем по токенам, нам следует запоминать все *операнды* до тех пор, пока мы не встретим *операцию*. Для этого и нужен стек. Все числа будут проанализированы и сохранены с удвоенной точностью, поэтому мы заводим стек элементов типа `double`:

```
    stack<double> val_stack;
```

5. Чтобы удобным образом получить доступ к элементам стека, реализуем вспомогательную функцию. Она изменяет стек, извлекая значение с его вершины, а затем возвращает это значение. Таким образом мы сможем выполнить нашу задачу за один шаг:

```
    auto pop_stack ([&](){
        auto r (val_stack.top());
        val_stack.pop();
        return r;
    });
```

6. Еще одним приготовлением будет определение всех поддерживаемых математических операций. Мы сохраним их в ассоциативный массив, где каждый токен операции будет связан с самой операцией. Операции представлены вызываемыми лямбда-выражениями, которые принимают два операнда, а затем, например, складывают или умножают их и возвращают результат:

```
map<string, double (*)(double, double)> ops {
    {"+", [](double a, double b) { return a + b; }},
    {"-", [](double a, double b) { return a - b; }},
    {"*", [](double a, double b) { return a * b; }},
    {"/", [](double a, double b) { return a / b; }},
    {"^", [](double a, double b) { return pow(a, b); }},
    {"%", [](double a, double b) { return fmod(a, b); }},
};
```

7. Теперь наконец можно проитерировать по входным данным. Предположив, что входные итераторы передали строки, мы передаем данные в новый поток `std::stringstream` токен за токеном, поскольку он может анализировать числа:

```
for (; it != end; ++it) {
    stringstream ss {*it};
```

8. Теперь, когда у нас есть все токены, попробуем получить на их основе значение типа `double`. Если эта операция завершается успешно, то у нас появляется *операнд*, который мы помещаем в стек:

```
    if (double val; ss >> val) {
        val_stack.push(val);
    }
```

9. Если же операция завершается *неудачно*, то перед нами нечто отличное от оператора. Это может быть только *операнд*. Зная, что все поддерживаемые нами операции *бинарны*, нужно вытолкнуть *два* последних операнда из стека:

```
    else {
        const auto r {pop_stack()};
        const auto l {pop_stack()};
```

10. Теперь мы получаем операнд путем разыменования итератора `it`, который возвращает строки. Обратившись в ассоциативный массив `ops`, мы получаем лямбда-объект, принимающий в качестве параметров два операнда — `l` и `r`:

```
        try {
            const auto & op     (ops.at(*it));
            const double result {op(l, r)};
            val_stack.push(result);
        }
```

11. Мы окружили математическую часть приложения блоком `try`, поэтому можем отловить потенциально возникающие исключения. Вызов функции `at` для контейнера `map` сгенерирует исключение `out_of_range`, если пользователь даст команду выполнить математическую операцию, о которой мы не знаем. В таком

случае мы повторно сгенерируем другое исключение, сообщающее о том, что полученный аргумент некорректен (`invalid argument`), и содержащее строку, которая оказалась неизвестной для нас:

```
            catch (const out_of_range &) {
                throw invalid_argument(*it);
            }
```

12. На этом все. Когда цикл прекратит свою работу, у нас в стеке будет итоговый результат. Так что мы просто вернем его. (В тот момент можно проверить, равен ли размер стека единице. Если нет, значит, некоторые операции были пропущены.)

```
        }
    }
    return val_stack.top();
}
```

13. Теперь можно воспользоваться нашим анализатором ОПН. Для этого обернем стандартный поток ввода данных с помощью пары итераторов `std::istream_iterator` и передадим его функции-анализатору ОПН. Наконец, выведем результат на экран:

```
int main()
{
    try {
        cout << evaluate_rpn(istream_iterator<string>{cin}, {})
             << '\n';
    }
```

14. Опять же эту строку мы обернули в блок `try`, поскольку существует вероятность, что пользовательские данные содержат операции, которые у нас не реализованы. В таких ситуациях нужно перехватывать генерируемое исключение и выводить на экран сообщение об ошибке:

```
    catch (const invalid_argument &e) {
        cout << "Invalid operator: " << e.what() << '\n';
    }
}
```

15. После компиляции программы можно с ней поэкспериментировать. Входные данные `"3 1 2 + * 2 /"` представляют собой выражение `( 3 * (1 + 2) ) / 2`, которое приложение преобразует к корректному результату:

```
$ echo "3 1 2 + * 2 /" | ./rpn_calculator
4.5
```

Как это работает

Весь пример строится на помещении операндов в стек до тех пор, пока мы не найдем во входных данных операцию. В этой ситуации мы выталкиваем два последних операнда из стека, применяем к ним операцию и помещаем результат обратно

в стек. Чтобы понять весь код примера, важно разобраться, как мы различаем *операнды* и *операторы*, работаем со стеком, а также выбираем и применяем правильную математическую операцию.

Работа со стеком

Мы помещаем элементы в стек с помощью функции `push` класса `std::stack`:

```
val_stack.push(val);
```

Выталкивание элемента из стека выглядит чуть сложнее, поскольку нам пришлось реализовывать для этого лямбда-выражение, которое принимает ссылку на объект `val_stack`. Взглянем на код, только теперь добавим к нему комментарии:

```
auto pop_stack ([&](){
    auto r (val_stack.top()); // Получаем копию верхнего значения
    val_stack.pop();          // Удаляем верхнее значение
    return r;                 // Возвращаем копию
});
```

Это лямбда-выражение необходимо для получения верхнего значения стека и его *удаления* из самого адаптера всего за *один* шаг. Интерфейс класса `std::stack` не позволяет делать это с помощью *одного* простого вызова. Однако определить лямбда-выражение нетрудно, так что теперь можно получать значения следующим образом:

```
double top_value {pop_stack()};
```

Различаем в пользовательском вводе операнды и операторы

В основном цикле функции `evaluate_rpn` мы получаем текущий токен строки из итератора и затем смотрим, является ли он операндом. Если строка может быть преобразована в переменную типа `double`, то данное число тоже операнд. Все остальные токены, которые нельзя легко преобразовать в число (например, `"+"`), мы считаем *операторами*.

Скелет кода для выполнения именно этой задачи выглядит следующим образом:

```
stringstream ss {*it};
if (double val; ss >> val) {
    // Это число!
} else {
    // Это что-то другое. Это операция!
}
```

Оператор потока `>>` говорит нам, является ли рассматриваемый объект числом. Сначала мы оборачиваем строку в `std::stringstream`. Затем используем способность объекта класса `stringstream` преобразовать объект типа `std::string` в переменную типа `double`, что включает в себя разбор. Если он *не работает*, то мы узнаем об этом, поскольку не получится преобразовать в число некий объект, который числом не является.

Выбираем и применяем нужную математическую операцию

Разобравшись, что текущий токен, полученный от пользователя, не является числом, мы предполагаем, что это операция наподобие + или *. Затем обращаемся к ассоциативному массиву `ops` с целью найти требуемую операцию и получить функцию, принимающую два операнда и возвращающую сумму, произведение или другой подходящий результат.

Сам тип такого массива выглядит относительно сложно:

```
map<string, double (*)(double, double)> ops { ... };
```

Он соотносит строки и значения типа `double (*)(double, double)`. Что означает вторая часть данного выражения? Это описание типа читается как «указатель на функцию, которая принимает два числа типа `double` и возвращает одно». Представьте, будто часть `(*)` представляет собой имя функции, как, например, `double sum(double, double)`, что гораздо проще прочитать. Идея заключается в следующем: наше лямбда-выражение `[](double, double) { return /* какое-то число типа double */ }` можно преобразовать в указатель на функцию, фактически соответствующий описанию этого указателя. Лямбда-выражения, которые *не захватывают переменных из внешнего контекста*, могут быть преобразованы в указатели на функции.

Таким образом, это удобный способ запросить у ассоциативного массива корректную операцию:

```
const auto & op     (ops.at(*it));
const double result {op(l, r)};
```

Ассоциативный массив неявно решает еще одну задачу. Если мы выполняем вызов `ops.at("foo")`, то в данном случае `"foo"` является корректным значением ключа, но мы не сохранили операцию с таким именем. В подобных случаях массив сгенерирует исключение, которое мы отлавливаем в нашем примере. При его перехвате мы генерируем другое исключение, чтобы представить более подробное сообщение об ошибке. Пользователь будет лучше понимать, что означает полученное исключение, сообщающее о некорректном аргументе (`invalid argument`), в отличие от исключения, гласящего о выходе за пределы контейнера. Обратите внимание: пользователь функции `evaluate_rpn` может быть незнаком с ее реализацией и поэтому не знает о том, что мы применяем ассоциативный массив.

Дополнительная информация

Поскольку функция `evaluate_rpn` принимает итераторы, ей можно легко передавать разные входные данные, не только стандартный поток ввода. Это позволяет довольно просто протестировать ее, а также адаптировать к различным источникам данных, получаемых от пользователя.

Передача в эту функцию итераторов строкового потока или вектора строк, например, выглядит следующим образом. При этом код функции `evaluate_rpn` остается без изменений:

```
int main()
{
    stringstream s {"3 2 1 + * 2 /"};
    cout << evaluate_rpn(istream_iterator<string>{s}, {}) << '\n';
    vector<string> v {"3", "2", "1", "+", "*", "2", "/"};
    cout << evaluate_rpn(begin(v), end(v)) << '\n';
}
```

Используйте итераторы везде, где это имеет смысл. Так вы сможете многократно применять свой код.

Подсчитываем частоту встречаемости слов с применением контейнера std::map

Контейнер `std::map` очень полезен в тех случаях, когда нужно разбить данные на категории и собрать соответствующую статистику. Прикрепляя изменяемые объекты к каждому ключу, который представляет собой категорию объектов, вы легко можете реализовать, к примеру, гистограмму, показывающую частоту встречаемости слов. Этим мы и займемся.

Как это делается

В этом примере мы считаем все данные, которые пользователь передает через стандартный поток ввода и которые, скажем, могут оказаться текстовым файлом. Мы разобьем полученный текст на слова, чтобы определить частоту встречаемости каждого слова.

1. Как обычно, включим все заголовочные файлы для тех структур данных, которые планируем использовать:

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

2. Чтобы сэкономить немного времени на наборе, объявляем об использовании пространства имен `std`:

   ```
   using namespace std;
   ```

3. Задействуем одну вспомогательную функцию, с помощью которой будем обрезать прикрепившиеся знаки препинания (например, запятые, точки и двоеточия):

   ```
   string filter_punctuation(const string &s)
   {
       const char *forbidden {".,:; "};
   ```

```
    const auto  idx_start (s.find_first_not_of(forbidden));
    const auto  idx_end   (s.find_last_not_of(forbidden));
    return s.substr(idx_start, idx_end - idx_start + 1);
}
```

4. Теперь начнем писать саму программу. Создадим ассоциативный массив, в котором будут связаны каждое встреченное нами слово и счетчик, показывающий, насколько часто это слово встречается. Дополнительно введем переменную, которая будет содержать величину самого длинного встреченного нами слова, чтобы в конце работы программы перед выводом на экран мы могли красиво выровнять полученную таблицу:

```
int main()
{
    map<string, size_t> words;
    int max_word_len {0};
```

5. Когда мы выполняем преобразование из `std::cin` в переменную типа `std::string`, поток ввода обрезает лишние пробельные символы. Таким образом мы получаем входные данные слово за словом:

```
    string s;
    while (cin >> s) {
```

6. Текущее слово может содержать запятые, точки или двоеточие, поскольку может находиться в середине или в конце предложения. Избавимся от этих знаков с помощью вспомогательной функции, которую определили ранее:

```
        auto filtered (filter_punctuation(s));
```

7. В том случае, если текущее слово оказалось самым длинным из всех встреченных нами, обновляем переменную `max_word_len`:

```
        max_word_len = max<int>(max_word_len, filtered.length());
```

8. Теперь увеличим значение счетчика в нашем ассоциативном массиве `words`. Если слово встречается в первый раз, то оно неявно добавляется в массив перед выполнением операции инкремента:

```
        ++words[filtered];
    }
```

9. После завершения цикла мы знаем, что сохранили все уникальные слова из потока ввода в ассоциативный массив `words` вместе со счетчиками, указывающими на частоту встречаемости каждого слова. Ассоциативный массив использует слова в качестве ключей, они отсортированы в *алфавитном* порядке. Нужно вывести все слова, отсортировав их по частоте *встречаемости*, чтобы наиболее частые слова были первыми. Для данной цели создадим вектор нужного размера, куда поместим все эти пары:

```
    vector<pair<string, size_t>> word_counts;
    word_counts.reserve(words.size());
    move(begin(words), end(words), back_inserter(word_counts));
```

10. Теперь вектор содержит все пары «слово — частота» в том же порядке, в каком они находились в ассоциативном массиве `words`. Далее отсортируем его снова, чтобы наиболее частые слова оказались в начале, а самые редкие — в конце:

```
    sort(begin(word_counts), end(word_counts),
        [](const auto &a, const auto &b) {
            return a.second > b.second;
        });
```

11. Все данные выстроены в нужном порядке, поэтому отправим их на консоль. Используя манипулятор потока `std::setw`, красиво отформатируем данные с помощью отступов так, чтобы они были похожи на таблицу:

```
    cout << "# " << setw(max_word_len) << "<WORD>" << " #<COUNT>\n";
    for (const auto & [word, count] : word_counts) {
        cout << setw(max_word_len + 2) << word << " #"
             << count << '\n';
    }
}
```

12. После компиляции программы можно обработать любой текстовый файл и получить для него таблицу частоты встречаемости слов:

```
$ cat lorem_ipsum.txt | ./word_frequency_counter
#       <WORD> #<COUNT>
            et #574
         dolor #302
           sed #273
          diam #273
           sit #259
         ipsum #259
...
```

Как это работает

Этот пример посвящен сбору всех слов в контейнере `std::map` и последующему их перемещению в контейнер `std::vector`, где они будут отсортированы для вывода на экран. Почему?

Взглянем на пример. Если мы подсчитаем частоту встречаемости слов в строке `"a a b c b b b d c c"`, то получим следующее содержимое массива:

```
a -> 2
b -> 4
c -> 3
d -> 1
```

Однако мы хотели бы представить данные пользователю в другом порядке. Программа сначала должна вывести на экран `b`, поскольку это слово встречается чаще остальных. Затем `c`, `a` и `d`. К сожалению, мы не можем запросить у ассоциативного массива *ключ с максимальным значением*, а потом *ключ со вторым по величине значением* и т. д.

Здесь в игру вступает вектор. Мы указали, что в него будут входить пары, состоящие из строки и значения счетчика. Таким образом, он станет принимать именно те значения, которые хранятся в массиве:

```
vector<pair<string, size_t>> word_counts;
```

Далее мы заполняем вектор парами «слово — частота» с помощью алгоритма `std::move`. Он выгодно отличается от других: та часть строки, которая находится в куче, не будет продублирована, а только перемещена из ассоциативного массива в вектор. Это позволит избежать создания множества копий.

```
move(begin(words), end(words), back_inserter(word_counts));
```

В некоторых реализациях STL используется оптимизация коротких строк: если строка не слишком длинная, то в куче для нее не будет выделена память, вместо этого ее сохранят непосредственно в объекте строки. В таком случае скорость перемещения не увеличивается. Но она и не уменьшается!

Следующий интересный шаг — операция сортировки, в которой в качестве пользовательского оператора сравнения применяется лямбда-выражение:

```
sort(begin(word_counts), end(word_counts),
       [](const auto &a, const auto &b) { return a.second > b.second; });
```

Алгоритм сортировки будет принимать элементы попарно и сравнивать их, этим он ничем не отличается от других алгоритмов сортировки. Предоставляя такую лямбда-функцию, мы даем алгоритму команду не просто определить, меньше ли значение `a`, чем значение `b` (реализация по умолчанию), но и сравнить значения `a.second` и `b.second`. Обратите внимание: все объекты являются *парами* «строка и ее значение счетчика», и с помощью нотации `a.second` мы получаем доступ к значению счетчика для слова. Таким образом, наиболее часто встречающиеся слова перемещаются в начало вектора, а наиболее редко встречающиеся — в конец.

Вспомогательный стилистический редактор для поиска длинных предложений в текстах с помощью std::multimap

Когда большое количество элементов нужно сохранить в упорядоченном виде, а ключи могут встречаться несколько раз, пригодится контейнер `std::multimap`.

Придумаем пример, где можно было бы это использовать. В текстах на немецком языке нередко встречаются очень длинные предложения, что не так актуально для английского. Мы реализуем инструмент, который позволит немецким авторам анализировать текстовые файлы, написанные на английском языке, опираясь на длину всех предложений. Чтобы помочь автору улучшить его стиль, программа

сгруппирует предложения по длине. Таким образом, автор сможет выбрать самые длинные предложения и разбить их на части.

Как это делается

В этом примере мы считаем данные, введенные пользователем, из стандартного потока ввода, и разобьем их на предложения (а не на слова, как делали раньше). Далее поместим все предложения в контейнер `std::multimap` в паре с переменной, в которой записана их длина. После этого выведем пользователю все предложения, отсортировав их по длине.

1. Как обычно, включим все необходимые заголовочные файлы. Контейнер `std::multimap` поставляется оттуда же, откуда и контейнер `std::map`:

```
#include <iostream>
#include <iterator>
#include <map>
#include <algorithm>
```

2. Мы будем применять множество функций из пространства имен `std`, поэтому объявим о его (автоматическом) использовании:

```
using namespace std;
```

3. При токенизации строк путем извлечения их содержимого, стоящего между точками, получим предложения текста, окруженные пробелами, символами перехода на новую строку и т. д. Это нежелательным образом увеличивает их размер, так что отбросим лишние символы с помощью вспомогательной функции, которую сейчас определим:

```
string filter_ws(const string &s)
{
    const char *ws {" \r\n\t"};
    const auto a (s.find_first_not_of(ws));
    const auto b (s.find_last_not_of(ws));
    if (a == string::npos) {
        return {};
    }
    return s.substr(a, b - a + 1);
}
```

4. Функция определения длины предложения станет принимать гигантскую строку, содержащую весь текст, и возвращать контейнер `std::multimap`, в котором будут соотнесены предложения и их длины:

```
multimap<size_t, string> get_sentence_stats(const string &content)
{
```

5. Начнем с объявления структуры `multimap`, которая станет возвращаемым значением, а также нескольких итераторов. Поскольку мы создадим цикл, нам понадобится конечный итератор. Далее воспользуемся двумя итераторами,

чтобы указать на две соседние точки (знаки препинания) внутри текста. Все находящееся между ними будем считать предложением.

```
multimap<size_t, string> ret;
const auto end_it (end(content));
auto it1 (begin(content));
auto it2 (find(it1, end_it, '.'));
```

6. Итератор `it2` всегда будет указывать на одну точку дальше, чем итератор `it1`. До тех пор пока итератор `it1` не достигнет конца текста, все будет в порядке. Второе условное выражение проверяет, действительно ли итератор `it2` указывает на позицию, стоящую на несколько символов дальше. Если это не так, то у нас не осталось непрочитанных символов.

```
while (it1 != end_it && distance(it1, it2) > 0) {
```

7. Создаем строку из всех символов между итераторами, после чего удаляем лишние пробельные символы из ее начала и конца, чтобы определить точную длину предложения:

```
    string s {filter_ws({it1, it2})};
```

8. Возможно, приложение не содержит ничего, кроме пробельных символов. В этом случае просто отбрасываем его. В противном случае установим его длину, определив количество слов. Это делается легко, поскольку все слова разделены одним пробелом[1]. Затем сохраняем количество слов и само предложение в контейнер `multimap`:

```
    if (s.length() > 0) {
        const auto words (count(begin(s), end(s), ' ') + 1);
        ret.emplace(make_pair(words, move(s)));
    }
```
[2]

9. На следующей итерации цикла мы переводим ведущий итератор `it2` на символ точки в следующем предложении. Догоняющий итератор `it1` переводим на следующий символ относительно *предыдущей* позиции ведущего итератора:

```
    it1 = next(it2, 1);
    it2 = find(it1, end_it, '.');
}
```
[3]

---

[1] Это предположение накладывает соответствующее ограничение на содержимое вводимого текста.

[2] Автор забыл вставить выход из цикла при достижении конца введенного текста (это исправление есть в репозитории с кодом к книге):

```
if (it2 == end_it) {
  break;
}
```

[3] Цифра 1 совпадает со значением аргумента по умолчанию. Кроме того, это еще одно ограничение на вводимый текст — точка не обязательно обозначает конец. Это может быть часть многоточия, сокращение, разделитель дробной части числа и т. п.

10. После завершения работы цикла контейнер будет содержать все предложения, объединенные в пары с количеством слов, содержащихся в них. Наконец можно вернуть полученный результат:

```
    return ret;
}
```

11. Теперь воспользуемся вновь добавленной функцией. Сначала укажем `std::cin` не пропускать пробельные символы, поскольку нам нужны предложения с пробелами как единое целое. Чтобы считать весь файл, инициализируем `std::string` на основе итераторов потока ввода, которые инкапсулируют `std::cin`:

```
int main()
{
    cin.unsetf(ios::skipws);
    string content {istream_iterator<char>{cin}, {}};
```

12. Поскольку нужен полученный контейнер `multimap` лишь для того, чтобы вывести на экран результат, помещаем вызов функции `get_sentence_stats` непосредственно в цикл и передаем ему нашу строку. В теле цикла выведем элементы построчно:

```
    for (const auto & [word_count, sentence]
              : get_sentence_stats(content)) {
        cout << word_count << " words: " << sentence << ".\n";
    }
}
```

13. После компиляции кода мы можем дать приложению команду использовать содержимое любого текстового файла. Текст-пример Lorem Ipsum даст следующий результат. Вывод на экран довольно велик для длинных текстов с большим количеством предложений: сначала выводятся самые короткие предложения, а затем — самые длинные. В результате мы сразу видим самые длинные предложения, поскольку обычно по умолчанию на экране отображается нижняя часть вывода:

```
$ cat lorem_ipsum.txt | ./sentence_length[1]
...
10 words: Nam quam nunc, blandit vel, luctus pulvinar,
hendrerit id, lorem.
10 words: Sed consequat, leo eget bibendum sodales,
augue velit cursus nunc,.
12 words: Cum sociis natoque penatibus et magnis dis
parturient montes, nascetur ridiculus mus.
17 words: Maecenas tempus, tellus eget condimentum rhoncus,
sem quam semper libero, sit amet adipiscing sem neque sed ipsum.
```

[1] Можно просто:

./sentence_length < lorem_ipsum.txt

Как это работает

Весь этот пример посвящен разбиению большой строки на предложения, после чего мы определяем их длину и помещаем в контейнер `multimap` в упорядоченном виде. Поскольку контейнер `std::multimap` сам по себе прост в использовании, сложной частью программы является цикл, который проходит по всем предложениям:

```
const auto end_it (end(content));
auto it1 (begin(content));          // (1) Начало строки
auto it2 (find(it1, end_it, '.')); // (1) Первая точка '.'

while (it1 != end_it && std::distance(it1, it2) > 0) {
    string sentence {it1, it2};

    // Что-то делаем со строкой предложения...

    it1 = std::next(it2, 1);      // Один символ справа от текущей точки '.'
    it2 = find(it1, end_it, '.'); // Следующая точка или конец строки
}
```

Взглянем на код, имея при этом в виду следующий рисунок, на котором приведены три предложения (рис. 2.5).

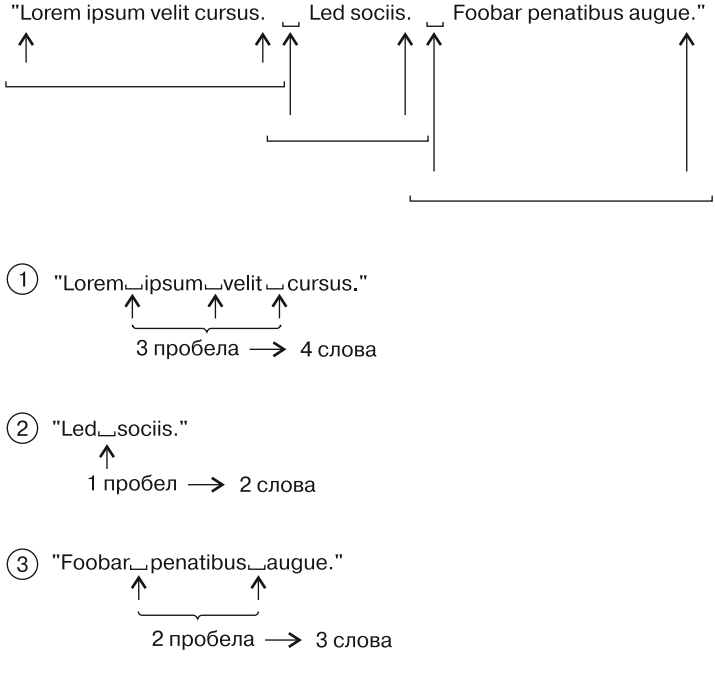

Рис. 2.5

Итераторы `it1` и `it2` всегда перемещаются вперед по строке вместе. Таким образом, они всегда указывают на начало и конец *одного и того же* предложения. Алгоритм `std::find` заметно облегчает нам жизнь, поскольку работает по принципу *«начинаем с текущей позиции, а затем возвращаем итератор, указывающий на следующий символ точки. Если такого итератора нет, то возвращаем конечный итератор»*.

После извлечения строки с предложением определим, сколько слов в ней содержится, а затем вставим ее в контейнер `multimap`. Мы используем *количество слов* в качестве ключа для элементов массива, а саму строку — как объект, связанный с данным ключом. Мы не можем воспользоваться контейнером `std::map`, так как предложений одинаковой длины может быть довольно много. Но это не проблема, поскольку мы задействуем контейнер `std::multimap`, который легко справляется с совпадающими ключами. Данный контейнер хранит ключи *в упорядоченном виде*, что позволяет нам вывести предложения в порядке возрастания их длины.

Дополнительная информация

После того как мы считали весь файл в одну большую строку, мы проходим по ней и создаем копию каждого предложения. Однако это не обязательно, ведь можно воспользоваться `std::string_view`; данный вопрос мы рассмотрим далее в книге.

Еще один способ получить строки между двумя соседними точками — задействовать класс `std::regex_iterator`, который мы также рассмотрим несколько позже.

Реализуем личный список текущих дел с помощью std::priority_queue

Класс `std::priority_queue` — еще один класс-адаптер для контейнеров, как и `std::stack`. Он является оболочкой для другой структуры данных (по умолчанию это `std::vector`) и предоставляет для нее интерфейс очереди. То есть элементы можно помещать туда и выталкивать оттуда пошагово. Все, что было помещено туда *раньше*, *раньше* очередь и покинет. Обычно данный принцип обозначается как **FIFO** (first in, first out — «первый вошел — первый вышел»). Он полностью противоположен принципу работы стека, где *последний* помещенный элемент будет вытолкнут *первым*.

Несмотря на то что мы описали, как работает контейнер `std::queue`, в текущем разделе будет показана работа контейнера `std::priority_queue`. Он особенный, поскольку не только имеет характеристики FIFO, но и объединяет их с приоритетами. Иными словами, содержимое, по сути, разбивается на подочереди, работающие по принципу FIFO, которые упорядочены в соответствии с указанным для них приоритетом.

Как это делается

В данном примере мы создадим простую структуру, которая может служить в качестве *списка текущих дел*. Для сокращения программы мы не будем анализировать входные данные и сконцентрируемся на `std::priority_queue`. Поэтому просто заполняем очередь с приоритетом на основе неупорядоченного списка элементов, имеющих приоритет и описание. Затем считываем их как из структуры данных, работающей по принципу очереди FIFO, где все элементы сгруппированы по приоритетам отдельных элементов.

1. Сначала включим некоторые заголовочные файлы. Контейнер `std::priority_queue` располагается в заголовочном файле `<queue>`:

```
#include <iostream>
#include <queue>
#include <tuple>
#include <string>
```

2. Как же мы сохраняем элементы списка дел в очереди с приоритетом? Проблема заключается в том, что нельзя просто добавить элемент и дополнительно прикрепить к нему приоритет. Очередь с приоритетом попытается использовать *естественный порядок* всех элементов очереди. Можно реализовать собственную структуру `todo_item` и задать для нее число, указывающее на приоритет, и строку с описанием дела, а затем реализовать оператор сравнения `<`, чтобы впоследствии упорядочить данные элементы. Или же можно просто задействовать тип `std::pair`; это позволит объединить два свойства в одном типе, при этом сравнение уже реализовано за нас:

```
int main()
{
    using item_type = std::pair<int, std::string>;
```

3. Сейчас у нас имеется новый тип `item_type`, содержащий число, описывающее приоритет, и строку-описание. Поэтому создадим экземпляр очереди с приоритетом, в которой будут находиться такие элементы:

```
    std::priority_queue<item_type> q;
```

4. Теперь заполним эту очередь разными элементами, имеющими разные приоритеты. Нам следует предоставить *неструктурированный* список, а затем очередь с приоритетом укажет, *что* сделать и *в каком порядке*. Например, если нужно прочитать комикс и выполнить домашнюю работу, то последняя должна находиться выше в списке. К сожалению, класс `std::priority_queue` не имеет конструктора, принимающего списки инициализации, который мы могли бы использовать для заполнения очереди. (Это сработало бы, примени мы вектор или обычный список.) Так что сначала определим список и заполним его на следующем этапе:

```
    std::initializer_list<item_type> il {
        {1, "dishes"},
```

```
        {0, "watch tv"},
        {2, "do homework"},
        {0, "read comics"},
    };
```

5. Теперь можно легко проитерировать по неупорядоченному списку текущих дел и вставить их по одному с помощью функции `push`:

```
    for (const auto &p : il) {
        q.push(p);
    }
```

6. Все элементы будут упорядочены неявным образом, и теперь у нас есть очередь, которая выдает элементы с наивысшим приоритетом:

```
    while(!q.empty()) {
        std::cout << q.top().first << ": " << q.top().second << '\n';
        q.pop();
    }
    std::cout << '\n';
}
```

7. Скомпилируем и запустим нашу программу. Она сообщает следующее: сначала мы должны выполнить домашнюю работу, а затем, после того как помоем посуду, можем посмотреть телевизор и почитать комиксы:

```
$ ./main
2: do homework
1: dishes
0: watch tv
0: read comics
```

Как это работает

Контейнер `std::priority_queue` очень прост в использовании. Нам понадобилось всего три функции.

1. `q.push(item)` помещает элемент в очередь.
2. `q.top()` возвращает ссылку на элемент, который первым покинет очередь.
3. `q.pop()` удаляет первый элемент из очереди.

Но каким образом происходит упорядочение элементов? Мы сгруппировали числа, указывающие на приоритет, и строки, описывающие элементы списка текущих дел, в объекты типа `std::pair`, что позволило упорядочить элементы автоматически. Если у нас есть экземпляр `p` типа `std::pair<int, std::string>`, то с помощью нотации `p.first` можно получить *число*, а благодаря нотации `p.second` — *строку*. Мы сделали это в цикле, где выводятся на экран все наши текущие дела.

Но как очередь с приоритетом узнала, что пара `{2, "do homework"}` *важнее*, чем `{0, "watch tv"}`? Мы ведь не говорили ей сравнивать числовую часть.

Оператор сравнения < по-разному обрабатывает разные ситуации. Предположим, у нас имеется сравнение `left < right`, где `left` и `right` представляют собой пары.

1. Если выполняется условие `left.first != right.first`, то возвращается результат сравнения `left.first < right.first`.
2. Если выполняется условие `left.first == right.first`, то возвращается результат сравнения `left.second < right.second`.

Таким образом можно упорядочить все, что нам угодно. Важным здесь является тот факт, что приоритет — *первый* член пары, а описание — *второй*. В противном случае контейнер `std::priority_queue` упорядочил бы элементы по алфавиту, а не по числам, указывающим на приоритеты. (В данном случае очередь предложит нам *сначала* посмотреть телевизор (watch TV), а затем, *спустя какое-то время*, заняться домашней работой (do homework). Это бы понравилось самым ленивым из нас!)

3 Итераторы

В этой главе:

- построение собственного итерабельного диапазона данных;
- обеспечение совместимости ваших итераторов с категориями итераторов STL;
- использование оболочек для итераторов для заполнения обобщенных структур данных;
- реализация алгоритмов с помощью итераторов;
- перебор (итерирование) в обратную сторону с применением обратных адаптеров для итераторов;
- завершение перебора диапазонов данных с использованием ограничителей;
- автоматическая проверка кода итератора с помощью проверяемых итераторов;
- создание собственного адаптера для итераторов-упаковщиков.

Введение

Итераторы — крайне важная концепция языка C++. Библиотека STL создавалась максимально гибкой и обобщенной, а итераторы способствуют этому. К сожалению, иногда применять их несколько утомительно, и многие новички избегают их и возвращаются к стилю программирования, свойственному языку C. Программист, который избегает использования итераторов, по сути, отказывается от половины потенциала библиотеки STL. В данной главе мы рассмотрим итераторы, постаравшись разобраться в их достоинствах и недостатках. Надеюсь, показанные примеры помогут вам понять основные принципы работы с итераторами.

Многие классы-контейнеры, а также массивы в стиле C так или иначе содержат диапазон неких элементов. Для выполнения множества повседневных задач, связанных с обработкой больших объемов данных, не нужно знать, как эти данные были получены. Однако если у нас есть, например, массив целых чисел и список,

содержащий целые числа, то следует воспользоваться двумя разными алгоритмами, представленными ниже.

- Один алгоритм, который работает с массивом, проверяя его размер и суммируя все его члены, выглядит так:

```
int sum {0};
for (size_t i {0}; i < array_size; ++i) { sum += array[i]; }
```

- Другой алгоритм, который работает со связанным списком и итерирует по нему до конца, выглядит следующим образом:

```
int sum {0};
while (list_node != nullptr) {
    sum += list_node->value; list_node = list_node->next;
}
```

Оба алгоритма *суммируют целые числа*, но какая часть введенных нами символов *непосредственно* связана с решением задачи? Работает ли какой-нибудь из этих алгоритмов с другими видами структур данных, например с `std::map`, или нужно реализовывать еще одну версию алгоритма суммирования? Отсутствие итераторов приведет к нелепым решениям.

Только с помощью итераторов можно реализовать этот алгоритм в обобщенном виде:

```
int sum {0};
for (int i : array_or_vector_or_map_or_list) { sum += i; }
```

Это красивое и короткое выражение, названное «основанный на диапазоне цикл `for`», существует еще со времен C++11. Оно представляет собой лишь синтаксический сахар, который развертывается в нечто похожее на следующий код:

```
{
    auto && __range = array_or_vector_or_map_or_list ;
    auto __begin = std::begin(__range);
    auto __end   = std::end(__range);
    for ( ; __begin != __end; ++__begin) {
        int i = *__begin;
        sum += i;
    }
}
```

Такие циклы хорошо знакомы всем, кто уже работал с итераторами, но кажутся черной магией для тех, кто еще этого не делал. Представьте, что наш вектор, содержащий целые числа, выглядит следующим образом (рис. 3.1).

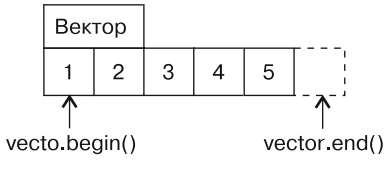

Рис. 3.1

Команда `std::begin(vector)` аналогична команде `vector.begin()`, она возвращает итератор, который указывает на первый элемент (1). Команда `std::end(vector)` аналогична команде `vector.end()`, она возвращает итератор, указывающий на элемент, стоящий *за последним* элементом (5).

На каждой итерации цикл проверяет, равен ли начальный итератор конечному. Если это не так, то мы *разыменовываем* начальный итератор и получаем доступ к числовому значению, на которое он указывает. Далее выполняем операцию *инкремента* для итератора, повторяем сравнение с конечным итератором и т. д. В этот момент полезно прочесть код цикла снова, представляя, что итераторы — обычные указатели, взятые из языка C. Фактически такие указатели тоже являются итераторами.

Категории итераторов

Существует несколько категорий итераторов, каждая из которых имеет разные ограничения. Их несложно запомнить, однако имейте в виду: возможности, требуемые одной категорией, унаследованы из другой, более мощной. Вся суть категорий итераторов заключается в том, что если при реализации алгоритма вы знаете, с каким именно итератором будете работать, то сможете реализовать оптимизированную версию. Таким образом, программисту достаточно просто выразить свое намерение, а компилятор выберет *оптимальную реализацию* для поставленной задачи.

Рассмотрим их в правильном порядке (рис. 3.2).

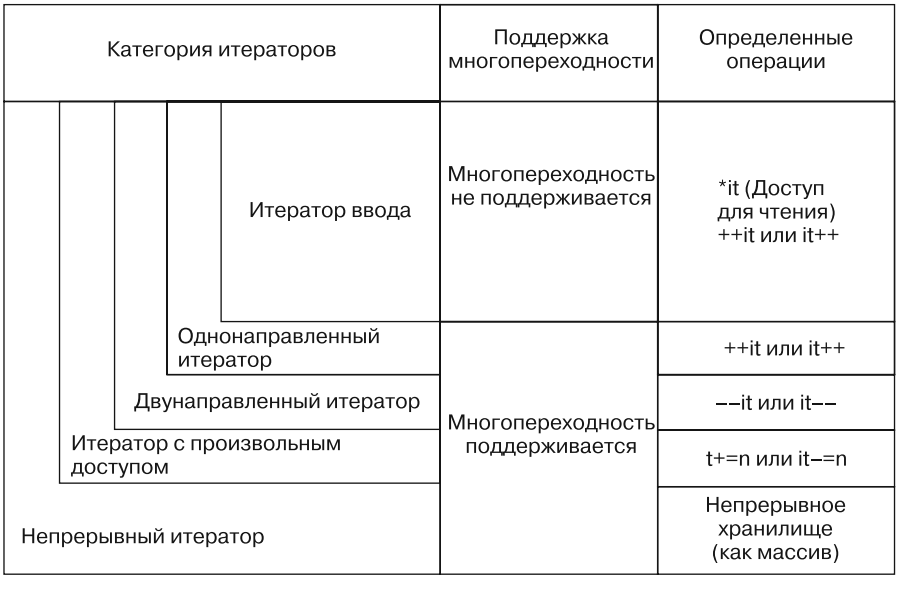

Рис. 3.2

Итераторы ввода

Могут быть разыменованы только для того, чтобы *прочесть* значение, на которое указывают. При их инкрементировании последнее значение, на которое они указывают, становится *недействительным* во время данной операции, а значит, вы не можете итерировать по такому диапазону данных несколько раз. Примером итератора этой категории является `std::istream_iterator`.

Однонаправленные итераторы

Аналогичны итераторам ввода, но отличаются тем, что по диапазонам данных, которые они представляют, вы можете проитерировать несколько раз. В качестве примера такого итератора приведем `std::forward_list`. По такому списку можно проитерировать только *вперед*, но не назад, однако это можно сделать требуемое количество раз.

Двунаправленные итераторы

Как следует из названия, их можно инкрементировать и декрементировать, что позволяет итерировать вперед и назад. Эту возможность, например, поддерживают итераторы для контейнеров `std::list`, `std::set` и `std::map`.

Итераторы с произвольным доступом

Позволяют перескакивать через несколько значений сразу вместо того, чтобы двигаться пошагово. Примером таких итераторов являются итераторы для контейнеров `std::vector` и `std::deque`.

Непрерывные итераторы

Соответствуют всем указанным выше требованиям, но при этом необходимо, чтобы данные, по которым выполняется итерирование, находились в непрерывной памяти, как, например, в массиве `std::vector`.

Итераторы вывода

Вынесены в отдельную категорию. Причина такова: итератор может быть чистым итератором вывода, который можно только инкрементировать и использовать для *записи* данных в указываемое им место. При выполнении операции чтения значение будет неопределенным.

Изменяемые итераторы

Если итератор является итератором вывода, а также состоит в какой-то другой категории, то называется изменяемым. С его помощью можно считывать и записывать данные. При получении из неконстантного экземпляра контейнера итератор, как правило, будет состоять именно в этой категории.

Создаем собственный итерабельный диапазон данных

Мы уже знаем, что итераторы в некоторой степени представляют собой *стандартный интерфейс* для выполнения перебора во всех видах контейнеров. Нужно только реализовать оператор префиксного инкремента ++, оператор разыменования * и оператор сравнения объектов ==, и получится примитивный итератор, подходящий для работы с циклом for, основанным на диапазоне, который появился в C++11.

Чтобы немного освоиться с итераторами, рассмотрим пример реализации одного из них, который просто генерирует диапазон чисел при переборе. Он не дополняется структурой-контейнером. Числа генерируются непосредственно при переборе.

Как это делается

В этом примере мы реализуем собственный класс итератора, а затем проитерируем по нему.

1. Сначала включим заголовочный файл, который позволит выводить данные на консоль:

```
#include <iostream>
```

2. Наш класс итератора будет называться num_iterator:

```
class num_iterator {
```

3. Его единственным членом выступит целое число, которое послужит для счета. Оно будет инициализироваться в конструкторе. Создание *явных* конструкторов — хороший стиль программирования, поскольку это позволяет избежать случайных *неявных* преобразований. Обратите внимание: мы предоставляем значение по умолчанию для переменной position, что делает возможным создание экземпляров класса num_iterator с помощью конструктора по умолчанию. Хотя в данном примере мы не будем использовать такой конструктор, эта возможность очень важна, поскольку некоторые алгоритмы STL зависят от того, можно ли создать экземпляры итераторов, применяя конструкторы по умолчанию:

```
    int i;
public:

    explicit num_iterator(int position = 0) : i{position} {}
```

4. При разыменовании наш итератор (*it) генерирует целое число:

```
    int operator*() const { return i; }
```

5. Инкрементирование итератора (`++it`) просто увеличит значение его внутреннего счетчика `i`:

```
    num_iterator& operator++() {
        ++i;
        return *this;
    }
```

6. Цикл `for` будет сравнивать итератор с конечным итератором. Если они *не равны*, то продолжим перебор:

```
    bool operator!=(const num_iterator &other) const {
        return i != other.i;
    }
};
```

7. Это был класс итератора. Нам все еще нужен промежуточный объект для записи `for (int i : intermediate(a, b)) {...}`, который содержит начальный и конечный итераторы и будет перепрограммирован так, чтобы итерировал от `a` до `b`. Мы назовем его `num_range`:

```
class num_range {
```

8. Он содержит два члена, представляющие собой целые числа. Они обозначают число, с которого начнется перебор, а также число, стоящее непосредственно за последним числом. Это значит, что если мы хотим проитерировать по числам от `0` до `9`, то `a` будет иметь значение `0`, а `b` — `10`:

```
    int a;
    int b;

public:
    num_range(int from, int to)
        : a{from}, b{to}
    {}
```

9. Нужно реализовать всего две функции-члена: `begin` и `end`. Обе эти функции возвращают итераторы, которые указывают на начало и конец численного диапазона:

```
    num_iterator begin() const { return num_iterator{a}; }
    num_iterator end()   const { return num_iterator{b}; }
};
```

10. На этом все. Можно использовать полученный объект. Напишем функцию `main`, в которой просто проитерируем по диапазону значений от `100` до `109` и выведем эти значения:

```
int main()
{
    for (int i : num_range{100, 110}) {
        std::cout << i << ", ";
```

```
        }
        std::cout << '\n';
    }
```

11. Компиляция и запуск программы дадут следующий результат:

```
100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
```

Как это работает

Представьте, что мы написали следующий код:

```
for (auto x : range) { code_block; }
```

Компилятор развернет его в такую конструкцию:

```
{
    auto __begin = std::begin(range);
    auto __end   = std::end(range);
    for ( ; __begin != __end; ++__begin) {
        auto x = *__begin;
        code_block
    }
}
```

При взгляде на этот код становится очевидно, что для создания итератора необходимо реализовать всего три оператора:

- `operator!=` — определение равенства;
- `operator++` — префиксный инкремент;
- `operator*` — разыменование.

Требования к диапазону данных заключаются в том, что он должен иметь методы `begin` и `end`, которые будут возвращать два итератора для обозначения начала и конца диапазона.

В данной книге мы будем использовать преимущественно std::begin(x) вместо x.begin(). Это хороший вариант, поскольку функция std::begin(x) автоматически вызывает метод x.begin(), при условии, что он доступен. Если x представляет собой массив, не имеющий метода begin(), то функция std::begin(x) автоматически определит, как с этим справиться. То же верно и для std::end(x). Пользовательские типы, не имеющие методов begin()/end(), не смогут работать с методами std::begin/std::end.

В рамках этого примера мы разместили простой алгоритм счета в интерфейсе однонаправленного итератора. Реализация итератора и диапазона данных зачастую включает в себя написание минимального объема стереотипного кода, что, с одной

стороны, может слегка раздражать. С другой стороны, взглянув на цикл, который использует `num_range`, мы понимаем, что это здорово, поскольку цикл выглядит *очень просто*!

Пролистайте книгу назад и еще раз взгляните на то, какие методы итератора и диапазон классов являются константными. Если не сделать их таковыми, то компилятор во многих ситуациях может отклонить ваш код, поскольку перебор константных объектов происходит довольно часто.

Обеспечиваем совместимость собственных итераторов с категориями итераторов STL

Какую бы структуру данных вы ни создали, для эффективного *объединения* ее с библиотекой STL нужно добавить интерфейсы для итераторов. В последнем разделе мы научились делать это, но затем быстро поняли, что *некоторые* алгоритмы STL *плохо компилируются* с нашими итераторами. Почему так происходит?

Проблема заключается в том, что многие алгоритмы STL пытаются больше узнать об итераторах, с которыми должны работать. Разные *категории* итераторов имеют разные возможности, и поэтому существует несколько вариантов реализации *одного* алгоритма. Например, *обычные числа* из одного вектора в другой можно скопировать с помощью быстрого вызова `memcpy`. Если мы копируем данные из списка или в него, то такой вызов сделать *нельзя* и элементы нужно копировать по одному. Авторы алгоритмов STL хорошо продумали подобную автоматическую оптимизацию. Чтобы помочь им, мы укажем некоторую *информацию* о наших итераторах. В этом разделе показано, как достичь той же цели.

Как это делается

В этом примере мы реализуем примитивный итератор, считающий числа, и используем его вместе с алгоритмом STL, с которым он изначально не будет компилироваться. Затем сделаем все, чтобы итератор стал совместим с STL.

1. Сначала, как обычно, включим некоторые заголовочные файлы:

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

2. Далее реализуем примитивный итератор для подсчета чисел, как было показано в предыдущем разделе. При переборе он генерирует обычные увеличивающиеся целые числа. Диапазон данных `num_range` выступает в роли удобного донора *начального* и *конечного* итераторов:

```
class num_iterator
{
```

```
    int i;
public:
    explicit num_iterator(int position = 0) : i{position} {}
    int operator*() const { return i; }
    num_iterator& operator++() {
        ++i;
        return *this;
    }
    bool operator!=(const num_iterator &other) const {
        return i != other.i;
    }
    bool operator==(const num_iterator &other) const {
        return !(*this != other);
    }
};
class num_range {
    int a;
    int b;
public:
    num_range(int from, int to)
        : a{from}, b{to}
    {}
    num_iterator begin() const { return num_iterator{a}; }
    num_iterator end()   const { return num_iterator{b}; }
};
```

3. Чтобы избавиться от префикса пространства имен `std::` и поддерживать код читабельным, объявим об использовании пространства имен `std`:

```
using namespace std;
```

4. Теперь просто создадим диапазон данных, содержащий числа от `100` до `109`. Обратите внимание: конечный итератор стоит в позиции `110`. Это значит, что `110` — *первое* число, которое находится за пределами диапазона (именно поэтому он начинается со значения `100` и заканчивается значением `109`):

```
int main()
{
    num_range r {100, 110};
```

5. Теперь воспользуемся им для `std::minmax_element`. Алгоритм возвращает объект типа `std::pair` с двумя членами: итератором, указывающим на минимальное значение, и другим итератором, указывающим на максимальное значение. В нашем примере этими значениями будут `100` и `109`, поскольку именно с их помощью мы создавали диапазон данных:

```
    auto [min_it, max_it] (minmax_element(begin(r), end(r)));
    cout << *min_it << " - " << *max_it << '\n';
}
```

6. Компиляция этого кода приведет к следующему сообщению об ошибке (рис. 3.3). Вы увидите ошибку, связанную с `std::iterator_traits`. Подробнее об этом вы

узнаете позже. Может случиться так, что вы увидите другую ошибку, если применяете другие компиляторы и/или реализацию библиотеки STL; а возможно даже, что ошибок не будет. Сообщение об ошибке, показанное на рис. 3.3, появляется при использовании компилятора clang version 5.0.0 (trunk 299766).

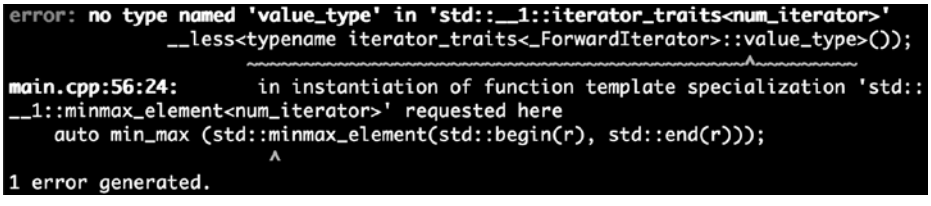

Рис. 3.3

7. Для исправления ситуации активизируем возможности типажей итераторов для нашего класса итератора. Сразу после определения `num_iterator` укажем следующую спецификацию шаблона структуры для типа `std::iterator_traits`. Она сообщает STL, что наш итератор `num_iterator` является однонаправленным и итерирует по целочисленным значениям:

```
namespace std {
template <>
struct iterator_traits<num_iterator> {
    using iterator_category = std::forward_iterator_tag;
    using value_type        = int;
};
}
```

8. Скомпилируем код снова; мы видим, что он работает! Результат работы функций `min/max` выглядит следующим образом:

```
100 – 109
```

Как это работает

Одним алгоритмам STL нужно знать характеристики типа итератора, применяемого вместе с ними, другим — тип элементов, среди которых выполняется перебор. От этого зависят варианты реализации алгоритмов.

Однако все алгоритмы STL будут получать эту информацию о типе с помощью `std::iterator_traits<my_iterator>`, предполагая, что итератор имеет тип `my_iterator`. Этот класс типажа содержит до пяти разных определений членов.

- `difference_type` — значение какого типа мы получим в результате выполнения конструкции `it1 – it2`?
- `value_type` — какой тип имеет элемент, к которому мы получаем доступ с помощью `*it` (обратите внимание: для чистых итераторов вывода этот тип будет `void`)?

- `pointer` — к какому типу должен относиться указатель, чтобы иметь возможность указывать на элемент?
- `reference` — какой тип должна иметь ссылка, чтобы она могла работать как полагается?
- `iterator_category`: к какой категории принадлежит итератор?

Определения типов `pointer`, `reference` и `difference_type` не нужны для нашего итератора `num_iterator`, поскольку он не итерирует по реальным значениям *памяти* (мы просто *возвращаем* значения `int`, но они не доступны на постоянной основе, как это было бы в случае использования массива). Поэтому лучше не определять их, поскольку если алгоритм зависит от доступности элементов диапазона в памяти, то при работе с нашим итератором он может генерировать *ошибки*.

Дополнительная информация

До появления C++17 поощрялось наследование итераторами типа `std::iterator<...>`, что автоматически заполняет наш класс всеми описаниями типов. Этот механизм все еще работает, но, начиная с C++17, больше не рекомендуется.

Используем оболочки итераторов для заполнения обобщенных структур данных

Часто требуется заполнить некий контейнер большим количеством данных, но источник данных и контейнер *не имеют общего интерфейса*. В таких ситуациях нужно создать собственные алгоритмы, которые помогают разместить данные из источника в приемник. Обычно это отвлекает от реальной работы по решению конкретной проблемы.

Задачу по перемещению данных между концептуально разными структурами можно решить с помощью одной строки кода благодаря абстракциям, предоставляемым *адаптерами итераторов* из библиотеки STL. В этом разделе мы рассмотрим некоторые из них, чтобы вы могли получить представление о том, насколько они полезны.

Как это делается

В этом примере мы используем оболочки для итераторов, просто чтобы продемонстрировать их наличие и способы, которые могут помочь в решении часто встречающихся задач.

1. Сначала включим необходимые заголовочные файлы.

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

```
#include <iterator>
#include <sstream>
#include <deque>
```

2. Объявим об использовании пространства имен `std`, чтобы сэкономить время.

```
using namespace std;
```

3. Начнем с итератора `std::istream_iterator`. Мы специализируем его для типа `int`. Таким образом, он станет преобразовывать данные из стандартного потока ввода в целые числа. Например, при итерировании по нему он будет выглядеть так, будто имеет тип `std::vector<int>`. Конечный итератор имеет тот же тип, но не принимает аргументы конструктора.

```
int main()
{
    istream_iterator<int> it_cin {cin};
    istream_iterator<int> end_cin;
```

4. Далее создадим экземпляр типа `std::deque<int>` и просто скопируем туда все целые числа из стандартного потока ввода. Двусторонняя очередь сама по себе не является итератором, поэтому обернем ее в `std::back_insert_iterator` с помощью вспомогательной функции `std::back_inserter`. Эта особая оболочка для итератора позволит выполнить метод `v.push_back(item)` для каждого из элементов, получаемых из стандартного ввода. Таким образом, двусторонняя очередь будет расти автоматически!

```
    deque<int> v;
    copy(it_cin, end_cin, back_inserter(v));
```

5. Воспользуемся `std::istringstream`, чтобы скопировать элементы в *середину* двусторонней очереди. Поэтому определим несколько чисел в форме строки и создадим на их основе объект типа `stream`:

```
    istringstream sstr {"123 456 789"};
```

6. Далее понадобится подсказка, которая поможет определить, куда именно нужно вставить элемент. Нам нужна середина, поэтому применим начальный указатель, передав его в функцию `std::next`. Второй аргумент данной функции говорит о том, что вернет итератор, смещенный на позицию `v.size() / 2` шагов, то есть на половину очереди. (Мы преобразуем `v.size()` к типу `int`, поскольку второй параметр функции `std::next` представляет собой `difference_type` итератора, использованного в качестве первого параметра. В данном случае это целочисленный тип со знаком. В зависимости от установленных флажков компилятор может предупредить вас, если преобразование не было выполнено явно.)

```
    auto deque_middle (next(begin(v),
                            static_cast<int>(v.size()) / 2));
```

7. Теперь можно скопировать проанализированные целые числа шаг за шагом из потока ввода строк в очередь. Опять же конечный итератор оболочки итератора

потока представляет собой пустой объект типа `std::istream_iterator<int>` без аргументов конструктора (поэтому вы можете видеть пустые скобки `{}` в строке кода). Очередь оборачивается в итератор вставки, который представляет собой `std::insert_iterator`, указывающий на середину очереди с помощью итератора `deque_middle`:

```
copy(istream_iterator<int>{sstr}, {}, inserter(v, deque_middle));
```

8. Воспользуемся итератором `std::front_insert_iterator`, чтобы вставить элементы в начало очереди:

```
initializer_list<int> il2 {-1, -2, -3};
copy(begin(il2), end(il2), front_inserter(v));
```

9. На последнем шаге выведем содержимое очереди на консоль. Здесь итератор `std::ostream_iterator` работает как итератор вывода, что в нашем случае просто перенаправляет все скопированные целые числа в `std::cout`, прикрепляя символ `", "` после каждого элемента:

```
copy(begin(v), end(v), ostream_iterator<int>{cout, ", "});
cout << '\n';
}
```

10. Компиляция и запуск программы дадут следующий результат. Можете ли вы определить, какие строки кода добавили каждое число?

```
$ echo "1 2 3 4 5" | ./main
-3, -2, -1, 1, 2, 123, 456, 789, 3, 4, 5,
```

Как это работает

Мы использовали множество разных адаптеров. Они имеют только одну схожую черту — оборачивают объект, не являющийся итератором сам по себе, в итератор.

std::back_insert_iterator

В адаптер `back_insert_iterator` можно обернуть типы `std::vector`, `std::deque`, `std::list` и т. д. Он будет вызывать метод контейнера `push_back`, который вставит новый элемент *после* всех существующих элементов. Если объект контейнера недостаточно велик, то его размер увеличится автоматически.

std::front_insert_iterator

Адаптер `front_insert_iterator` делает то же самое, что и адаптер `back_insert_iterator`, но вызывает метод контейнера `push_front`, который вставляет новый элемент *перед* существующими. Обратите внимание: для контейнеров наподобие `std::vector` это значит, что все существующие элементы нужно сдвинуть на одну позицию вправо для освобождения места под новый элемент.

std::insert_iterator

Этот адаптер итератора аналогичен другим итераторам вставки, но может вставлять новые элементы *между* существующими. Вспомогательная функция `std::inserter`, которая создает подобную оболочку, принимает два аргумента. Первый — контейнер, а второй — итератор, указывающий на позицию, в которую будут вставлены новые элементы.

std::istream_iterator

Адаптер `istream_iterator` тоже довольно удобен. Он подходит для любого объекта `std::istream` (который может представлять собой стандартный поток ввода или файлы) и будет пытаться преобразовывать полученные данные в соответствии с параметром шаблона. В этом примере мы использовали конструкцию `std::istream_iterator<int>(std::cin)`, получающую из потока ввода целые числа.

Как правило, мы не знаем длину потока. Это оставляет открытым вопрос: куда указывает *конечный* итератор, если нам неизвестно, где заканчивается поток? Суть в том, что итератор знает, когда достигает конца потока. При сравнении с конечным итератором он фактически не будет сравнивать себя с конечным итератором, но даст знать, остались ли токены в потоке. Именно поэтому конструктор конечного итератора не принимает никаких аргументов.

std::ostream_iterator

Адаптер `ostream_iterator` аналогичен адаптеру `istream_iterator`, но работает по обратному принципу: не принимает токены *из потока ввода*, а отправляет их *в поток вывода*. Еще одно отличие заключается в том, что его конструктор принимает второй аргумент, являющийся строкой, которая будет помещена в поток вывода после каждого элемента. Это полезно, поскольку таким способом можно вывести на экран разделяющую запятую `", "` или символ перехода на новую строку.

Реализуем алгоритмы с помощью итераторов

Итераторы обычно итерируют, *переходя* с одного элемента контейнера на другой. Но не обязательно использовать итераторы только для того, чтобы итерировать по структурам данных. Они вполне подходят для реализации алгоритмов, в таком случае будут высчитывать следующее значение при инкрементировании (`++it`) и возвращать это значение при разыменовании (`*it`).

В данном разделе мы рассмотрим этот принцип, реализовав функцию, выводящую на экран последовательность чисел Фибоначчи с помощью итераторов. Такая функция рекурсивно определяется следующим образом: `F(n) = F(n - 1) + F(n - 2)`. Она начинается со стартовых значений `F(0) = 0` и `F(1) = 1`. Это приводит к появлению такой последовательности чисел:

- `F(0) = 0`;
- `F(1) = 1`;

- `F(2) = F(1) + F(0) = 1`;
- `F(3) = F(2) + F(1) = 2`;
- `F(4) = F(3) + F(2) = 3`;
- `F(5) = F(4) + F(3) = 5`;
- `F(6) = F(5) + F(4) = 8`;
- `...` и т. д.

Если мы реализуем это в форме вызываемой функции, которая возвращает значение Фибоначчи для любого числа *n*, то получим рекурсивную функцию, вызывающую саму себя, или цикл. Такой результат приемлем, но что, если мы напишем программу, принимающую числа Фибоначчи по некоему шаблону одно за другим? У нас есть два варианта: либо выполнять все рекурсивные вызовы для каждого числа Фибоначчи (это, по сути, растрата вычислительного времени), либо сохранять два последних числа во временных переменных и использовать их для вычисления следующего. Во втором случае придется *объединить* код функции Фибоначчи и код остальной части нашей программы, которая решает другую задачу:

```
size_t a {0};
size_t b {1};

for (size_t i {0}; i < N; ++i) {
    const size_t old_b {b};
    b += a;
    a  = old_b;
    // сделаем что-нибудь с b, текущим числом Фибоначчи
}
```

Итераторы позволяют решить задачу оригинальным способом. Можно обернуть шаги, которые нужно сделать в реализации, основанной на цикле функции Фибоначчи, в префиксный оператор `++` *итератора*. В данном разделе вы увидите, как просто это сделать.

Как это делается

В этом примере мы сконцентрируемся на реализации итератора, который генерирует числа на основе последовательности чисел Фибоначчи.

1. Чтобы иметь возможность вывести на экран числа Фибоначчи, включим соответствующий заголовочный файл:

   ```
   #include <iostream>
   ```

2. Определим итератор Фибоначчи, `fibit`. Он будет содержать член `i`, в котором будет сохраняться индекс позиции в последовательности Фибоначчи, а также члены `a` и `b`, в которых будут храниться два последних значения Фибоначчи. При вызове конструктора по умолчанию итератор Фибоначчи инициализируется значениями `F(0)`.

   ```
   class fibit
   {
   ```

```
    size_t i {0};
    size_t a {0};
    size_t b {1};
```

3. Далее определим стандартный конструктор и конструктор, который позволит инициализировать итератор любым этапом вычисления чисел Фибоначчи:

```
public:
    fibit() = default;
    explicit fibit(size_t i_)
        : i{i_}
    {}
```

4. Разыменование итератора (`*it`) вернет текущее число Фибоначчи на данном шаге:

```
    size_t operator*() const { return b; }
```

5. При инкрементировании итератор (`++it`) переместится на следующее число Фибоначчи. Эта функция содержит тот же код, что и реализация функции Фибоначчи, основанная на цикле:

```
    fibit& operator++() {
        const size_t old_b {b};
        b += a;
        a = old_b;
        ++i;
        return *this;
    }
```

6. При использовании в цикле инкрементированный итератор сравнивается с конечным итератором, для чего следует реализовать оператор `!=`. Мы выполняем сравнение только на том шаге, на котором в данный момент находятся итераторы Фибоначчи, что позволяет проще определить конечный итератор, скажем, для шага `1000000`, поскольку не нужно выполнять трудоемкие вычисления этого довольно большого числа Фибоначчи *заранее*.

```
    bool operator!=(const fibit &o) const { return i != o.i; }
};
```

7. Чтобы иметь возможность использовать итератор Фибоначчи в основанном на диапазоне цикле `for`, реализуем класс диапазона заранее. Назовем его `fib_range`. Его конструктор будет принимать один параметр, который скажет, насколько далеко нужно проитерировать по диапазону данных:

```
class fib_range
{
    size_t end_n;
public:
    fib_range(size_t end_n_)
        : end_n{end_n_}
    {}
```

8. Функции `begin` и `end` возвращают итераторы, которые указывают на позиции `F(0)` и `F(end_n)`:

```
    fibit begin() const { return fibit{}; }
    fibit end()   const { return fibit{end_n}; }
};
```

9. О'кей, теперь забудем о реализации стереотипного кода, связанного с итераторами. Теперь у нас есть вспомогательный класс, который аккуратно скрывает детали реализации! Выведем на экран первые десять чисел Фибоначчи.

```
int main()
{
    for (size_t i : fib_range(10)) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}
```

10. Компиляция и запуск программы дадут следующий результат:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

Дополнительная информация

Использование этого итератора совместно с библиотекой STL предполагает, что он должен поддерживать класс `std::iterator_traits`. Для того чтобы увидеть, как это делается, взглянем на другой пример, который решает именно эту задачу: *обеспечивает совместимость ваших итераторов с категориями итераторов STL.*

Представим, как эта задача решается с помощью итераторов. Во многих ситуациях у нас получится очень красивый код. Не волнуйтесь насчет производительности: компилятор без малейшего труда оптимизирует программу, отсекая стереотипный код, связанный с итераторами!

Чтобы не усложнять пример, мы ничего с ним не сделали, хотя могли опубликовать итератор Фибоначчи в виде библиотеки. В таком случае станет понятно, что у нее есть один недостаток: экземпляр `fibit`, который создается с помощью параметра конструктора, может быть использован только как конечный итератор, поскольку не содержит корректных значений Фибоначчи. Наша небольшая библиотека не предусматривает подобного варианта применения. Есть несколько способов это исправить.

- Сделать конструктор `fibit(size_t i_)` закрытым и объявить, что класс `fib_range` является дружественным для класса `fibit`. Таким образом, пользователи могут применять его только корректным способом.

- Задействовать особые символы, чтобы помешать пользователям разыменовать конечный итератор. Взгляните на пример, в котором мы делаем именно это: *завершаем перебор диапазонов данных с помощью особых символов*.

Перебор в обратную сторону с применением обратных адаптеров для итераторов

Иногда может быть полезно итерировать по диапазону данных не вперед, а *назад*. Основанный на диапазонах цикл `for`, а также все алгоритмы STL обычно итерируют по диапазонам данных, инкрементируя итераторы, однако перебор (итерирование) в обратную сторону требует выполнения операции *декремента*. Конечно, можно обернуть итераторы в слой, преобразующий операцию инкремента в операцию декремента. Но складывается впечатление, что у нас получится множество стереотипного кода на каждый тип, для которого мы собираемся поддерживать эту возможность.

Библиотека STL предоставляет полезный *обратный адаптер для итератора*, позволяющего задействовать итераторы подобным образом.

Как это делается

В этом примере мы будем разными способами применять обратные итераторы, чтобы показать варианты их использования.

1. Как обычно, включим некоторые заголовочные файлы:

```
#include <iostream>
#include <list>
#include <iterator>
```

2. Далее объявляем об использовании пространства имен `std` с целью сэкономить немного времени:

```
using namespace std;
```

3. Чтобы у нас появился объект, по которому мы сможем итерировать, создадим список, содержащий целые числа:

```
int main()
{
    list<int> l {1, 2, 3, 4, 5};
```

4. Теперь выведем эти числа на экран в обратном порядке. Для этого проитерируем по списку благодаря функциям `rbegin` и `rend` класса `std::list` и выведем данные значения с помощью стандартного потока вывода, используя удобный адаптер `ostream_iterator`:

```
    copy(l.rbegin(), l.rend(), ostream_iterator<int>{cout, ", "});
    cout << '\n';
```

5. Если контейнер не предоставляет функций `rbegin` и `rend`, но хотя бы выдает двунаправленные итераторы, то поможет функция `std::make_reverse_iterator`. Она принимает *обычные* итераторы и преобразует их в *обратные*:

```
    copy(make_reverse_iterator(end(l)),
         make_reverse_iterator(begin(l)),
         ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```

6. Компиляция и запуск программы дадут следующий результат:

```
5, 4, 3, 2, 1,
5, 4, 3, 2, 1,
```

Как это работает

Преобразование обычного итератора в обратный возможно при поддержке двунаправленного перебора. Это требование выполняется любым итератором, который находится в категории *двунаправленных* или выше.

Обратный итератор *содержит* обычный итератор и *подражает* его интерфейсу, но изменяет операцию инкремента на операцию декремента.

Еще одна деталь связана с позициями начала и конца. Взглянем на рис. 3.4, на котором показана стандартная последовательность чисел, хранящаяся в итерабельном диапазоне данных. Если она содержит числа от `1` до `5`, то начальный итератор должен указывать на элемент `1`, а конечный итератор — на элемент, стоящий сразу после `5`.

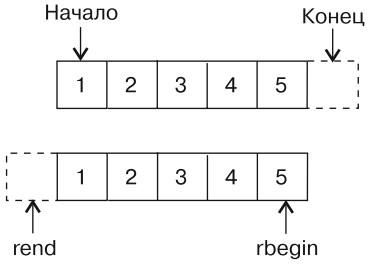

Рис. 3.4

При определении обратных итераторов итератор `rbegin` должен указывать на элемент `5`, а итератор `rend` — на элемент, стоящий сразу *перед* `1`. Переверните книгу вверх ногами — и убедитесь в том, что это имеет смысл.

Если мы хотим, чтобы наши собственные классы-контейнеры поддерживали обратный перебор, то не нужно реализовывать все эти детали самостоятельно, можно просто обернуть обычные итераторы в обратные с помощью вспомогательной функции `std::make_reverse_iterator`, и она сделает всю работу за нас.

Завершение перебора диапазонов данных с использованием ограничителей

Как алгоритмы STL, так и основанные на диапазонах циклы `for` предполагают, что начальная и конечная позиции для перебора известны *заранее*. В некоторых ситуациях, однако, нельзя узнать конечную позицию *до того*, как она будет достигнута при переборе.

Самый простой пример такой ситуации — это перебор в стиле C простых строк, длина которых во время *выполнения* неизвестна. Код, итерирующий по таким строкам, обычно выглядит следующим образом:

```
for (const char *c_ponter = some_c_string; *c_pointer != '\0'; ++c_pointer)
{
    const char c = *c_pointer;
    // сделаем что-нибудь с переменной c
}
```

Единственный способ поработать с этими строками в основанном на диапазоне цикле `for` заключается в том, чтобы обернуть их в объект `std::string`, который поддерживает функции `begin()` и `end()`:

```
for (char c : std::string(some_c_string)) { /* сделаем что-нибудь с c */ }
```

Однако конструктор класса `std::string` будет итерировать по всей строке до того, как этим сможет заняться созданный нами цикл. В C++17 появился класс `std::string_view`, но его конструктор также один раз проитерирует по всей строке. *Короткие* строки не стоят таких хлопот, но это только пример одного из проблемных *классов*, для которого подобная возня может быть оправдана *в других ситуациях*. Итератор `std::istream_iterator` тоже сталкивается с подобными случаями в момент приема входящих данных из `std::cin`, поскольку его конечный итератор не может реалистично указывать на конец потока данных, когда пользователь *еще вводит* текст.

Начиная с C++17 начальный и конечный итераторы не обязаны иметь один тип. В данном разделе мы продемонстрируем, как *правильно использовать* это *небольшое изменение в правилах*.

Как это делается

В этом примере мы создадим итератор и класс диапазона, который позволит проитерировать по строке неизвестной длины, не зная *конечной* позиции *заранее*.

1. Сначала, как и всегда, включим заголовочные файлы:

```
#include <iostream>
```

2. Ограничитель итератора — самый важный элемент этого раздела. Удивительно, но определение его класса остается полностью пустым:

```
class cstring_iterator_sentinel {};
```

3. Теперь реализуем итератор. Он будет содержать указатель на строку, которая и станет тем *контейнером*, по которому мы будем итерировать:

```
class cstring_iterator {
    const char *s {nullptr};
```

4. В конструкторе просто инициализируется внутренний указатель на строку, предоставляемую пользователем. Сделаем конструктор явным, чтобы предотвратить неявные преобразования строк к строковым итераторам:

```
public:
    explicit cstring_iterator(const char *str)
        : s{str}
    {}
```

5. При разыменовании итератор в какой-то момент просто вернет символьное значение в этой позиции:

```
    char operator*() const { return *s; }
```

6. Операция инкремента для итератора просто инкрементирует позицию в строке:

```
    cstring_iterator& operator++() {
        ++s;
        return *this;
    }
```

7. Здесь начинается самое интересное. Мы реализуем оператор сравнения `!=`, который используется алгоритмами STL и основанным на диапазоне циклом `for`. Однако в этот раз мы будем реализовывать его для сравнения итераторов не с другими *итераторами*, а с *ограничителями*. При сравнении итераторов можно проверить только тот факт, что их внутренние указатели на строку указывают на один и тот же адрес; это несколько ограничивает наши возможности. Сравнивая итератор с пустым объектом-ограничителем, можно применить совершенно другую семантику: проверить, указывает ли наш итератор на завершающий символ `'\0'`, поскольку он представляет собой *конец* строки!

```
    bool operator!=(const cstring_iterator_sentinel) const {
        return s != nullptr && *s != '\0';
    }
};
```

8. Чтобы использовать эту возможность в основанном на диапазоне цикле `for`, нужен класс диапазона, который предоставит конечный и начальный итераторы:

```
class cstring_range {
    const char *s {nullptr};
```

9. Единственное, что пользователь должен предоставить при создании экземпляра этого класса, — строка, по которой мы будем итерировать:

```
public:
    cstring_range(const char *str)
        : s{str}
    {}
```

10. Вернем обычный итератор `cstring_iterator` из функции `begin()`, который указывает на начало строки. Из функции `end()` мы вернем *тип ограничителя*. Обратите внимание: без типа ограничителя мы также будем возвращать итератор, но как же узнать о достижении конца строки, если мы не нашли его заранее?

```
    cstring_iterator begin() const {
        return cstring_iterator{s};
    }
    cstring_iterator_sentinel end() const {
        return {};
    }
};
```

11. На этом все. Мы можем мгновенно применить итератор. Строки, которые поступают от пользователя, представляют собой лишь один пример входных данных, чью длину мы не знаем заранее. Чтобы заставить пользователя предоставить какие-нибудь входные данные, мы станем завершать работу программы, если тот не указал хотя бы один параметр при ее запуске в оболочке:

```
int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cout << "Please provide one parameter.\n";
        return 1;
    }
```

12. Если программа все еще работает, то мы знаем, что в `argv[1]` содержится какая-то пользовательская строка:

```
    for (char c : cstring_range(argv[1])) {
        std::cout << c;
    }
    std::cout << '\n';
}
```

13. Компиляция и запуск программы дадут следующий результат:

```
$ ./main "abcdef"
abcdef
```

Цикл выводит на экран введенные нами данные, и это неудивительно, поскольку мы рассмотрели небольшой пример реализации диапазона итераторов на базе ограничителей. Такой способ завершения перебора позволит вам реализовать собственные итераторы, если вы столкнетесь с ситуацией, когда *сравнение с конечной позицией* не помогает.

Автоматическая проверка кода итераторов с помощью проверяемых итераторов

Хотя итераторы очень полезны и предоставляют общие интерфейсы, есть шанс использовать их *неправильно*, как и указатели. При работе с указателями код нужно писать так, чтобы никогда не разыменовывать их в те моменты, когда они указывают на некорректные точки в памяти. То же верно и для итераторов, но для них предусмотрено множество правил, которые позволяют определить, корректен ли итератор. С этими правилами можно ознакомиться, изучив документацию к STL, но вероятность написания кода с ошибками не исчезнет.

В лучшем случае такой код даст сбой при тестировании на машине разработчика, а не на машине клиента. Однако довольно часто код на первый взгляд работает, но при этом в некоторых ситуациях в нем разыменовываются висящие указатели, итераторы и т. д. В таких случаях мы хотим, чтобы нас оповестили, если мы напишем код, демонстрирующий неопределенное поведение.

К счастью, спасение есть! Реализация GNU STL имеет *режим отладки*, а компиляторы GNU C++ и LLVM clang C++ поддерживают *дополнительные библиотеки*, пригодные для создания *сверхчувствительных* и *избыточных* бинарных файлов, которые будут мгновенно сообщать о большинстве ошибок. Их легко использовать, и они очень полезны, что мы и продемонстрируем в этом разделе. Стандартная библиотека Microsoft Visual C++ также предоставляет возможность проведения дополнительных проверок.

Как это делается

В этом примере мы напишем программу, которая намеренно получает доступ к некорректному итератору.

1. Сначала включим заголовочные файлы:

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

2. Теперь создадим вектор, содержащий целые числа, и получим итератор, указывающий на первый элемент, — значение `1`. Мы применим функцию `shrink_to_fit()` для вектора с целью убедиться, что его емкость действительно равна `3`, поскольку данная реализация может выделять фрагмент памяти больше необходимого, чтобы благодаря этому небольшому резерву будущие операции вставки проходили быстрее:

```
int main()
{
    std::vector<int> v {1, 2, 3};
    v.shrink_to_fit();
    const auto it (std::begin(v));
```

3. Далее выведем на экран разыменованный итератор, что совершенно корректно:

```
    std::cout << *it << '\n';
```

4. Добавим в вектор новое число. Поскольку он недостаточно велик, чтобы свободно принять новое число, он автоматически увеличится в размере. Это достигается за счет выделения более крупного фрагмента памяти, перемещения всех существующих элементов в новый фрагмент и удаления *старого*.

```
v.push_back(123);
```

5. Теперь снова выведем на экран значение `1` из вектора с помощью данного итератора. Это кончится плохо. Почему? Когда вектор переместил все свои значения в новый фрагмент памяти и удалил старый, он не сообщил итератору о текущем изменении. То есть итератор все еще указывает на старую позицию, и мы точно не знаем, что с тех пор произошло.

```
    std::cout << *it << '\n'; // плохо плохо плохо!
}
```

6. Компиляция и запуск программы приводят к идеальному выполнению. Приложение не дает сбой, но данные, которые оно выводит на экран при разыменовании некорректного указателя, выглядят совершенно случайными. В таком виде программу оставлять нельзя, но к этому моменту никто не сообщит нам о данной ошибке, если мы не заметим ее сами (рис. 3.5).

```
$ g++ -std=c++17 main.cpp -o main
$ ./main
1
0
```

Рис. 3.5

7. Ситуацию спасут флаги отладки! Реализация GNU STL поддерживает макрос препроцессора `_GLIBCXX_DEBUG`, который активизирует много функций для проверки достоверности STL. Это замедляет выполнение программы, но зато помогает *находить ошибки*. Можно активизировать макрос, добавив флаг `-D_GLIBCXX_DEBUG` в командную строку компилятора или определив его в начале файла кода, поместив его до директив `include`. Как видите, он завершает работу приложения, после чего запускаются разные средства очистки. Скомпилируем код с флагом для активизации проверяемых итераторов (в компиляторе Microsoft Visual C++ выглядит как `/D_ITERATOR_DEBUG_LEVEL=1`) (рис. 3.6).

```
$ g++ -std=c++17 main.cpp -D_GLIBCXX_DEBUG -o main
$ ./main
1
/opt/gcc_latest/include/c++/7.0.0/debug/safe_iterator.h:270:
Error: attempt to dereference a singular iterator.

Objects involved in the operation:
    iterator "this" @ 0x0x7ffc06323730 {
      type = __gnu_debug::_Safe_iterator<__gnu_cxx::__normal_iterator<int*, std::__cxx1998::vecto
r<int, std::allocator<int> > >, std::__debug::vector<int, std::allocator<int> > > (mutable iterat
or);
      state = singular;
      references sequence with type 'std::__debug::vector<int, std::allocator<int> >' @ 0x0x7ffc0
6323760
    }
Aborted (core dumped)
```

Рис. 3.6

8. Реализация STL для LLVM/clang тоже имеет флаги отладки, но они нужны для отладки самой STL, а не пользовательского кода. Для последнего можно активизировать различные средства очистки. Скомпилируем код для clang с помощью флагов `-fsanitize=address -fsanitize=undefined` и посмотрим, что произойдет (рис. 3.7).

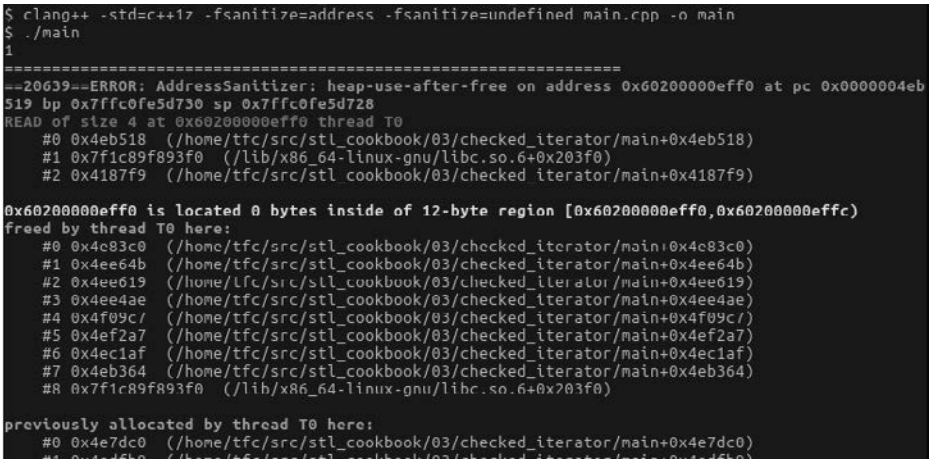

Рис. 3.7

Ого! Перед нами очень точное описание того, что именно пошло не так. Если бы мы не обрезали этот скриншот, то он занял бы несколько страниц книги. Обратите внимание: это характерно не только для clang, но и для GCC.

Если вы видите ошибки во время выполнения программы из-за того, что отсутствует какая-то библиотека, то значит, ваш компилятор не поставляется с библиотеками libasan и libubsan. Попробуйте установить их с помощью вашего менеджера пакетов или чего-то аналогичного.

Как это работает

Как видите, нам ничего не нужно менять в программе, чтобы включить эту функциональность для кода, генерирующего ошибки. Мы, по сути, получили ее бесплатно, просто добавив некоторые флаги компилятора в командную строку при компиляции программы.

Эта возможность реализуется *средствами очистки*. Обычно это дополнительный модуль компилятора и библиотека, работающая во время выполнения программы. При активизации средства очистки компилятор добавит *дополнительную информацию* и *код* в бинарный файл, который представляет собой нашу

программу. Во время выполнения библиотеки средства очистки, связанные с бинарным файлом программы, например, заменяют функции `malloc` и `free`, чтобы *проанализировать*, как программа работает с получаемой памятью.

Средства очистки помогают обнаруживать различные баги. Перечислю лишь несколько полезных примеров.

- **Выход за пределы диапазона:** ошибка случается, когда мы получаем доступ к элементу массива, вектора или аналогичного контейнера, лежащему за пределами корректной области памяти.
- **Использование после освобождения:** ошибка происходит, если мы обращаемся к памяти кучи после того, как она была освобождена (что мы и сделали в данном разделе).
- **Переполнение переменной типа `int`:** ошибка появляется, если целочисленная переменная переполняется в результате подсчета значений, которые в нее не помещаются. Для целых чисел со знаком это приведет к неопределенному поведению.
- **Выравнивание указателя:** в некоторых архитектурах невозможно обратиться к памяти, если блоки памяти выровнены неаккуратно.

Средства очистки могут обнаруживать и многие другие ошибки.

Вы не всегда можете активизировать все доступные средства очистки, поскольку это *замедляет* программу. Однако хорошим тоном является их полная активизация в *блочных* и *интеграционных* тестах.

Дополнительная информация

Существует множество средств очистки для разных категорий ошибок, и все они еще разрабатываются. Мы можем и должны информировать других пользователей о том, как они могут улучшить свои тесты. На домашних страницах проектов GCC и LLVM в разделе документации перечислены доступные средства очистки:

- https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html;
- http://clang.llvm.org/docs/index.html (найдите в содержании раздел Sanitizers (Средства очистки)).

Каждый программист должен знать о тестировании с использованием средств очистки и всегда уметь его выполнять. К сожалению, в пугающе большом количестве компаний этого не происходит, несмотря даже на то, что код с ошибками — самая главная точка входа для *вредоносного ПО* и *компьютерных вирусов*.

Заняв должность разработчика в новой для вас компании, сразу убедитесь, что в вашей команде используются все доступные средства очистки. Если это не так, то у вас есть уникальный шанс исправить важные и незаметные ошибки в свой первый рабочий день!

Создаем собственный адаптер для итераторов-упаковщиков

Работа с разными языками программирования требует использования различных стилей программирования. Это неудивительно, поскольку каждый язык программирования был разработан для решения конкретных задач.

Существует особый стиль программирования — *чистое функциональное программирование*. Он значительно отличается от императивного, к которому привыкли программисты, работающие на C и C++. Несмотря на то что этот стиль значительно отличается от других, во многих ситуациях он позволяет писать очень элегантный код.

Один из примеров проявления данной элегантности — реализация формул, например скалярного произведения. Если даны два математических вектора, то нахождение их скалярного произведения означает попарное умножение чисел на одинаковых позициях вектора, а затем суммирование этих умноженных значений. Скалярное произведение векторов `(a, b, c) * (d, e, f)` равно `(a * e + b * e + c * f)`. (Скорее всего, имеется в виду `a * d`. — *Примеч. пер.*). Конечно, это можно сделать и с помощью языков C и C++. Код выглядел бы следующим образом:

```
std::vector<double> a {1.0, 2.0, 3.0};
std::vector<double> b {4.0, 5.0, 6.0};

double sum {0};
for (size_t i {0}; i < a.size(); ++i) {
    sum += a[i] * b[i];
}
// sum = 32.0
```

Как же выглядит аналогичный код в языках, которые считаются более элегантными?

Haskell — чистый функциональный язык, на этом языке вычислить скалярное произведение двух векторов можно с помощью следующей волшебной строки (рис. 3.8).

```
Prelude> a = [1.0, 2.0, 3.0]
Prelude> b = [4.0, 5.0, 6.0]
Prelude> sum $ zipWith (*) a b
32.0
```

Рис. 3.8

Python не является чистым функциональным языком, но в некоторой степени использует аналогичные шаблоны, что видно в следующем примере (рис. 3.9).

```
>>> a = [1.0, 2.0, 3.0]
>>> b = [4.0, 5.0, 6.0]
>>> sum([ p[0] * p[1] for p in zip(a, b) ])
32.0
```

Рис. 3.9

В библиотеке STL вы можете найти специальный алгоритм: `std::inner_product`, тоже решающий эту конкретную задачу в одну строку. Но идея заключается в том, что во многих языках программирования такой код можно писать динамически одной строкой, не подключая конкретные функции библиотек, которые решают именно эту задачу.

Не погружаясь в объяснения синтаксиса других языков, выделим важную деталь, которая является общей в обоих примерах, — магическую функцию `zip`. Что она делает? Принимает два вектора `a` и `b` и преобразует их в смешанный вектор. Например, при вызове этой функции векторы `[a1, a2, a3]` и `[b1, b2, b3]` будут выглядеть как `[ (a1, b1), (a2, b2), (a3, b3) ]`. Посмотрите на него внимательно; он работает почти так же, как и ускорители упаковки!

Важное значение имеет тот факт, что теперь вы можете проитерировать по одному объединенному промежутку, выполнив попарное умножение и сложив результаты в переменную-аккумулятор. Именно это и происходит в примерах кода на языках Haskell и Python, где не используются ни циклы, ни ненужные индексные переменные.

Код на языке C++ нельзя сделать таким же элегантным, как код на языке Haskell или Python, но в этом разделе мы поговорим о способах реализации подобных возможностей с помощью итераторов путем добавления *итератора-упаковщика*. Определить скалярное произведение двух векторов можно более элегантно, задействуя конкретные библиотеки, но данный вопрос не относится к теме нашей книги. Однако я пытаюсь показать, насколько библиотеки, основанные на итераторах, могут помочь при написании выразительного кода, предоставляя очень обобщенные модули.

Как это делается

В этом примере мы воссоздадим функцию `zip`, известную из языков Haskell и Python. Она будет работать только для векторов, содержащих значения типа `double`, чтобы не отвлекаться от механики итераторов.

1. Сначала включим некоторые заголовочные файлы:

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

2. Далее определим класс `zip_iterator`. При переборе диапазонов данных `zip_iterator` мы будем получать на каждом этапе пару значений из двух контейнеров. Это значит, что мы итерируем по двум контейнерам одновременно:

```
class zip_iterator {
```

3. Итератор-упаковщик должен сохранять два итератора, по одному для каждого контейнера:

```
    using it_type = std::vector<double>::iterator;
    it_type it1;
    it_type it2;
```

4. Конструктор просто сохраняет итераторы обоих контейнеров, по которым нужно проитерировать:

```
public:
    zip_iterator(it_type iterator1, it_type iterator2)
        : it1{iterator1}, it2{iterator2}
    {}
```

5. Инкрементирование итератора-упаковщика означает инкрементирование обоих итераторов-членов:

```
    zip_iterator& operator++() {
        ++it1;
        ++it2;
        return *this;
    }
```

6. Два итератора-упаковщика считаются неравными, если оба их итератора-члена не равны своим коллегам из другого итератора-упаковщика. Обычно вы можете использовать логическое ИЛИ (`||`) вместо логического И (`&&`), но представьте, что диапазоны данных имеют неравную длину. В таких случаях нельзя соотнести *оба* конечных итератора одновременно. Таким образом, можно прервать выполнение цикла при достижении *первого* конечного итератора в *одном* из диапазонов данных:

```
    bool operator!=(const zip_iterator& o) const {
        return it1 != o.it1 && it2 != o.it2;
    }
```

7. Оператор сравнения равенства реализуется с помощью другого оператора, изменяя результат его работы на противоположный:

```
    bool operator==(const zip_iterator& o) const {
        return !operator!=(o);
    }
```

8. Разыменование итератора-упаковщика открывает доступ к обоим контейнерам в одной и той же позиции:

```
    std::pair<double, double> operator*() const {
        return {*it1, *it2};
    }
};
```

9. Мы рассмотрели код итератора. Нужно сделать итератор совместимым с алгоритмами STL, поэтому следует определить стереотипный код для типажа. По сути, он говорит, что данный итератор является обычным однонаправленным и при разыменовании возвращает пары значений типа `double`. Несмотря на то, что мы не использовали в текущем примере `difference_type`, для некоторых реализаций STL это может понадобиться для успешной компиляции кода:

```
namespace std {
template <>
```

```
struct iterator_traits<zip_iterator> {
   using iterator_category = std::forward_iterator_tag;
   using value_type = std::pair<double, double>;
   using difference_type = long int;
};
}
```

10. Следующий шаг — определение класса диапазона данных, функции `begin` и `end` которого возвращают итераторы-упаковщики:

```
class zipper {
    using vec_type = std::vector<double>;
    vec_type &vec1;
    vec_type &vec2;
```

11. Он должен сослаться на два существующих контейнера, чтобы создать итераторы-упаковщики:

```
public:
    zipper(vec_type &va, vec_type &vb)
        : vec1{va}, vec2{vb}
    {}
```

12. Функции `begin` и `end` просто передают пары начальных и конечных указателей, чтобы создать с их помощью экземпляры итераторов-упаковщиков:

```
    zip_iterator begin() const {
        return {std::begin(vec1), std::begin(vec2)};
    }
    zip_iterator end() const {
        return {std::end(vec1), std::end(vec2)};
    }
};
```

13. Как и в примерах кода на языках Haskell и Python, определяем два вектора, содержащих значения типа `double`. Кроме того, указываем, что используем пространство имен `std` внутри функции `main` по умолчанию:

```
int main()
{
    using namespace std;
    vector<double> a {1.0, 2.0, 3.0};
    vector<double> b {4.0, 5.0, 6.0};
```

14. Объект класса `zipper` объединяет их в один диапазон данных, напоминающий вектор, где можно увидеть пары значений векторов `a` и `b`:

```
    zipper zipped {a, b};
```

15. Используем метод `std::accumulate`, чтобы сложить все элементы диапазона данных. Сделать это напрямую нельзя, поскольку в результате мы сложим экземпляры типа `std::pair<double, double>`, для которых концепция суммирования не определена. Поэтому зададим вспомогательное лямбда-выражение, которое принимает пару, перемножает ее члены и складывает результат со зна-

чением переменной-аккумулятора. Функция `std::accumulate` хорошо работает с лямбда-выражениями со следующей сигнатурой:

```
const auto add_product ([](double sum, const auto &p) {
   return sum + p.first * p.second;
});
```

16. Теперь передадим его функции `std::accumulate`, а также пару итераторов для упаковываемых диапазонов и стартовое значение `0.0` для переменной-аккумулятора, которая, в конечном счете, будет содержать сумму произведений:

```
const auto dot_product (accumulate(
        begin(zipped), end(zipped), 0.0, add_product));
```

17. Выведем на экран полученный результат:

```
    cout << dot_product << '\n';
}
```

18. Компиляция и запуск программы дадут правильный результат:

```
32
```

Дополнительная информация

Да, нам пришлось написать много строк, чтобы добавить в код немного синтаксического сахара, и он все еще не так элегантен, как версия кода на Haskell. Большим недостатком является тот факт, что наш итератор-упаковщик жестко закодирован — он работает только для векторов, содержащих переменные типа `double`.

С помощью шаблонов и типажей типов такой итератор может стать более обобщенным. Таким образом, он сможет работать со списками, векторами, двусторонними очередями и ассоциативными массивами, даже если они специализированы для совершенно других типов элементов контейнера.

Нельзя недооценивать объем работы, которую нужно выполнить, чтобы сделать эти классы обобщенными. К счастью, такие библиотеки уже существуют. Одной из популярных библиотек, не входящих в STL, является *Boost* (`zip_iterator`). Ее легко использовать для самых разных классов.

Кстати, если хотите узнать о наиболее элегантном способе определения *скалярного произведения* в C++ и вам не особо нужна концепция итераторов-упаковщиков, то обратите внимание на `std::valarray`. Взгляните сами:

```
#include <iostream>
#include <valarray>

int main()
{
    std::valarray<double> a {1.0, 2.0, 3.0};
    std::valarray<double> b {4.0, 5.0, 6.0};
    std::cout << (a * b).sum() << '\n';
}
```

Библиотека ranges. Это очень интересная библиотека для языка C++, которая поддерживает упаковщики и все прочие виды волшебных адаптеров итераторов, фильтров и т. д. Ее создатели вдохновлялись библиотекой Boost ranges, и какое-то время казалось, что она может попасть в C++17, но, к сожалению, придется ждать следующего стандарта. Библиотека значительно улучшит возможности написания выразительного и быстрого кода на C++ путем создания сложных механизмов из универсальных и простых блоков кода. В документации к ней можно найти очень простые примеры.

1. Определение суммы квадратов всех чисел от `1` до `10`:

```
const int sum = accumulate(view::ints(1)
                         | view::transform([](int i){return i*i;})
                         | view::take(10), 0);
```

2. Фильтрация всех четных чисел из вектора чисел и преобразование остальных чисел в строки:

```
std::vector<int> v {1,2,3,4,5,6,7,8,9,10};
auto rng = v | view::remove_if([](int i){return i % 2 == 1;})
             | view::transform([](int i){return std::to_string(i);});
// rng == {"2"s,"4"s,"6"s,"8"s,"10"s};
```

Если вы заинтересовались и не можете дождаться выхода следующего стандарта C++, то обратитесь к документации для ranges, которая находится по адресу https://ericniebler.github.io/range-v3/.

4 Лямбда-выражения

В этой главе:

- динамическое определение функций с помощью лямбда-выражений;
- добавление полиморфизма путем оборачивания лямбда-выражений в конструкцию `std::function`;
- создание функций с помощью конкатенации;
- создание сложных предикатов с помощью логической конъюнкции;
- вызов нескольких функций с одними и теми же входными данными;
- реализация `transform_if` с применением `std::accumulate` и лямбда-выражений;
- генерация декартова произведения на основе любых входных данных во время компиляции.

Введение

Одной из важных новых функций C++11 были лямбда-выражения. В C++14 и C++17 они получили новые возможности, и это сделало их еще мощнее. Но что же такое лямбда-выражение?

Лямбда-выражения или лямбда-функции создают замыкания. Замыкание — очень обобщенный термин для *безымянных объектов*, которые можно *вызывать* как функции. Чтобы предоставить подобную возможность в C++, такой объект должен реализовывать оператор вызова функции `()`, с параметрами или без. Создание аналогичного объекта без лямбда-выражений до появления C++11 выглядело бы так:

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

int main() {
    struct name_greeter {
        std::string name;
        void operator()() {
            std::cout << "Hello, " << name << '\n';
        }
    };
```

```
    name_greeter greet_john_doe {"John Doe"};
    greet_john_doe();
}
```

Экземпляры структуры `name_greeter`, очевидно, содержат строку. Обратите внимание: тип этой структуры и объект не являются безымянными, в отличие от лямбда-выражений. С точки зрения замыканий можно утверждать, что они *захватывают* строку. Когда экземпляр-пример вызывается как функция без параметров, на экран выводится строка `"Hello, John Doe"`, поскольку мы указали строку с таким именем.

Начиная с C++11 создавать подобные замыкания стало проще:

```
#include <iostream>

int main() {
    auto greet_john_doe ([] {
        std::cout << "Hello, John Doe\n";
    });

    greet_john_doe();
}
```

На этом все. Целая структура `name_greeter` заменяется небольшой конструкцией `[] { /* сделать что-то */ }`, которая на первый взгляд выглядит странно, но уже в следующем разделе мы рассмотрим все возможные случаи ее применения.

Лямбда-выражения помогают поддерживать код *обобщенным* и *чистым*. Они могут применяться как параметры для обобщенных алгоритмов, чтобы уточнить их при обработке конкретных типов, определенных пользователем. Они также могут служить для оборачивания рабочих пакетов и данных, чтобы их можно было запускать в потоках или просто сохранять работу и откладывать само выполнение пакетов. После появления C++11 было создано множество библиотек, работающих с лямбда-выражениями, поскольку они стали естественной частью языка C++. Еще одним вариантом использования лямбда-выражений является метапрограммирование, поскольку они могут быть оценены во время выполнения программы. Однако мы не будем рассматривать этот вопрос, так как он не относится к теме данной книги.

В текущей главе мы в значительной мере будем опираться на отдельные шаблоны *функционального программирования*, которые могут показаться странными для новичков и даже опытных программистов, еще не работавших с такими шаблонами. Если в последующих примерах вы увидите лямбда-выражения, возвращающие лямбда-выражения, которые опять-таки возвращают лямбда-выражения, то, пожалуйста, не теряйтесь. Мы несколько выходим за рамки привычного стиля программирования, чтобы подготовиться к работе с современным языком C++, в котором шаблоны функционального программирования встречаются все чаще и чаще. Если код какого-то примера выглядит слишком сложным, то уделите время тому, чтобы подробнее разобрать его. Как только вы с этим разберетесь, сложные лямбда-выражения в реальных проектах больше не будут вас смущать.

Динамическое определение функций с помощью лямбда-выражений

Применяя лямбда-выражения, можно инкапсулировать код, чтобы вызвать его позже или даже в другом месте, поскольку его разрешено копировать. Кроме того, можно инкапсулировать код несколько раз с несколько различающимися параметрами, при этом не нужно будет реализовывать новый класс функции для этой задачи.

Синтаксис лямбда-выражений выглядел новым в C++11, и к C++17 он несколько изменился. В этом разделе мы увидим, как сейчас выглядят лямбда-выражения и что они означают.

Как это делается

В этом примере мы напишем небольшую программу, в которой поработаем с лямбда-выражениями, чтобы понять основные принципы взаимодействия с ними.

1. Для работы с лямбда-выражениями не нужна поддержка библиотек, но мы будем выводить сообщения на консоль и использовать строки, поэтому понадобятся соответствующие заголовочные файлы:

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

2. В данном примере все действие происходит в функции `main`. Мы определим два объекта функций, которые не принимают параметры, и вернем целочисленные константы со значениями `1` и `2`. Обратите внимание: выражение `return` окружено фигурными скобками `{}`, как это делается в обычных функциях, а круглые скобки `()`, указывающие на функцию без параметров, являются *необязательными*, мы не указываем их во втором лямбда-выражении. Но квадратные скобки `[]` должны присутствовать:

```
int main()
{
    auto just_one ( [](){ return 1; } );
    auto just_two ( []  { return 2; } );
```

3. Теперь можно вызвать оба объекта функций, просто написав имя переменных, которые в них сохранены, и добавив скобки. В этой строке их не отличить от *обычных функций*:

```
    std::cout << just_one() << ", " << just_two() << '\n';
```

4. Забудем о них и определим еще один объект функции, который называется `plus`, — он принимает два параметра и возвращает их сумму:

```
    auto plus ( [](auto l, auto r) { return l + r; } );
```

5. Использовать такой объект довольно просто, в этом плане он похож на любую другую бинарную функцию. Мы указали, что его параметры имеют тип `auto`, вследствие чего объект будет работать со всеми типами данных, для которых определен оператор `+`, например со строками.

```
std::cout << plus(1, 2) << '\n';
std::cout << plus(std::string{"a"}, "b") << '\n';
```

6. Не нужно сохранять лямбда-выражение в переменной, чтобы использовать его. Мы также можем определить его *в том месте*, где это необходимо, а затем разместить параметры для данного выражения в круглых скобках сразу после него `(1, 2)`:

```
std::cout
  << [](auto l, auto r){ return l + r; }(1, 2)
  << '\n';
```

7. Далее определим замыкание, которое содержит целочисленный счетчик. При каждом вызове значение этого счетчика будет увеличиваться на 1 и возвращать новое значение. Для указания на то, что замыкание содержит внутренний счетчик, разместим в скобках выражение `count = 0` — оно указывает, что переменная `count` инициализирована целочисленным значением `0`. Чтобы позволить ему изменять собственные переменные, мы используем ключевое слово `mutable`, поскольку в противном случае компилятор не разрешит это сделать:

```
auto counter (
    [count = 0] () mutable { return ++count; }
);
```

8. Теперь вызовем объект функции пять раз и выведем возвращаемые им значения с целью увидеть, что значение счетчика увеличивается:

```
for (size_t i {0}; i < 5; ++i) {
    std::cout << counter() << ", ";
}
std::cout << '\n';
```

9. Мы также можем взять существующие переменные и захватить их *по ссылке* вместо того, чтобы создавать копию значения для замыкания. Таким образом, значение переменной станет увеличиваться в замыкании и при этом будет доступно за его пределами. Для этого мы поместим в скобках конструкцию `&a`, где символ `&` означает, что мы сохраняем *ссылку* на переменную, но не *копию*:

```
int a {0};
auto incrementer ( [&a] { ++a; } );
```

10. Если это работает, то можно вызвать данный объект функции несколько раз, а затем пронаблюдать, действительно ли меняется значение переменной `a`:

```
incrementer();
incrementer();
incrementer();

std::cout
```

```
        << "Value of 'a' after 3 incrementer() calls: "
        << a << '\n';
```

11. Последний пример демонстрирует *каррирование*. Оно означает, что мы берем функцию, принимающую некоторые параметры, а затем сохраняем ее в другом объекте функции, принимающем *меньше* параметров. В этом случае мы сохраняем функцию `plus` и принимаем только *один* параметр, который будет передан в функцию `plus`. Другой параметр имеет значение `10`; его мы сохраняем в объекте функции. Таким образом, мы получаем функцию и назовем ее `plus_ten`, поскольку она может добавить значение `10` к единственному принимаемому ею параметру.

```
    auto plus_ten ( [=] (int x) { return plus(10, x); } );
    std::cout << plus_ten(5) << '\n';
}
```

12. Перед компиляцией и запуском программы пройдем по коду еще раз и попробуем предугадать, какие именно значения выведем в терминале. Затем запустим программу и взглянем на реальные выходные данные:

```
1, 2
3
ab
3
1, 2, 3, 4, 5,
Value of a after 3 incrementer() calls: 3
15
```

Как это работает

То, что мы сейчас сделали, выглядит не слишком сложно: сложили числа, а затем инкрементировали их и вывели на экран. Мы даже выполнили конкатенацию строк с помощью объекта функций, который был реализован для сложения чисел. Но для тех, кто еще незнаком с синтаксисом лямбда-выражений, это может показаться запутанным.

Итак, сначала рассмотрим все особенности, связанные с лямбда-выражениями (рис. 4.1).

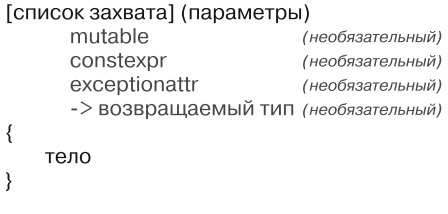

Рис. 4.1

Как правило, можно опустить большую часть этих параметров, чтобы сэкономить немного времени. Самым коротким лямбда-выражением является выражение `[]{}`.

Оно не принимает никаких параметров, ничего не захватывает и, по сути, *ничего* не делает.

Что же значит остальная часть?

Список для захвата

Определяет, что именно мы захватываем и выполняем ли захват вообще. Есть несколько способов сделать это. Рассмотрим два «ленивых» варианта.

1. Если мы напишем `[=] () {...}`, то захватим каждую внешнюю переменную, на которую ссылается замыкание, по значению; то есть эти значения будут *скопированы*.
2. Запись `[&] () {...}` означает следующее: все внешние объекты, на которые ссылается замыкание, захватываются только *по ссылке*, что не приводит к копированию.

Конечно, можно установить настройки захвата для каждой переменной отдельно. Запись `[a, &b] () {...}` означает, что переменную `a` мы захватываем *по значению*, а переменную `b` — *по ссылке*. Для этого потребуется напечатать больше текста, но, как правило, данный способ безопаснее, поскольку мы не можем случайно захватить что-то ненужное из-за пределов замыкания.

В текущем примере мы определили лямбда-выражение следующим образом: `[count=0] () {...}`. В этом особом случае мы не захватываем никаких переменных из-за пределов замыкания, только определили новую переменную с именем `count`. Тип данной переменной определяется на основе значения, которым мы ее инициализировали, а именно `0`, так что она имеет тип `int`.

Кроме того, можно захватить одни переменные по значению, а другие — по ссылке, например:

- `[a, &b] () {...}` — копируем `a` и берем ссылку на `b`;
- `[&, a] () {...}` — копируем `a` и применяем ссылку на любую другую переданную переменную;
- `[=, &b, i{22}, this] () {...}` — получаем ссылку на `b`, копируем значение `this`, инициализируем новую переменную `i` значением `22` и копируем любую другую использованную переменную.

Если вы попытаетесь захватить переменную-член некоторого объекта, то не сможете сделать это с помощью конструкции [member_a] () {...}. Вместо этого нужно определить либо this, либо *this.

mutable (необязательный)

Если объект функции должен иметь возможность *модифицировать* получаемые им переменные путем *копирования* (`[=]`), то его следует определить как `mutable`. Это же касается вызова неконстантных методов захваченных объектов.

constexpr (необязательный)

Если мы явно пометим лямбда-выражение с помощью ключевого слова `constexpr`, то компилятор сгенерирует *ошибку*, когда это выражение не будет соответствовать критериям функции `constexpr`. Преимущество использования функций `constexpr` и лямбда-выражений заключается в том, что компилятор может оценить их результат во время компиляции, если они вызываются с параметрами, постоянными на протяжении данного процесса. Это приведет к тому, что позднее в бинарном файле будет меньше кода.

Если мы не указываем явно, что лямбда-выражения являются `constexpr`, но эти выражения соответствуют всем требуемым критериям, то они все равно будут считаться `constexpr`, только *неявно*. Если *нужно*, чтобы лямбда-выражение было `constexpr`, то лучше явно задавать его таковым, поскольку иначе в случае наших *неверных* действий компилятор начнет генерировать ошибки.

exception attr (необязательный)

Здесь определяется, может ли объект функции генерировать исключения, если при вызове столкнется с ошибкой.

return type (необязательный)

При необходимости иметь полный контроль над возвращаемым типом, вероятно, не нужно, чтобы компилятор определял его автоматически. В таких случаях можно просто использовать конструкцию `[] () -> Foo {}`, которая укажет компилятору, что мы всегда будем возвращать объекты типа `Foo`.

Добавляем полиморфизм путем оборачивания лямбда-выражений в std::function

Предположим, нужно написать функцию-наблюдатель для какого-то значения, которое может изменяться время от времени, что приведет к оповещению других объектов, например индикатора давления газа, цены на акцию т. п. При изменении значения должен вызываться список объектов-наблюдателей, которые затем по-своему на это отреагируют.

Для реализации задачи можно поместить несколько объектов функции-наблюдателя в вектор, все они будут принимать в качестве параметра переменную типа `int`, которая представляет наблюдаемое значение. Мы не знаем, что именно станут делать данные функции при вызове, но нам это и неинтересно.

Какой тип будут иметь объекты функций, помещенные в вектор? Нам подойдет тип `std::vector<void (*)(int)>`, если мы захватываем указатели на *функции*, имеющие сигнатуры наподобие `void f(int);`. Данный тип сработает с любым лямбда-выражением, которое захватывает нечто, имеющее *совершенно другой тип* в сравнении с обычной функцией, поскольку это не просто указатель на функцию, а *объект*, объединяющий некий объем данных с функцией! Подумайте о временах

до появления C++11, когда лямбда-выражений не существовало. Классы и структуры были естественным способом связывания данных с функциями, и при изменении типов членов класса получится совершенно другой класс. Это *естественно*, что вектор не может хранить значения разных типов, используя одно имя типа.

Не стоит указывать пользователю, что он может сохранить объекты функции наблюдателя, которые ничего не захватывают, поскольку это ограничивает варианты применения. Как же позволить ему сохранять любые объекты функций, ограничивая лишь интерфейс вызова, принимающий конкретный диапазон параметров в виде наблюдаемых значений?

В этом разделе мы рассмотрим способ решения данной проблемы с помощью объекта `std::function`, который может выступать в роли полиморфической оболочки для любого лямбда-выражения, независимо от того, какие значения оно захватывает.

Как это делается

В этом примере мы создадим несколько лямбда-выражений, значительно отличающихся друг от друга, но имеющих одинаковую сигнатуру вызова. Затем сохраним их в одном векторе с помощью `std::function`.

1. Сначала включим необходимые заголовочные файлы:

```
#include <iostream>
#include <deque>
#include <list>
#include <vector>
#include <functional>
```

2. Реализуем небольшую функцию, которая возвращает лямбда-выражение. Она принимает контейнер и возвращает объект функции, захватывающий этот контейнер по ссылке. Сам по себе объект функции принимает целочисленный параметр. Когда данный объект получает целое число, он *добавит* его в свой контейнер.

```
static auto consumer (auto &container){
    return [&] (auto value) {
        container.push_back(value);
    };
}
```

3. Еще одна небольшая вспомогательная функция выведет на экран содержимое экземпляра контейнера, который мы предоставим в качестве параметра:

```
static void print (const auto &c)
{
    for (auto i : c) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}
```

4. В функции `main` мы создадим объекты классов `deque`, `list` и `vector`, каждый из которых будет хранить целые числа:

```
int main()
{
    std::deque<int>  d;
    std::list<int>   l;
    std::vector<int> v;
```

5. Сейчас воспользуемся функцией `consumer` для работы с нашими экземплярами контейнеров `d`, `l` и `v`: создадим для них объекты-потребители функций и поместим их в экземпляр `vector`. Эти объекты функций будут захватывать ссылку на один из объектов контейнера. Последние имеют разные типы, как и объекты функций. Тем не менее вектор хранит экземпляры типа `std::function<void(int)>`. Все объекты функций неявно оборачиваются в объекты типа `std::function`, которые затем сохраняются в векторе:

```
    const std::vector<std::function<void(int)>> consumers
        {consumer(d), consumer(l), consumer(v)};
```

6. Теперь поместим десять целочисленных значений во все структуры данных, проходя по значениям в цикле, а затем пройдем в цикле по объектам функций-потребителей, которые вызовем с записанными значениями:

```
    for (size_t i {0}; i < 10; ++i) {
        for (auto &&consume : consumers) {
            consume(i);
        }
    }
```

7. Все три контейнера теперь должны содержать одинаковые десять чисел. Выведем на экран их содержимое:

```
    print(d);
    print(l);
    print(v);
}
```

8. Компиляция и запуск программы дадут следующий результат, который выглядит именно так, как мы и ожидали:

```
$ ./std_function
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Как это работает

Самой сложной частью этого примера является следующая строка:

```
const std::vector<std::function<void(int)>> consumers
       {consumer(d), consumer(l), consumer(v)};
```

Объекты `d`, `l` и `v` обернуты в вызов `consumer(...)`. Он возвращает объекты функций, каждый из которых захватывает ссылки на один из объектов — `d`, `l` или `v`. Хотя все объекты функций принимают в качестве параметров целочисленные значения, тот факт, что они захватывают абсолютно *разные* переменные, также делает их *типы* совершенно разными. Это похоже на попытку разместить в векторе переменные типов `A`, `B` и `C`, когда сами типы не имеют *ничего* общего.

Чтобы это исправить, нужно найти *общий* тип, способный хранить *разные* объекты функций, например `std::function`. Объект типа `std::function<void(int)>` может хранить любой объект функции или традиционную функцию, которая принимает целочисленный параметр и ничего не возвращает. С помощью полиморфизма он отвязывает тип от лежащего в его основе типа объекта функции.

Представьте, будто мы написали такой код:

```
std::function<void(int)> f (
    [&vector](int x) { vector.push_back(x); });
```

Здесь объект функции, создаваемый на основе лямбда-выражения, обернут в объект типа `std::function`, и всякий раз вызов `f(123)` приводит к *виртуальному вызову функции*, который перенаправляется реальному объекту функции, находящемуся внутри.

При сохранении объектов функции экземпляры `std::function` применяют некоторую логику. При захвате все большего и большего количества переменных лямбда-выражение будет увеличиваться. Если его размер относительно мал, то `std::function` может хранить его внутри себя. Если же размер сохраненного объекта функций слишком велик, то `std::function` выделит фрагмент памяти в куче и сохранит объект там. Возможности нашего кода в подобном случае затронуты не будут, но знать об этом нужно, поскольку такая ситуация может повлиять на его *производительность*.

Многие программисты-новички думают или надеются, что std::function<...> на самом деле выражает тип лямбда-выражения. Отнюдь. Это полиморфическая вспомогательная библиотечная функция, которая полезна для оборачивания лямбда-выражений и сокрытия различий в их типах.

Создаем функции методом конкатенации

Многие проблемы можно решить, не полагаясь исключительно на собственный код. Например, взглянем на то, как решается задача поиска уникальных слов в тексте на языке программирования Haskell. В первой строке определяется функция `unique_words`, а во второй показывается ее использование на примере строки (рис. 4.2).

```
Prelude> unique_words = length . group . sort . words . (map toLower)
Prelude> unique_words "A B c d a b c d e"
5
```

Рис. 4.2

Ого! Программа получилась действительно короткой. Не вдаваясь особо в синтаксис языка Haskell, взглянем на то, что делает код. Определяется функция `unique_words`, в которой к входным данным применяется набор функций. Сначала все символы преобразуются в строчные с помощью `map toLower`. Таким образом, слова наподобие FOO и foo могут считаться *одним* словом. Далее функция `words` разбивает предложение на отдельные слова. Например, из строки `"foo bar baz"` мы получим массив `["foo", "bar", "baz"]`. Следующий шаг — сортировка нового списка слов. В результате последовательность слов `["a", "b", "a"]` будет выглядеть как `["a", "a", "b"]`. Теперь в дело вступает функция `group`. Она группирует последовательные слова в списки, то есть конструкция `["a", "a", "b"]` получит вид `[["a", "a"], ["b"] ]`. Задача практически выполнена, и теперь нужно сосчитать, сколько получилось групп одинаковых слов. В этом поможет функция `length`.

Это замечательный стиль программирования, так как мы можем прочитать код справа налево, поскольку, по сути, описываем процесс преобразования предложения. Нет нужды знать, как реализованы отдельные фрагменты (если только они не будут работать слишком медленно и с ошибками).

Однако мы здесь не ради восхваления Haskell, а чтобы улучшить навыки работы с языком C++. Он позволяет написать аналогичный код. Мы не сможем достичь того же уровня элегантности, какой видели в языке Haskell, зато у нас под рукой самый быстрый язык программирования. В этом разделе показано, как инициировать *конкатенацию функций* в C++ с помощью лямбда-выражений.

Как это делается

В данном примере мы определим несколько простых объектов функций и *сконкатенируем* их, чтобы получить одну функцию, которая применяет такие же функции одну за другой для полученных входных данных. Для этого напишем собственную вспомогательную функцию для конкатенации.

1. Сначала включим несколько заголовочных файлов:

```
#include <iostream>
#include <functional>
```

2. Далее реализуем вспомогательную функцию `concat`, которая принимает множество параметров. Таковыми выступят функции наподобие `f`, `g` и `h`, а результатом

будет еще один объект функции, применяющий функции `f(g(h(...)))` для любых входных данных.

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts)
{
```

3. Теперь задача усложняется. Когда пользователь предоставит функции `f`, `g` и `h`, мы оценим это выражение как `f( concat(g, h) )`, которое будет распаковано в `f( g( concat(h) ) )`, на чем рекурсия остановится, и мы получим выражение `f( g( h(...) ) )`. Данная цепочка вызовов функций, представляющая конкатенацию пользовательских функций, захватывается лямбда-выражением, которое затем может принять какие-то параметры `p` и передать в вызов `f(g(h(p)))`. Мы будем возвращать это лямбда-выражение. Конструкция `if constexpr` проверяет, находимся ли мы на шаге рекурсии, требующем сконкатенировать более чем одну функцию:

```
    if constexpr (sizeof...(ts) > 0) {
        return [=](auto ...parameters) {
            return t(concat(ts...)(parameters...));
        };
    }
```

4. Еще одна ветвь конструкции `if constexpr` будет выбрана компилятором в том случае, если достигнут *конец* рекурсии. В таких ситуациях просто возвращаем функцию, `t`, поскольку она является единственным оставшимся параметром:

```
    else {
        return t;
    }
}
```

5. Теперь применим нашу новую функцию конкатенации, передав в нее несколько функций. Начнем с функции `main`, где определим два дешевых объекта функций:

```
int main()
{
    auto twice  ([] (int i) { return i * 2; });
    auto thrice ([] (int i) { return i * 3; });
```

6. Выполним конкатенацию. Объединим два объекта функций умножения с помощью функции STL `std::plus<int>`, которая принимает два параметра и возвращает их сумму. Таким образом, получим функцию, выполняющую вызов `twice(thrice(plus(a, b )))`.

```
    auto combined (
        concat(twice, thrice, std::plus<int>{})
    );
```

7. Воспользуемся тем, что получилось. Функция `combined` теперь выглядит как обычная, и компилятор может объединять эти функции без особых задержек:

```
    std::cout << combined(2, 3) << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат, и он не будет неожиданным, поскольку `2 * 3 * (2 + 3)` равно `30`:

```
$ ./concatenation
30
```

Как это работает

Самой сложной частью этого раздела является функция `concat`. Она выглядит очень мудреной, поскольку разворачивает набор параметров `ts` в другое лямбда-выражение, которое рекурсивно снова вызывает функцию `concat`, теперь уже с меньшим количеством параметров:

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts)
{
    if constexpr (sizeof...(ts) > 0) {
        return [=](auto ...parameters) {
            return t(concat(ts...)(parameters...));
        };
    } else {
        return [=](auto ...parameters) {
            return t(parameters...);
        };
    }
}
```

Напишем более простую версию этой функции, которая объединяет ровно три функции:

```
template <typename F, typename G, typename H>
auto concat(F f, G g, H h)
{
    return [=](auto ... params) {
        return f( g( h( params... ) ) );
    };
}
```

Эта функция выглядит аналогично, но уже не так сложна. Мы возвращаем лямбда-выражение, которое захватывает `f`, `g` и `h`. Оно принимает произвольно большое количество параметров и просто перенаправляет их по цепочке вызовов `f`, `g` и `h`. Если мы пользуемся конструкцией `auto combined (concat(f, g, h))`, а затем вызываем данный объект функции с двумя параметрами, например `combined(2, 3)`, то `2, 3` представлены набором параметров из предыдущей функции `concat`.

Повторный взгляд на гораздо более сложную обобщенную функцию `concat` позволяет увидеть следующее: единственное, что мы действительно делаем по-другому, — выполняем конкатенацию `f ( g( h( params... ) ) )`. Вместо этого мы пишем выражение `f(concat(g, h) )(params...)`, которое будет преобразовано в конструкцию `f( g( concat(h) ) )(params...)` при следующем рекурсивном вызове, а затем — в конструкцию `f( g( h( params... ) ) )`.

Создаем сложные предикаты с помощью логической конъюнкции

При фильтрации данных с помощью обобщенного кода мы определяем *предикаты*, которые указывают, какие именно данные нужны. Иногда предикаты являются *комбинациями* нескольких «собратьев».

При фильтрации строк, например, можно реализовать предикат, который возвращает значение `true`, если входная строка *начинается* со слова `"foo"`. Еще один предикат должен возвращать значение `true`, если входная строка *заканчивается* словом `"bar"`.

Вместо того чтобы постоянно писать собственные предикаты, можно *повторно использовать* предикаты, объединив их. Для фильтрации строк, которые начинаются со слова `"foo"` и заканчиваются словом `"bar"`, можно просто выбрать уже *существующие* предикаты и *объединить* их с помощью логического И. В данном разделе мы будем работать с лямбда-выражениями, чтобы найти удобный способ сделать это.

Как это делается

В этом примере мы реализуем очень простые предикаты для фильтрации строк, а затем объединим их с помощью небольшой вспомогательной функции, которая создаст их комбинацию в обобщенном виде.

1. Как обычно, сначала включим несколько заголовочных файлов:

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

2. Поскольку они понадобятся нам в дальнейшем, реализуем две простые функции-предиката. Одна из них говорит о том, начинается ли строка с символа `'a'`, а вторая — заканчивается ли строка символом `'b'`:

```
static bool begins_with_a (const std::string &s)
{
    return s.find("a") == 0;
}

static bool ends_with_b (const std::string &s)
{
    return s.rfind("b") == s.length() - 1;
}
```

3. Теперь реализуем вспомогательную функцию и назовем ее `combine`. Она принимает бинарную функцию в качестве первого параметра — это может быть,

например, логическое И либо логическое ИЛИ. Затем она принимает два других параметра, представляющих собой две функции-предиката, которые нужно объединить:

```
template <typename A, typename B, typename F>
auto combine(F binary_func, A a, B b)
{
```

4. Просто возвращаем лямбда-выражение, которое захватывает новую *комбинацию* предикатов. Оно направляет параметр обоим предикатам, а затем помещает результаты работы их обоих в бинарную функцию и возвращает ее результат:

```
    return [=](auto param) {
        return binary_func(a(param), b(param));
    };
}
```

5. Укажем, что будем использовать пространство имен `std` с целью сэкономить немного времени при написании функции `main`:

```
using namespace std;
```

6. Теперь объединим две функции-предиката в другую функцию-предикат, говорящую, начинается ли заданная строка с символа `a` *и* заканчивается ли символом `b`, как, например, строки `"ab"` или `"axxxb"`. На роль бинарной функции выбираем `std::logical_and`. Это шаблонный класс, экземпляр которого нужно создавать, вследствие чего будем использовать его вместе с фигурными скобками. Обратите внимание: мы не предоставляем дополнительный параметр шаблона, поскольку для данного класса он по умолчанию будет равен `void`. Эта специализация класса самостоятельно выводит все типы параметров:

```
int main()
{
    auto a_xxx_b (combine(
        logical_and<>{},
        begins_with_a, ends_with_b));
```

7. Итерируем по стандартному потоку ввода и выводим на экран все слова, которые удовлетворяют условиям предиката:

```
    copy_if(istream_iterator<string>{cin}, {},
            ostream_iterator<string>{cout, ", "},
            a_xxx_b);
    cout << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат. Мы передаем программе четыре слова, но только два из них удовлетворяют условиям предиката:

```
$ echo "ac cb ab axxxb" | ./combine
ab, axxxb,
```

Дополнительная информация

Библиотека STL уже предоставляет несколько функциональных объектов наподобие `std::logical_and`, `std::logical_or`, а также множество других, поэтому не нужно реализовывать их в каждом проекте. Ознакомиться со справочным материалом по C++ и исследовать доступные варианты можно на http://en.cppreference.com/w/cpp/utility/functional.

Вызываем несколько функций с одинаковыми входными данными

Существует множество задач, чьи решения требуют написания повторяющегося кода. Большую его часть можно легко заменить лямбда-выражениями, и очень просто создать вспомогательную функцию, которая будет оборачивать подобные повторяющиеся задачи.

В данном разделе мы поработаем с лямбда-выражениями, чтобы направить один вызов функции со всеми его параметрами нескольким получателям. Мы сделаем это, не задействовав никаких дополнительных структур данных, так что перед компилятором будет стоять простая задача: сгенерировать бинарный файл без лишних затрат.

Как это делается

В этом примере мы напишем вспомогательную функцию для работы с лямбда-выражениями, которая направляет один вызов нескольким объектам, и еще одну такую функцию, направляющую один вызов нескольким функциям. Эту комбинацию мы используем в нашем примере, чтобы вывести на экран одно сообщение с помощью разных функций-принтеров.

1. Включим заголовочный файл, необходимый для вывода данных на экран:

   ```
   #include <iostream>
   ```

2. Сначала реализуем функцию `multicall`, которая является основной для этого примера. Она принимает произвольное количество функций в качестве параметров и возвращает лямбда-выражение, принимающее один параметр. Она перенаправляет данный параметр всем функциям, предоставленным ранее. Таким образом, можно определить функцию `auto call_all (multicall(f, g, h))`, а затем вызов `call_all(123)` приведет к серии вызовов `f(123); g(123); h(123);`. Эта функция уже выглядит сложной, поскольку требуется распаковать набор параметров, в котором находятся функции, в набор вызовов с помощью конструктора `std::initializer_list`.

   ```
   static auto multicall (auto ...functions)
   {
   ```

```
    return [=](auto x) {
        (void)std::initializer_list<int>{
            ((void)functions(x), 0)...
        };
    };
}
```

3. Следующая вспомогательная функция принимает функцию `f` и набор параметров `xs`. После этого она вызывает функцию `f` для каждого из параметров. Таким образом, вызов `for_each(f, 1, 2, 3)` приводит к серии вызовов: `f(1); f(2); f(3);`. Функция, по сути, использует такой же синтаксический прием для распаковки набора параметров `xs` в набор вызовов функций, как и функция, показанная ранее.

```
static auto for_each (auto f, auto ...xs) {
    (void)std::initializer_list<int>{
        ((void)f(xs), 0)...
    };
}
```

4. Функция `brace_print` принимает два символа и возвращает новый объект функции, принимающий один параметр `x`. Она выводит его на экран, окружив двумя символами, которые мы только что захватили.

```
static auto brace_print (char a, char b) {
    return [=] (auto x) {
        std::cout << a << x << b << ", ";
    };
}
```

5. Теперь наконец можно использовать все эти функции в функции `main`. Сначала определим функции `f`, `g` и `h`. Они представляют функции `print`, которые принимают значения и выводят их на экран, окружив разными скобками. Функция `nl` принимает любой параметр и просто выводит на экран символ переноса строки.

```
int main()
{
    auto f  (brace_print('(', ')'));
    auto g  (brace_print('[', ']'));
    auto h  (brace_print('{', '}'));
    auto nl ([](auto) { std::cout << '\n'; });
```

6. Объединим все эти функции с помощью вспомогательной функции `multicall`:

```
    auto call_fgh (multicall(f, g, h, nl));
```

7. Мы хотим, чтобы каждое предоставленное нами число было выведено на экран трижды в разных скобках. Таким образом, нужно выполнить один вызов функции, который приведет к пяти вызовам нашей мультифункции, а та, в свою очередь, выполнит четыре вызова для функций `f`, `g`, `h` и `nl`:

```
    for_each(call_fgh, 1, 2, 3, 4, 5);
}
```

8. Перед компиляцией и запуском программы подумаем, какой результат должны получить:

```
$ ./multicaller
(1), [1], {1},
(2), [2], {2},
(3), [3], {3},
(4), [4], {4},
(5), [5], {5},
```

Как это работает

Вспомогательные функции, которые мы только что реализовали, выглядят очень сложными. Так произошло из-за распаковки набора параметров с помощью `std::initializer_list`. Почему мы вообще использовали эту структуру данных? Еще раз взглянем на `for_each`:

```
auto for_each ([](auto f, auto ...xs) {
    (void)std::initializer_list<int>{
        ((void)f(xs), 0)...
    };
});
```

Сердцем данной функции является выражение `f(xs)`. `xs` — набор параметров, и нужно *распаковать* его, чтобы получить отдельные значения и передать их отдельным вызовам функции `f`. К сожалению, мы не можем просто написать конструкцию `f(xs)...` с помощью нотации `...`, с которой уже знакомы.

Вместо этого можно создать список значений с помощью `std::initializer_list`, имеющего конструктор с переменным числом параметров. Выражение наподобие `return std::initializer_list<int>{f(xs)...};` решает задачу, но имеет *недостатки*. Взглянем на реализацию функции `for_each`, которая тоже работает и при этом выглядит проще нашего варианта:

```
auto for_each ([](auto f, auto ...xs) {
    return std::initializer_list<int>{f(xs)...};
});
```

Она более проста для понимания, но имеет следующие недостатки.

1. Создает список инициализаторов для возвращаемых значений на основе вызовов функции `f`. К этому моменту нас не волнуют возвращаемые значения.
2. *Возвращает* данный список, а нам нужна функция, работающая в стиле *«запустил и забыл»*, которая не возвращает *ничего*.
3. Вполне возможно, что `f` — функция, которая не возвращает ничего, в таком случае код даже не будет скомпилирован.

Гораздо более сложная функция `for_each` решает все эти проблемы. Она делает следующее.

1. *Не возвращает* список инициализаторов, а приводит все выражение к типу `void` с помощью `(void)std::initializer_list<int>{...}`.

2. Внутри инициализирующего выражения преобразует выражение `f(xs)...` в выражение `(f(xs), 0)...`. Это приводит к тому, что возвращаемое выражение *отбрасывается*, а значение `0` все еще помещается в список инициализаторов.
3. Конструкция `f(xs)` в выражении `(f(xs), 0)...` также преобразуется к типу `void`, поэтому возвращаемое значение, если таковое существует, нигде не обрабатывается.

Объединение этих особенностей, к сожалению, ведет к появлению уродливой конструкции, но она корректно работает и компилируется для множества объектов функций независимо от того, возвращают ли они какое-то значение.

Приятной особенностью описанного механизма является тот факт, что порядок вызовов функций будет сохраняться в *строгой последовательности*.

Выполнять преобразование конструкции (void)*выражение* в рамках старой нотации языка C не рекомендуется, поскольку в языке C++ имеются собственные операции преобразования. Вместо этого стоит использовать конструкцию reinterpret_cast<void>(выражение), но данный вариант еще больше снизит удобочитаемость кода.

Реализуем функцию transform_if с применением std::accumulate и лямбда-выражений

Большинство разработчиков, применяющих `std::copy_if` и `std::transform`, могли задаваться вопросом, почему не существует функции `std::transform_if`. Функция `std::copy_if` копирует элементы из исходного диапазона по месту назначения, но опускает элементы, не соответствующие определенной пользователем функции-*предикату*. Функция `std::transform` безусловно копирует все элементы из исходного диапазона по месту назначения, но при этом преобразует их в процессе. Это происходит с помощью функции, которая определена пользователем и может выполнять как нечто простое (например, умножение чисел), так и полные преобразования к другим типам.

Эти функции существуют достаточно давно, но функции `std::transform_if` *все еще* нет. Ее можно легко создать, реализовав функцию, которая итерирует по диапазонам данных и копирует все элементы, соответствующие предикату, выполняя в процессе их преобразование. Однако мы воспользуемся случаем и разберем решение данной задачи с точки зрения лямбда-выражений.

Как это делается

В этом примере мы создадим собственную функцию `transform_if`, которая работает, передавая алгоритму `std::accumulate` правильные объекты функций.

1. Как и всегда, включим некоторые заголовочные файлы:

```
#include <iostream>
#include <iterator>
#include <numeric>
```

2. Сначала реализуем функцию с именем `map`. Она принимает функцию преобразования входных данных и возвращает объект функции, который будет работать с функцией `std::accumulate`:

```
template <typename T>
auto map(T fn)
{
```

3. Мы будем возвращать объект функции, принимающий функцию `reduce`. Когда данный объект вызывается с этой функцией, он возвращает другой объект функции, который принимает аккумулятор и входной параметр. Он вызывает функцию `reduce` для этого аккумулятора и преобразованной входной переменной `fn`. Если это описание кажется вам слишком сложным — не волнуйтесь, далее мы соберем все вместе и посмотрим, как работают эти функции.

```
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            return reduce_fn(accum, fn(input));
        };
    };
}
```

4. Теперь реализуем функцию `filter`. Она работает точно так же, как и функция `map`, но *не затрагивает* входные данные, в то время как `map` *преобразует* их с помощью функции `transform`. Вместо этого принимаем функцию-предикат и *опускаем* те входные переменные, которые не соответствуют данному предикату, не выполняя для них функцию `reduce`.

```
template <typename T>
auto filter(T predicate)
{
```

5. Два лямбда-выражения имеют такие же сигнатуры функций, что и выражения в функции `map`. Единственное отличие заключается в следующем: входной параметр остается неизменным. Функция-предикат используется для определения того, будем ли мы вызывать функцию `reduce_fn` для входных данных или же получим доступ к аккумулятору, не внося никаких изменений.

```
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            if (predicate(input)) {
                return reduce_fn(accum, input);
            } else {
                return accum;
            }
        };
    };
}
```

6. Теперь воспользуемся этими вспомогательными функциями. Создадим экземпляры итераторов, которые позволяют считать целочисленные значения из стандартного потока ввода:

```
int main()
{
    std::istream_iterator<int> it {std::cin};
    std::istream_iterator<int> end_it;
```

7. Далее определим функцию-предикат `even`, которая возвращает значение `true`, если перед нами *четное число*. Функция преобразования `twice` умножает свой целочисленный параметр на `2`:

```
    auto even  ([](int i) { return i % 2 == 0; });
    auto twice ([](int i) { return i * 2; });
```

8. Функция `std::accumulate` принимает диапазон значений и *аккумулирует* их. Аккумулирование по умолчанию означает *суммирование* значений с помощью оператора `+`. Мы хотим предоставить собственную функцию аккумулирования. Таким образом, хранить *сумму* значений не нужно. Мы присвоим каждое значение из диапазона разыменованному итератору `it`, а затем вернем его после *продвижения* вперед.

```
    auto copy_and_advance ([](auto it, auto input) {
        *it = input;
        return ++it;
    });
```

9. Наконец мы готовы собрать все воедино. Мы итерируем по стандартному потоку ввода и предоставляем вывод `ostream_iterator`, который выдает значения в консоль. Объект функции `copy_and_advance` работает с этим итератором вывода, присваивая ему целые числа, полученные от пользователя. По сути, данное действие выводит на экран присвоенные элементы. Но мы хотим видеть только четные числа, полученные от пользователя, и умножить их. Для этого оборачиваем функцию `copy_and_advance` в *фильтр* `even`, а затем — в *преобразователь* `twice`.

```
    std::accumulate(it, end_it,
        std::ostream_iterator<int>{std::cout, ", "},
        filter(even)(
            map(twice)(
                copy_and_advance
            )
        ));
    std::cout << '\n';
}
```

10. Компиляция и запуск программы дадут следующий результат. Значения `1`, `3` и `5` отбрасываются, поскольку являются нечетными, а `2`, `4` и `6` выводятся на экран после их умножения на два.

```
$ echo "1 2 3 4 5 6" | ./transform_if
4, 8, 12,
```

Как это работает

Этот пример выглядит довольно сложным, поскольку мы активно использовали вложенные лямбда-выражения. Чтобы понять, как все работает, взглянем на внутреннее содержимое функции `std::accumulate`. Именно так она выглядит в обычной реализации, предлагаемой в библиотеке STL:

```
template <typename T, typename F>
T accumulate(InputIterator first, InputIterator last, T init, F f)
{
    for (; first != last; ++first) {
        init = f(init, *first);
    }
    return init;
}
```

Параметр функции `f` выполняет здесь остальную работу, а цикл собирает результаты в предоставленной пользователем переменной `init`. В обычном варианте использования диапазон итераторов может представлять собой вектор чисел, например `0, 1, 2, 3, 4`, а переменная `init` будет иметь значение `0`. Функция `f` является простой бинарной функцией, которая может определять *сумму* двух элементов с помощью оператора `+`.

В нашем примере цикл просто складывает все элементы и записывает результат в переменную `init`, это выглядит, например, так: `init = (((0 + 1) + 2) + 3) + 4`. Подобная запись помогает понять, что `std::accumulate` представляет собой функцию *свертки*. Выполнение свертки для диапазона значений означает применение бинарной операции для переменной-аккумулятора и каждого элемента диапазона пошагово (результат каждой операции является значением-аккумулятором для последующей операции). Поскольку эта функция обобщена, с ее помощью можно решать разные задачи, например реализовать функцию `std::transform_if`! Функция `f` в таком случае будет называться функцией *reduce* (свертки).

Очень прямолинейная реализация функции `transform_if` будет выглядеть так:

```
template <typename InputIterator, typename OutputIterator,
          typename P, typename Transform>
OutputIterator transform_if(InputIterator first, InputIterator last,
                            OutputIterator out,
                            P predicate, Transform trans)
{
    for (; first != last; ++first) {
        if (predicate(*first)) {
            *out = trans(*first);
            ++out;
        }
    }
    return out;
}
```

Данная реализация очень похожа на `std::accumulate`, если считать параметр `out` переменной `init` и каким-то образом заменить функцией `f` конструкцию `if-construct` и ее тело!

Мы на самом деле сделали это — создали данную конструкцию `if-construct` и ее тело с помощью объекта бинарной функции, предоставленного в качестве параметра функции `std::accumulate`:

```
auto copy_and_advance ([](auto it, auto input) {
    *it = input;
    return ++it;
});
```

Функция `std::accumulate` помещает переменную `init` в параметр бинарной функции `it`. Второй параметр — текущее значение из диапазона, получаемое в цикле. Мы предоставили *итератор вывода* как параметр `init` функции `std::accumulate`.. Таким образом, функция `std::accumulate` не считает сумму, а перемещает элементы, по которым итерирует, в другой диапазон. Это значит следующее: мы повторно реализовали функцию `std::copy`, которая пока что не имеет предикатов и преобразований.

Фильтрацию с помощью предиката добавим путем обертывания функции `copy_and_advance` в *другой* объект функции, который пользуется функцией-предикатом:

```
template <typename T>
auto filter(T predicate)
{
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            if (predicate(input)) {
                return reduce_fn(accum, input);
            } else {
                return accum;
            }
        };
    };
}
```

Эта конструкция на первый взгляд выглядит сложной, но посмотрим на конструкцию `if`. Если функция-предикат вернет значение `true`, то параметры будут перенаправлены функции `reduce_fn`, в роли которой в нашем случае выступает функция `copy_and_advance`. Если же предикат вернет значение `false`, то переменная `accum`, выступающая в роли переменной `init` функции `std::accumulate`, будет возвращена без изменений. Так мы реализуем ту часть операции фильтрации, где *пропускаем* элемент. Конструкция `if` находится внутри лямбда-выражения, которое имеет такую же сигнатуру бинарной функции, что и функция `copy_and_advance`; это делает ее хорошей заменой.

Теперь мы можем *отфильтровать* элементы, но все еще не выполняем их *преобразование*. Это делается с помощью вспомогательной функции `map`:

```
template <typename T>
auto map(T fn)
{
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
```

```
            return reduce_fn(accum, fn(input));
        };
    };
}
```

Теперь код выглядит гораздо проще. Он тоже содержит внутреннее лямбда-выражение, которое имеет такую же сигнатуру, как и функция `copy_and_advance`, поэтому способен заменить ее. Реализация просто направляет дальше входные значения, но притом *преобразует правый* параметр вызова бинарной функции с помощью функции `fn`.

Далее, когда мы воспользуемся этими вспомогательными функциями, напишем следующее выражение:

```
filter(even)(
    map(twice)(
        copy_and_advance
    )
)
```

Вызов `filter(even)` захватывает предикат `even` и дает функцию, которая принимает бинарную функцию, чтобы обернуть ее в другую бинарную функцию, выполняющую *фильтрацию*. Функция `map(twice)` делает то же самое с функцией преобразования `twice`, но оборачивает бинарную функцию `copy_and_advance` в другую бинарную функцию, всегда преобразующую правый параметр.

Не выполнив оптимизацию, мы получим сложнейшую конструкцию, состоящую из вложенных функций, которые вызывают другие функции и при этом выполняют совсем немного работы. Однако компилятор может легко оптимизировать подобный код. Полученная бинарная функция так же проста, как и результат более прямолинейной реализации функции `transform_if`. Мы ничего не теряем с точки зрения производительности, приобретая компонуемость, свойственную функциям, поскольку смогли объединить предикат `even` и функцию преобразования `twice` столь же легко, как если бы на их месте были детали «Лего».

Генерируем декартово произведение на основе любых входных данных во время компиляции

Лямбда-выражения и наборы параметров можно использовать для решения сложных задач. В этом разделе мы реализуем объект функции, который принимает произвольное количество входных параметров и генерирует *декартово произведение* данного множества, умноженного *само на себя*.

Декартово произведение — математическая операция. Она обозначается как `A x B`, что означает «декартово произведение множества `A` на множество `B`». Результатом будет *одно множество*, содержащее пары *всех* комбинаций элементов из множеств `A` и `B`. Операция, по сути, означает *комбинирование каждого элемента из множества A с каждым элементом множества B*. Эта операция показана на рис. 4.3.

		B		
		1	2	3
A	x	(x, 1)	(x, 2)	(x, 3)
	y	(y, 1)	(y, 2)	(y, 3)
	z	(z, 1)	(z, 2)	(z, 3)

Рис. 4.3

Согласно схеме, если `A = (x, y, z)`, а `B = (1, 2, 3)`, то декартово произведение этих множеств будет равно `(x, 1)`, `(x, 2)`, `(x, 3)`, `(y, 1)`, `(y, 2)` и т. д.

Если мы решим, что множества `A` и `B` *одинаковы*, например `(1, 2)`, то их декартово произведение будет равно `(1, 1)`, `(1, 2)`, `(2, 1)` и `(2, 2)`. В некоторых случаях это может оказаться избыточным, поскольку комбинация элементов с *самими собой* (например, `(1, 1)`) или избыточные комбинации `(1, 2)` и `(2, 1)` способны стать ненужными. В таких случаях декартово произведение можно отфильтровать с помощью простого правила.

В этом разделе мы реализуем декартово произведение, не используя циклы, но применяя лямбда-выражения и распаковку набора параметров.

Как это делается

В примере мы реализуем объект функции, принимающий функцию `f` и набор параметров. Объект *создаст* декартово произведение набора параметров, *отфильтрует* избыточные части и *вызовет* для каждой пары функцию `f`.

1. Включим только тот заголовочный файл STL, который нужен для печати:

```
#include <iostream>
```

2. Затем определим простую вспомогательную функцию, которая выводит на экран пару значений, и начнем реализовывать функцию `main`:

```
static void print(int x, int y)
{
    std::cout << "(" << x << ", " << y << ")\n";
}
int main()
{
```

3. Теперь начинается сложная часть. Сначала реализуем вспомогательную функцию для функции `cartesian`, которую напишем на следующем шаге. Данная функция принимает параметр `f`, являющийся функцией вывода на экран. Другие ее параметры — это `x` и набор параметров `rest`. Они содержат реальные элементы, для которых мы будем искать декартово произведение. Взглянем на выражение `f(x, rest)`: для `x=1` и `rest=2, 3, 4` мы получим вызовы `f(1, 2); f(1, 3); f(1, 4);`. Проверка `(x < rest)` нужна для избавления от избыточности в сгенерированных парах. Мы рассмотрим этот вопрос более подробно позднее.

```
    constexpr auto call_cart (
        [=](auto f, auto x, auto ...rest) constexpr {
```

```
            (void)std::initializer_list<int>{
                (((x < rest)
                    ? (void)f(x, rest)
                    : (void)0)
                ,0)...
            };
        });
```

4. Функция `cartesian` — самая сложная часть кода всего примера. Она принимает набор параметров `xs` и возвращает захватывающий его объект функции. Полученный объект функции принимает объект функции `f`.

   Для набора параметров `xs=1, 2, 3` внутреннее лямбда-выражение сгенерирует следующие вызовы: `call_cart(f, 1, 1, 2, 3); call_cart(f, 2, 1, 2,3); call_cart(f, 3, 1, 2, 3);`. Из этого набора вызовов можно сгенерировать все необходимые пары произведения.

   Обратите внимание: мы *дважды* используем нотацию `...` для распаковки набора параметров `xs`, что на первый взгляд может показаться странным. Первое включение конструкции `...` распаковывает весь набор параметров `xs` в вызов `call_cart`. Второе включение приводит к нескольким вызовам функции `call_cart`, имеющим разные *вторые* параметры.

```
    constexpr auto cartesian ([=](auto ...xs) constexpr {
        return [=] (auto f) constexpr {
            (void)std::initializer_list<int>{
                ((void)call_cart(f, xs, xs...), 0)...
            };
        };
    });
```

5. Теперь сгенерируем декартово произведение для численного множества `1, 2, 3` и выведем полученные пары на экран. Если не учитывать избыточные пары, то мы должны получить следующий результат: `(1, 2)`, `(2, 3)` и `(1, 3)`. Другие комбинации невозможны при условии, что не важен порядок и не нужны одинаковые числа в паре. То есть не нужны пары вроде `(1, 1)`, а пары `(1, 2)` и `(2, 1)` считаются *одинаковыми*.

   Сначала сгенерируем объект функции, который содержит все возможные пары и принимает функцию `print`. Далее используем его, чтобы позволить вызывать данную функцию для всех этих пар. Объявляем переменную `print_cart` с модификатором `constexpr`; это позволит гарантировать, что хранимый ею объект функции (и все сгенерированные пары) будет создаваться во время компиляции:

```
    constexpr auto print_cart (cartesian(1, 2, 3));

    print_cart(print);
}
```

6. Компиляция и запуск программы дадут следующий ожидаемый результат. Можно убрать условие `(x < xs)` из функции `call_cart`, чтобы увидеть полное декартово произведение, содержащее избыточные пары и пары с одинаковыми номерами:

```
$ ./cartesian_product
(1, 2)
(1, 3)
(2, 3)
```

Как это работает

Мы создали еще одну очень сложную конструкцию с помощью лямбда-выражений. Но после того как разберемся с ней, нас больше не запутают никакие другие лямбда-выражения!

Взглянем на нее более внимательно. Нам следует получить представление о том, что должно произойти (рис. 4.4).

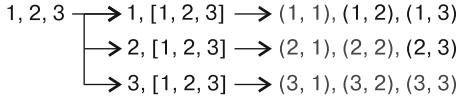

Рис. 4.4

Работа проходит в три шага.

1. Берем наше множество `1, 2, 3` и создаем на его основе *три новых*. Первая часть каждого из этих множеств — один элемент множества, а вторая — все множества.
2. Объединяем первый элемент с каждым элементом множества и получаем все *пары*.
3. Из полученных пар выбираем те, которые *не являются избыточными* (например, пары `(1, 2)` и `(2, 1)` избыточны) и не содержат одинаковых чисел (как, скажем, `(1, 1)`).

Теперь вернемся к реализации:

```
constexpr auto cartesian ([=](auto ...xs) constexpr {
     return [=](auto f) constexpr {
         (void)std::initializer_list<int>{
             ((void)call_cart(f, xs, xs...), 0)...
         };
     };
 });
```

Внутреннее выражение, `call_cart(xs, xs...)`, явно представляет собой разделение множества `(1, 2, 3)` на эти новые множества наподобие `1, [1, 2, 3]`. Полное выражение, `((void)call_cart(f, xs, xs...), 0)...`, имеющее снаружи

дополнительную конструкцию `...`, выполняет такое разделение для каждого значения множества, так что мы также получаем множества `2, [1, 2, 3]` и `3, [1, 2, 3]`.

Шаги 2 и 3 выполняются с помощью `call_cart`:

```
auto call_cart ([](auto f, auto x, auto ...rest) constexpr {
    (void)std::initializer_list<int>{
        (((x < rest)
            ? (void)f(x, rest)
            : (void)0)
        ,0)...
    };
});
```

Параметр `x` всегда содержит одно значение, взятое из множества, а `rest` включает все множество. Опустим условие `(x < rest)`. Здесь выражение `f(x, rest)` и распакованный набор параметров `...` генерируют вызовы функции `f(1, 1)`, `f(1, 2)` и т. д., что приводит к появлению пар на экране. Это был шаг 2.

Шаг 3 достигается за счет фильтрации всех пар, к которым применяется условие `(x < rest)`.

Мы указали, что все лямбда-выражения и переменные, их содержащие, имеют модификатор `constexpr`. Это гарантирует, что компилятор оценит их код во время компиляции и скомпилирует бинарный файл, который уже содержит все числовые пары, вместо того, чтобы делать это во время работы программы. Обратите внимание: так происходит *только* в том случае, если все аргументы, которые мы предоставляем функции с модификатором `constexpr`, *известны на этапе компиляции*.

5 Основы работы с алгоритмами STL

В этой главе:

- копирование элементов из одних контейнеров в другие;
- сортировка контейнеров;
- удаление конкретных элементов из контейнеров;
- преобразование содержимого контейнеров;
- поиск элементов в упорядоченных и неупорядоченных векторах;
- ограничение допустимых значений вектора конкретным численным диапазоном с помощью `std::clamp`;
- определение шаблонов в строках с помощью `std::search` и выбор оптимальной реализации;
- выборка данных из крупных векторов;
- создание перестановок во входных последовательностях;
- реализация инструмента для слияния словарей.

Введение

Библиотека STL содержит не только структуры данных, но и *алгоритмы*. В то время как структуры помогают *хранить* и *поддерживать* данные разными способами для различных целей, алгоритмы позволяют выполнять конкретные *преобразования* данных в этих структурах.

Рассмотрим стандартную задачу, например сложение элементов вектора. Это можно без труда сделать с помощью цикла, в котором мы суммируем все элементы вектора и поместим их в переменную-аккумулятор `sum`:

```
vector<int> v {100, 400, 200 /*, ... */ };

int sum {0};
for (int i : v) { sum += i; }

cout << sum << '\n';
```

Поскольку эта задача является стандартной, для ее решения предусмотрен алгоритм STL:

```
cout << accumulate(begin(v), end(v), 0) << '\n';
```

В таком случае при создании вручную цикл занимает не намного больше места и прочесть его не сложнее, чем одну строку, в которой говорится, что она делает: `accumulate`. Во многих ситуациях, однако, может возникнуть неловкий момент: приходится читать состоящий из десятка строк цикл только затем, чтобы узнать, что он решает стандартную задачу X, вместо того чтобы увидеть одну строку кода, в которой используется стандартный алгоритм, чье имя явно указывает на то, какую задачу он решает, например `accumulate`, `copy`, `move`, `transform` или `shuffle`.

Основная идея заключается в том, чтобы предоставить множество алгоритмов, которые программисты могут использовать в повседневных задачах, не реализуя каждый раз повторно. Таким образом, разработчики могут просто применить готовый алгоритм и сконцентрироваться на решении новых задач вместо того, чтобы тратить время на проблемы, *уже решенные средствами STL*. Еще одно преимущество — корректность. При реализации одного и того же решения снова и снова возникает вероятность того, что в одной из попыток может появиться небольшая *ошибка*. В результате вы можете оказаться в *неприятной* ситуации, если коллега укажет на ошибку в коде во время обзора, а ведь вы вместо того, чтобы писать свой код, могли воспользоваться стандартным алгоритмом.

Еще одним важным качеством алгоритмов STL является *эффективность*. Многие из них предоставляют несколько *специализированных* реализаций одного алгоритма, по-разному решающих задачу, в зависимости от *типа итератора*, для которого они используются. Например, обнулить элементы вектора, содержащего целые числа, можно с помощью алгоритма `std::fill`. Поскольку итератор вектора может указать компилятору, что итерирует по *непрерывной* памяти, он может выбрать ту реализацию алгоритма `std::fill`, которая задает процедуру C `memset`. Если программист изменяет тип контейнера с `vector` на `list`, то алгоритм STL больше не может использовать процедуру `memset` и должен итерировать по списку, чтобы обнулять элементы по одному. В том случае, если программист сам задействует процедуру `memset`, алгоритм обнуления сможет работать только с векторами и массивами, поскольку другие структуры не хранят данные во фрагментах непрерывной памяти. В большинстве случаев не стоит изобретать велосипед, поскольку авторы библиотеки STL уже реализовали эти идеи и вам ничто не мешает воспользоваться их трудом.

Подытожим. Алгоритмы STL предоставляют такие преимущества.

- **Легкость сопровождения:** по названию алгоритма сразу понятно, что именно он делает. Явные циклы зачастую труднее прочесть, и им нужно знать о том, какие именно структуры данных будут применяться, в отличие от стандартных алгоритмов.

- **Правильность:** библиотеку STL создавали и анализировали профессионалы, она используется и тестируется многими программистами, и вам, скорее всего, не удастся достичь той же степени правильности, если вы будете самостоятельно реализовывать сложные фрагменты алгоритмов.
- **Эффективность:** по умолчанию алгоритмы STL эффективны как минимум настолько же, насколько эффективны циклы, написанные вручную.

Большинство алгоритмов работают с *итераторами*. Принципы работы итераторов мы уже рассмотрели в главе 3. В настоящей главе сконцентрируемся на использовании алгоритмов STL для решения конкретных задач, чтобы понять, какие возможности они предоставляют. Разбор *всех* алгоритмов превратит эту книгу в очень скучный справочный материал по C++, а подобное руководство уже доступно для широкого круга читателей.

Самый лучший способ стать мастером STL заключается в том, чтобы всегда иметь справочный материал по C++ под рукой или хотя бы в закладках браузера. При решении какой-нибудь задачи каждый программист должен задуматься: «Существует ли в STL алгоритм для решения моей задачи?» — прежде чем писать код самостоятельно.

Хорошая и полная справка по C++ доступна по адресу http://en.cppreference.com/w/. Кроме того, этот материал можно скачать для чтения в режиме офлайн.

На собеседованиях хорошее знание алгоритмов STL зачастую считается показателем глубоких знаний языка C++.

Копируем элементы из одних контейнеров в другие

Большинство важных структур данных STL поддерживают итераторы. Это значит, что вы как минимум сможете получить итераторы с помощью функций `begin()` и `end()`, которые указывают на полезные данные и позволяют итерировать по ним. Перебор всегда выглядит одинаково, независимо от того, по какой структуре данных выполняется.

Можно получить итераторы для векторов, списков, двунаправленных очередей, ассоциативных массивов и т. д. С помощью адаптеров итераторов мы можем получить итераторы для файлов, а также стандартных потоков ввода и вывода. Более того, как вы видели в предыдущей главе, можно даже обернуть интерфейсы итераторов вокруг алгоритмов. Теперь, когда итераторы помогают нам получить доступ ко всему, можно объединить их с алгоритмами STL, которые принимают итераторы в качестве параметров.

На примере алгоритма `std::copy` можно легко понять, как итераторы помогают абстрагировать природу разных структур данных. Он просто копирует элементы из одного набора итераторов в итератор вывода. При использовании

подобных алгоритмов не нужно знать о природе структуры данных, с которой мы работаем.

Как это делается

В этом примере мы разберем разные варианты алгоритма `std::copy`.

1. Сначала включим заголовочные файлы для всех структур данных, которые будем использовать. Кроме того, объявим, что задействуем пространство имен `std`:

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <tuple>
#include <iterator>
#include <algorithm>

using namespace std;
```

2. Далее будем использовать пары, содержащие целое число и строку. Чтобы красиво вывести их на экран, нужно перегрузить оператор потока `<<`:

```
namespace std {
ostream& operator<<(ostream &os, const pair<int, string> &p)
{
    return os << "(" << p.first << ", " << p.second << ")";
}
}
```

3. В функции `main` заполним вектор пар «число — строка» некоторыми значениями по умолчанию. Кроме того, объявим переменную `map`, связывающую численные и строковые значения:

```
int main()
{
    vector<pair<int, string>> v {
        {1, "one"}, {2, "two"}, {3, "three"},
        {4, "four"}, {5, "five"}};
    map<int, string> m;
```

4. Теперь воспользуемся алгоритмом `std::copy_n`, чтобы скопировать три пары из вектора в ассоциативный массив. Поскольку векторы и ассоциативные массивы значительно отличаются друг от друга, нужно выполнить преобразование элементов вектора с помощью параметра адаптера `insert_iterator`. Его предоставляет функция `std::inserter`. Пожалуйста, всегда помните о том, что использование алгоритмов наподобие `std::copy_n` в комбинации с итераторами вставки — наиболее *обобщенный* способ скопировать/вставить элементы

в другие структуры данных, хотя и не самый *быстрый*. Зачастую эффективнее всего для вставки элементов применять функции-члены конкретных структур данных.

```
copy_n(begin(v), 3, inserter(m, begin(m)));
```

5. Выведем содержимое ассоциативного массива. На протяжении книги мы часто выводили на экран содержимое контейнера с помощью функции `std::copy`. Итератор `std::ostream_iterator` значительно помогает в этом вопросе, поскольку позволяет считать выходные данные пользовательской консоли еще одним контейнером, в который можно скопировать данные.

```
auto shell_it (ostream_iterator<pair<int, string>>{cout,
                                                   ", "});
copy(begin(m), end(m), shell_it);
cout << '\n';
```

6. Опустошим ассоциативный массив, чтобы подготовить его к следующему эксперименту. В этот раз мы переместим в него все элементы вектора:

```
m.clear();
move(begin(v), end(v), inserter(m, begin(m)));
```

7. Теперь снова выведем на экран содержимое ассоциативного массива. Более того, поскольку алгоритм `std::move` изменяет еще и *источник* данных, выведем на экран и содержимое вектора. Таким образом, можно увидеть, что с ним произошло, когда он стал источником данных.

```
copy(begin(m), end(m), shell_it);
cout << '\n';
copy(begin(v), end(v), shell_it);
cout << '\n';
}
```

8. Скомпилируем программу, запустим ее и посмотрим на результат. Первые две строки выглядят просто. Они отражают содержимое ассоциативного массива после применения алгоритмов `copy_n` и `move`. Третья строка выглядит интереснее, поскольку в ней показывается, что строки вектора, который мы использовали в качестве источника данных, теперь пусты. Это произошло потому, что содержимое строк было не скопировано, а *перемещено* (то есть ассоциативный массив применяет данные строк из кучи, на которые ссылались объекты строк, размещенные в векторе).

   Мы не должны получать доступ к элементам, находящимся в источнике данных, до того, как изменятся их значения, но в целях эксперимента проигнорируем данное правило.

```
$ ./copying_items
(1, one), (2, two), (3, three),
(1, one), (2, two), (3, three), (4, four), (5, five),
(1, ), (2, ), (3, ), (4, ), (5, ),
```

Как это работает

Поскольку алгоритм `std::copy` является одним их самых простых алгоритмов STL, его реализация очень короткая. Взглянем на то, как его можно было бы реализовать:

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator it, InputIterator end_it,
                    OutputIterator out_it)
{
    for (; it != end_it; ++it, ++out_it) {
        *out_it = *it;
    }
    return out_it;
}
```

Этот код выглядит как попытка вручную реализовать копирование элементов из одного итерабельного диапазона в другой. Кто-то может задаться вопросом: «Почему бы и не реализовать его вручную? Цикл выглядит достаточно просто, и мне даже не понадобится возвращаемое значение». Это, конечно, хороший вопрос.

Хотя алгоритм `std::copy` не самый лучший пример и не может продемонстрировать, как код становится короче, другие алгоритмы выглядят гораздо сложнее. Это может быть неочевидно, но для алгоритмов STL выполняется скрытая оптимизация. Если мы будем пользоваться алгоритмом `std::copy` для структур данных, которые хранят свои элементы в непрерывной памяти (например, `std::vector` и `std::array`), *и* самим элементам будет *легко присвоить копию*, то компилятор выберет совершенно другую реализацию (предполагается, что типом итератора является указатель):

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator it, InputIterator end_it,
                    OutputIterator out_it)
{
    const size_t num_items (distance(it, end_it));
    memmove(out_it, it, num_items * sizeof(*it));
    return it + num_items;
}
```

Перед вами упрощенная версия реализации алгоритма `std::copy` с помощью `memmove`. Она работает быстрее, чем стандартная версия с циклами, и в то же время не такая читабельная. Но тем не менее пользователи алгоритма `std::copy` автоматически получают от него выгоду, если типы их аргументов соответствуют требованиям, выполнение которых необходимо для оптимизации. Компилятор выбирает для заданного алгоритма самую быструю реализацию из возможных, а код пользователя выражает, что именно делает алгоритм, не обрастая ненужными деталями.

Алгоритмы STL зачастую предоставляют наилучшее сочетание *читабельности* и *оптимальности реализации*.

Типам обычно можно легко присвоить копию, если они состоят из одного или нескольких (обернутых в класс/структуру) скалярных типов или классов, которые легко переместить с помощью memcopy/memmove, не вызывая определенный пользователем оператор присваивания копии.

Кроме того, мы использовали алгоритм `std::move`. Он работает точно так же, как и алгоритм `std::copy`, но применяет `std::move(*it)` к итератору источника в цикле, чтобы преобразовать значения `lvalues` к значениям `rvalues`. Это позволяет компилятору выбрать оператор присваивания перемещением целевого объекта вместо оператора присваивания копированием. Для многих сложных объектов данный способ *работает* быстрее, но при этом *уничтожается* исходный объект.

Сортируем контейнеры

Сортировка значений — довольно стандартная процедура, которую можно выполнить несколькими способами. Это известно каждому изучавшему информатику студенту, которого заставляли разбирать большинство существующих алгоритмов сортировки.

Поскольку данную задачу уже когда-то решили, программисты не должны *снова* тратить на нее время; разве что для обучения.

Как это делается

В этом примере мы поработаем с алгоритмами `std::sort` и `std::partial_sort`.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Мы будем несколько раз выводить на экран состояние вектора, содержащего целые числа, поэтому напишем для данной задачи небольшую процедуру:

```
static void print(const vector<int> &v)
{
    copy(begin(v), end(v), ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```

3. Начнем с вектора, содержащего некоторые числа:

```
int main()
{
    vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

4. Поскольку мы перемешаем значения вектора несколько раз, чтобы исследовать разные функции сортировки, нам понадобится генератор случайных чисел:

```
    random_device rd;
    mt19937 g {rd()};
```

5. Функция `std::is_sorted` говорит о том, было ли отсортировано содержимое контейнера. Эта строка должна вывести значение `1`:

```
    cout << is_sorted(begin(v), end(v)) << '\n';
```

6. С помощью `std::shuffle` мы перемешаем содержимое вектора, чтобы позднее отсортировать его. Первые два аргумента указывают на диапазон данных, который будет перемешан, а третий — генератор случайных чисел:

```
    shuffle(begin(v), end(v), g);
```

7. Функция `is_sorted` должна теперь вернуть значение `false`, а на экране мы увидим значение `0`, значения вектора должны остаться прежними, но их порядок изменится. Мы увидим это, когда выведем на экран его содержимое:

```
    cout << is_sorted(begin(v), end(v)) << '\n';
    print(v);
```

8. Теперь восстановим исходный порядок элементов с помощью алгоритма `std::sort`. Вызвав те же команды на консоли, мы увидим значения вектора, отсортированные по возрастанию:

```
    sort(begin(v), end(v));
    cout << is_sorted(begin(v), end(v)) << '\n';
    print(v);
```

9. Еще одна интересная функция — `std::partition`. Возможно, мы не хотим полностью сортировать список, поскольку достаточно поместить в начало те элементы, чье значение не превышает некий предел. Секционируем вектор, чтобы переместить все элементы, значение которых меньше `5`, в начало, и выведем результат на экран:

```
    shuffle(begin(v), end(v), g);
    partition(begin(v), end(v), [] (int i) { return i < 5; });
    print(v);
```

10. Следующая функция, связанная с сортировкой, — `std::partial_sort`. Ее можно использовать для сортировки содержимого контейнера, но не в полной мере. Эта функция поместит `N` самых маленьких элементов в начало контейнера

в отсортированном виде. Остальная часть элементов останется во второй половине, они не будут отсортированы.

```
shuffle(begin(v), end(v), g);
auto middle (next(begin(v), int(v.size()) / 2));
partial_sort(begin(v), middle, end(v));
print(v);
```

11. Что, если мы хотим отсортировать структуру данных, *не поддерживающую* оператора сравнения? Определим такую структуру и создадим следующий вектор элементов:

```
struct mystruct {
    int a;
    int b;
};
vector<mystruct> mv {{5, 100}, {1, 50}, {-123, 1000},
                     {3, 70}, {-10, 20}};
```

12. Функция `std::sort` опционально принимает в качестве третьего аргумента функцию сравнения. Воспользуемся данным обстоятельством и передадим ей такую функцию. Для демонстрации того, что это возможно, мы будем сравнивать элементы на основе *второго* поля, `b`. Таким образом, элементы отсортируются на основе поля `mystruct::b`, а не поля `mystruct::a`:

```
sort(begin(mv), end(mv),
     [] (const mystruct &lhs, const mystruct &rhs) {
         return lhs.b < rhs.b;
     });
```

13. Последним шагом будет вывод на экран упорядоченного вектора, содержащего элементы типа `mystruct`.

```
    for (const auto &[a, b] : mv) {
        cout << "{" << a << ", " << b << "} ";
    }
    cout << '\n';
}
```

14. Скомпилируем и запустим программу.

    Первое значение `1` получено от вызова функции `std::is_sorted call` после инициализации упорядоченного вектора. Далее мы перемешали значения вектора и получили значение `0` после второго вызова функции `is_sorted`. В третьей строке показаны все элементы вектора после перемешивания. Следующее значение `1` — это результат вызова функции `is_sorted` после повторной сортировки с помощью алгоритма `std::sort`.

    Далее мы снова перемешали вектор и секционировали его, задействовав алгоритм `std::partition`. Можно увидеть, что все элементы, чье значение не превышает `5`, находятся слева от значения `5` в векторе. Остальные элементы,

чье значение превышает 5, стоят справа. За исключением этого они выглядят упорядоченными.

В предпоследней строке показан результат вызова алгоритма `std::partial_sort`. Все элементы в первой половине вектора выглядят отсортированными, а остальные — нет.

В последней строке мы видим наш вектор, содержащий экземпляры типа `mystruct`. Они строго отсортированы по значению *второго* члена.

```
$ ./sorting_containers
1
0
7, 1, 4, 6, 8, 9, 5, 2, 3, 10,
1
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
1, 2, 4, 3, 5, 7, 8, 10, 9, 6,
1, 2, 3, 4, 5, 9, 8, 10, 7, 6,
{-10, 20} {1, 50} {3, 70} {5, 100} {-123, 1000}
```

Как это работает

Мы использовали разные алгоритмы, связанные с сортировкой (табл. 5.1).

Таблица 5.1

Алгоритм	Предназначение
std::sort	Принимает в качестве аргумента диапазон данных и попросту сортирует их
std::is_sorted	Принимает в качестве аргумента диапазон данных и сообщает, отсортирован ли он
std::shuffle	Выполняет операцию, похожую на сортировку наоборот: принимает диапазон данных и перемешивает его элементы
std::partial_sort	Принимает в качестве аргумента диапазон данных и еще один итератор, который указывает, до какой позиции следует сортировать все элементы. Остальные элементы, стоящие после этого итератора, останутся неотсортированными
std::partition	Принимает диапазон данных и функцию-предикат. Все элементы, для которых данная функция возвращает значение true, перемещаются в начало диапазона. Остальные элементы переходят в его конец

Для объектов, не имеющих реализации оператора сравнения `<`, можно предоставить собственные функции сравнения. Этим функциям всегда следует иметь сигнатуру `bool имя_функции(const T &lhs, const T &rhs)`, и они не должны вызывать побочные эффекты во время выполнения.

Существуют и другие алгоритмы, такие как `std::stable_sort`, которые сортируют элементы, но сохраняют порядок элементов с одинаковым ключом сортировки, и `std::stable_partition`.

Алгоритм std::sort имеет разные реализации. В зависимости от природы аргументов итератора его можно реализовать как сортировку методом выбора, вставки или слияния или полностью оптимизировать для небольшого диапазона элементов. С точки зрения пользователя нас это, как правило, не волнует.

Удаляем конкретные элементы из контейнеров

Копирование, преобразование и фильтрация, возможно, наиболее распространенные операции, которые можно выполнить с диапазоном данных. В этом разделе мы сконцентрируемся на фильтрации элементов.

Фильтрация элементов из структур данных (или простое удаление конкретных элементов) работает по-разному для разных структур данных. В связанных списках (например, `std::list`) элемент может быть удален, если вы заставите его предшественника указывать на элемент, стоящий после удаляемого. После такого удаления узла из цепи ссылок этот узел можно вернуть распределителю ресурсов. В непрерывных структурах данных (`std::vector`, `std::array` и в некоторой степени `std::deque`) элементы можно удалить, только перезаписав их другими элементами. Если позиция элемента помечается как удаляемая, то все элементы, стоящие после него, должны сдвинуться вперед, чтобы заполнить пропуск. Кажется, возни для простой операции слишком много, но, например, просто удалить пробельные символы из строки можно с помощью относительно небольшого количества строк кода.

Нужно удалить элемент, не оглядываясь на тип нашей структуры данных. Здесь помогут алгоритмы `std::remove` и `std::remove_if`.

Как это делается

В этом примере мы преобразуем содержимое вектора, удалив из него элементы разными способами.

1. Импортируем все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Короткая вспомогательная функция `print` выведет на экран содержимое вектора:

```
void print(const vector<int> &v)
{
    copy(begin(v), end(v), ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```

3. Начнем с примера вектора, содержащего некие простые целочисленные значения. Мы также выведем его на экран, чтобы увидеть изменения, которые внесет наша функция.

```
int main()
{
    vector<int> v {1, 2, 3, 4, 5, 6};
    print(v);
```

4. Теперь удалим из вектора все элементы со значением 2. Функция `std::remove` переместит другие элементы так, что единственное значение 2, присутствующее в векторе, испарится. Поскольку длина реального содержимого вектора сократилась после удаления элементов, функция `std::remove` вернет итератор, указывающий на *новую конечную позицию*. Элементы, стоящие между новым и старым конечными итераторами, считаются мусором, так что дадим вектору команду *удалить* их. Окружаем две строки, связанные с удалением этих элементов, новой областью видимости, поскольку итератор `new_end` в дальнейшем станет некорректным, вследствие чего может мгновенно выйти из области видимости:

```
    {
        const auto new_end (remove(begin(v), end(v), 2));
        v.erase(new_end, end(v));
    }
    print(v);
```

5. Теперь удалим все нечетные числа. Для этого реализуем предикат, который сообщит, является ли число нечетным, и передадим его в функцию `std::remove_if`, принимающую подобные предикаты:

```
    {
        auto odd_number ([](int i) { return i % 2 != 0; });
        const auto new_end (
            remove_if(begin(v), end(v), odd_number));
        v.erase(new_end, end(v));
    }
    print(v);
```

6. Далее поработаем с алгоритмом `std::replace`. Воспользуемся им, чтобы переписать все значения `4` значениями `123`. Функция `std::replace` существует и в форме `std::replace_if`, которая также принимает функции-предикаты.

```
    replace(begin(v), end(v), 4, 123);
    print(v);
```

7. Заполним вектор совершенно новыми значениями и создадим два новых пустых вектора, чтобы провести еще один эксперимент:

```
    v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v2;
    vector<int> v3;
```

8. Далее снова реализуем предикат для нечетных чисел и еще одну функцию-предикат, которая решает прямо противоположную задачу: сообщает, является ли заданное число четным:

```
auto odd_number  ([](int i) { return i % 2 != 0; });
auto even_number ([](int i) { return i % 2 == 0; });
```

9. Следующие две строки делают одно и то же: копируют *четные* значения в векторы `v2` и `v3`. В первой строке это делается с помощью алгоритма `std::remove_copy_if`, копирующего все из исходного контейнера в другой контейнер, который *не соответствует* ограничению, налагаемому предикатом. В другой строке используется алгоритм `std::copy_if`, который копирует все значения, *удовлетворяющие* ограничению, налагаемому предикатом:

```
remove_copy_if(begin(v), end(v),
               back_inserter(v2), odd_number);
copy_if(begin(v), end(v),
        back_inserter(v3), even_number);
```

10. Вывод на экран двух векторов должен дать одинаковый результат:

```
    print(v2);
    print(v3);
}
```

11. Скомпилируем и запустим программу. В первой строке показан вектор после инициализации. Во второй — вектор, из которого мы удалили все значения `2`. В следующей строке представлен результат удаления всех нечетных чисел. Перед четвертой строкой мы заменили все значения `4` на значения `123`.

    В последних двух строках показываются векторы `v2` и `v3`:

```
$ ./removing_items_from_containers
1, 2, 3, 4, 5, 6,
1, 3, 4, 5, 6,
4, 6,
123, 6,
2, 4, 6, 8, 10,
2, 4, 6, 8, 10,
```

Как это работает

Мы использовали различные алгоритмы, связанные с фильтрацией данных (табл. 5.2).

Таблица 5.2

Алгоритм	Предназначение
std::remove	Принимает в качестве аргументов диапазон значений и удаляемое значение, затем удаляет из диапазона заданное значение. Возвращает новый конечный итератор измененного диапазона данных

Продолжение ⇩

Таблица 5.2 *(продолжение)*

Алгоритм	Предназначение
std::replace	Принимает в качестве аргументов диапазон значений и два значения, а затем заменяет в диапазоне данных первое указанное значение на второе
std::remove_copy	Принимает в качестве аргументов диапазон данных, итератор вывода и значение, а затем копирует из диапазона все значения, не равные заданному, в конечный итератор
std::replace_copy	Работает аналогично std::replace и std::remove_copy. Исходный диапазон данных не изменяется
std::copy_if	Работает как std::copy, но дополнительно принимает в качестве аргумента функцию-предикат, чтобы скопировать только те значения, которые соответствуют заданному предикатом условию

Для каждого из перечисленных алгоритмов существует версия *_if, принимающая вместо значения функцию-предикат, которая затем выводит, какие значения будут удалены или заменены.

Преобразуем содержимое контейнеров

Если `std::copy` является самым простым алгоритмом STL для работы с диапазонами данных, то `std::transform` — второй по сложности алгоритм STL. Как и `copy`, он копирует элементы из одного диапазона данных в другой, но дополнительно принимает функцию преобразования. Она может изменить значение выходного типа до того, как это значение окажется в выходном диапазоне. Более того, данная функция может создать значение совершенно другого типа, что может быть полезно, если входной и выходной диапазоны данных имеют разные типы. Этот алгоритм прост в использовании, но в то же время очень полезен, вследствие чего он становится стандартным компонентом, который можно применять во многих повседневных программах.

Как это делается

В этом примере мы воспользуемся алгоритмом `std::transform`, чтобы изменить элементы вектора путем копирования.

1. Как обычно, сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`, что сэкономит немного времени:

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>
#include <iterator>

using namespace std;
```

2. В качестве примера исходной структуры данных возьмем вектор, содержащий простые целые числа:

```
int main()
{
    vector<int> v {1, 2, 3, 4, 5};
```

3. Теперь скопируем все элементы в итератор вывода `ostream_iterator`, чтобы вывести их на экран. Функция `transform` принимает объект функции, который принимает элементы, чей тип совпадает с типом элементов контейнера, и преобразует их во время каждой операции копирования. В данном случае мы вычисляем *квадрат* каждого числа, поэтому наш код выведет значения элементов вектора, возведенные в квадрат, избавив от необходимости сохранять их куда-либо.

```
    transform(begin(v), end(v),
        ostream_iterator<int>{cout, ", "},
        [] (int i) { return i * i; });
    cout << '\n';
```

4. Выполним еще одно преобразование. Например, из числа `3` можно сгенерировать точную и читабельную строку `3^2 = 9`. Следующая функция `int_to_string` делает именно это с помощью объекта `std::stringstream`:

```
    auto int_to_string ([](int i) {
        stringstream ss;
        ss << i << "^2 = " << i * i;
        return ss.str();
    });
```

5. Функция, которую мы только что реализовали, возвращает строковые значения на основе числовых. Мы также могли бы сказать, что она *соотносит* строки и числа. С помощью функции `transform` можно скопировать эти соотношения из вектора чисел в вектор строк:

```
    vector<string> vs;
    transform(begin(v), end(v), back_inserter(vs),
              int_to_string);
```

6. Выведем полученный результат на экран, и пример закончится:

```
    copy(begin(vs), end(vs),
         ostream_iterator<string>{cout, "\n"});
}
```

7. Скомпилируем и запустим программу:

```
$ ./transforming_items_in_containers
1, 4, 9, 16, 25,
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

Как это работает

Функция `std::transform` работает точно так же, как и `std::copy`, но вместе с копированием элементов в новый диапазон применяет к значению пользовательскую функцию преобразования до того, как итоговый результат будет присвоен итератору по месту назначения.

Выполняем поиск элементов в упорядоченных и неупорядоченных векторах

Иногда требуется определить, *лежит ли* заданное значение в рамках некоторого диапазона. Если да, то часто нужно изменить его или получить доступ к данным, связанным с ним.

Существует несколько стратегий для поиска элементов. В случае отсортированных элементов можно выполнить бинарный поиск — это быстрее, чем проверка всех элементов один за другим. Если элементы не отсортированы, то придется рассмотреть их по порядку.

Типичные алгоритмы поиска STL могут решить задачу обоими способами, поэтому было бы неплохо познакомиться с ними и узнать их характеристики. Данный раздел посвящен алгоритмам `std::find` (простой линейный поиск), `std::equal_range` (бинарный поиск) и их вариациям.

Как это делается

В этом примере мы используем алгоритмы линейного и бинарного поиска на небольшом диапазоне данных.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Наш диапазон данных будет состоять из структур типа `city`, в которых хранится название города и количество проживающих в нем человек:

```
struct city {
    string name;
    unsigned population;
};
```

3. Алгоритмы поиска должны уметь сравнивать элементы друг с другом, поэтому перегружаем оператор == для экземпляров типа city:

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

4. Кроме того, мы хотим выводить на экран экземпляры типа city, поэтому перегрузим оператор потока <<:

```
ostream& operator<<(ostream &os, const city &city) {
    return os << "{" << city.name << ", "
              << city.population << "}";
}
```

5. Поисковые функции, как правило, возвращают итераторы. Последние указывают на элемент, если таковой был найден, или же на конечный итератор заданного контейнера. В последнем случае нам нельзя получить доступ к такому итератору. Поскольку мы собираемся вывести на экран результаты поиска, реализуем функцию, которая возвращает другой объект функции, инкапсулирующий конечный итератор структуры данных. При использовании этого объекта функции для вывода объектов на экран он сравнит свой итератор-аргумент с конечным итератором, а затем выведет на экран либо сам элемент, либо строку <end>.

```
template <typename C>
static auto opt_print (const C &container)
{
    return [end_it (end(container))] (const auto &item) {
        if (item != end_it) {
            cout << *item << '\n';
        } else {
            cout << "<end>\n";
        }
    };
}
```

6. Начнем с рассмотрения примера вектора, содержащего названия немецких городов:

```
int main()
{
    const vector<city> c {
        {"Aachen",        246000},
        {"Berlin",       3502000},
        {"Braunschweig",  251000},
        {"Cologne",      1060000}
    };
```

7. С помощью этой вспомогательной функции создадим функцию, выводящую на экран экземпляры типа city и принимающую конечный итератор нашего вектора c:

```
    auto print_city (opt_print(c));
```

8. Используем алгоритм `std::find` для поиска элемента вектора, он сохранит элемент для города Кельна. Поначалу эта операция поиска выглядит бессмысленной, поскольку мы получаем именно тот элемент, который искали. Но ранее мы не знали его позицию в векторе, а функция `find` возвращает нам и ее. Однако мы могли бы, например, создать такой перегруженный оператор `==` для структуры `city`, который сравнивал бы только названия городов, не зная численности населения. Но это был бы пример плохого стиля программирования. На следующем шаге мы сделаем это по-другому.

```
{
    auto found_cologne (find(begin(c), end(c),
        city{"Cologne", 1060000}));
    print_city(found_cologne);
}
```

9. Не зная численности населения города, а также не задействуя оператор `==`, можно выполнить поиск только путем сравнения названия с содержимым вектора. Функция `std::find_if` принимает объект функции-предиката вместо конкретного значения. Таким образом можно выполнить поиск элемента для города Кельна, зная только его название:

```
{
    auto found_cologne (find_if(begin(c), end(c),
        [] (const auto &item) {
            return item.name == "Cologne";
        }));
    print_city(found_cologne);
}
```

10. Чтобы сделать операцию поиска чуть более выразительной, можно реализовать конструктор предикатов. Объект функции `population_higher_than` принимает численность населения и возвращает функцию, которая сообщает, превышает ли данная численность захваченное значение. Воспользуемся им, чтобы найти в нашем небольшом диапазоне данных немецкий город, в котором проживает более двух миллионов человек. Внутри нашего вектора таким городом является только Берлин.

```
{
    auto population_more_than ([](unsigned i) {
        return [=] (const city &item) {
            return item.population > i;
        };
    });
    auto found_large (find_if(begin(c), end(c),
        population_more_than(2000000)));
    print_city(found_large);
}
```

11. Использованные нами функции поиска проходят по контейнерам линейно. Поэтому они имеют коэффициент сложности *O(n)*. В STL также содержатся

функции бинарного поиска, которые работают за время *O(log(n))*. Сгенерируем новый диапазон данных, включающий лишь целочисленные значения, и напишем для него еще одну функцию `print`:

```
const vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto print_int (opt_print(v));
```

12. Функция `std::binary_search` возвращает булевы значения и просто уведомляет *о нахождении* элемента, но при этом его не возвращает. Важно, чтобы контейнер, для которого мы выполняем поиск, был *упорядочен*, поскольку в противном случае бинарный поиск не будет выполнен корректно.

```
bool contains_7 {binary_search(begin(v), end(v), 7)};
cout << contains_7 << '\n';
```

13. Чтобы получить искомые элементы, нужно задействовать другие функции STL. Одной из них является функция `std::equal_range`. Она возвращает не один итератор для искомого элемента, а сразу *пару*. Первый итератор указывает на первый элемент, чье значение *не меньше* искомого, второй — на первый элемент, значение которого *больше* искомого. В нашем диапазоне данных, содержащем числа от `1` до `10`, первый итератор указывает на значение `7`, поскольку это первый элемент, чье значение не меньше `7`. Второй итератор — на значение `8`, так как это первый элемент, значение которого больше `7`. При наличии в нашем диапазоне нескольких элементов `7` оба итератора представляли бы собой *поддиапазон* элементов.

```
auto [lower_it, upper_it] (
    equal_range(begin(v), end(v), 7));
print_int(lower_it);
print_int(upper_it);
```

14. Если нужно получить только один итератор, то можно воспользоваться функциями `std::lower_bound` или `std::upper_bound`. Первая возвращает итератор на первый элемент, чье значение не меньше искомого, вторая — на первый элемент, чье значение больше искомого:

```
    print_int(lower_bound(begin(v), end(v), 7));
    print_int(upper_bound(begin(v), end(v), 7));
}
```

15. Скомпилируем и запустим программу для подтверждения того, что наши предположения и результат ее работы совпадают:

```
$ ./finding_items
{Cologne, 1060000}
{Cologne, 1060000}
{Berlin, 3502000}
1
7
8
7
8
```

Как это работает

В этом примере мы использовали следующие алгоритмы поиска (табл. 5.3).

Таблица 5.3

Алгоритм	Предназначение
std::find	Принимает в качестве аргументов диапазон данных и значение для сравнения. Возвращает итератор, который указывает на первый элемент, равный значению для сравнения. Поиск выполняется линейно
std::find_if	Работает точно так же, как и std::find, но использует функцию-предикат вместо конкретного значения
std::binary_search	Принимает в качестве аргументов диапазон данных и значение для сравнения. Выполняет бинарный поиск и возвращает значение true, если диапазон содержит заданное значение
std::lower_bound	Принимает диапазон данных и значение для сравнения, а затем выполняет бинарный поиск первого элемента, чье значение не меньше значения для сравнения. Возвращает итератор, указывающий на этот элемент
std::upper_bound	Работает точно так же, как и std::lower_bound, но возвращает итератор на первый элемент, чье значение больше значения для сравнения
std::equal_range	Принимает диапазон данных и значение для сравнения, а затем возвращает пару итераторов. Первый итератор — результат вызова std::lower_bound, а второй — вызова std::upper_bound

Все описанные функции в качестве необязательного дополнительного аргумента принимают пользовательские функции сравнения. Таким образом, операцию поиска можно изменять, что мы и сделали в этом примере.

Рассмотрим более подробно принцип работы `std::equal_range`. Представьте, что у нас есть вектор, `v = {0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 8}` и мы вызываем метод `equal_range(begin(v), end(v), 7);`, чтобы выполнить бинарный поиск значения `7`. Функция `equal_range` возвращает пару итераторов, указывающих на нижнюю и верхнюю границы; они указывают на диапазон `{7, 7, 7}`, поскольку в векторе содержится именно столько значений `7`. Для большей ясности взгляните на рис. 5.1.

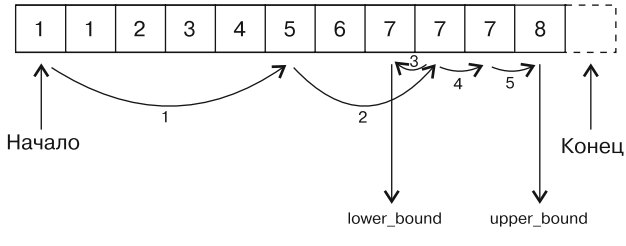

Рис. 5.1

Сначала функция `equal_range` использует типичный подход к выполнению бинарного поиска до тех пор, пока не встретит элементы, чье значение *не меньше*, чем искомое. Далее выполнение разбивается на вызовы методов `lower_bound` и `upper_bound`, чтобы объединить их возвращаемые значения в пару и затем вернуть эту пару вызывающей стороне.

Для получения функции бинарного поиска, которая просто возвращает первый элемент, соответствующий требованиям, мы могли бы реализовать следующее:

```
template <typename Iterator, typename T>
Iterator standard_binary_search(Iterator it, Iterator end_it, T value)
{
    const auto potential_match (lower_bound(it, end_it, value));
    if (potential_match != end_it && value == *potential_match) {
        return potential_match;
    }
    return end_it;
}
```

Эта функция использует вызов `std::lower_bound`, чтобы найти первый элемент, чье значение не меньше, чем `value`. Полученный результат `potential_match` может соответствовать одному из трех сценариев.

- Ни один элемент не соответствует нашему условию. В таком случае значение `potential_match` будет идентично `end_it`.
- Первый найденный элемент, соответствующий условию, *больше* искомого значения. В этом случае мы должны указать, что *не нашли* элемент, вернув `end_it`.
- Элемент, на который указывает `potential_match`, равен искомому значению. Поэтому он является не только *потенциальным*, но и *реальным* совпадением, и его можно вернуть.

Если наш тип `T` не поддерживает оператор `==`, то должен поддерживать хотя бы оператор `<` для выполнения бинарного поиска. Далее можно переписать сравнение как `!(value < *potential_match) && !(*potential_match < value)`. Если найденный элемент не меньше и не больше искомого, то он должен быть равен искомому.

Одной из потенциальных причин, по которой STL не предоставляет такую функцию, является отсутствие информации о том, что делать, если обнаружено несколько совпадений, как было показано на рис. 5.1, где вектор содержит несколько значений `7`.

Обратите внимание: структуры данных наподобие std::map, std::set и т. д. имеют собственные функции find. Они работают быстрее, чем более обобщенные алгоритмы, поскольку тесно связаны с реализациями структур данных.

Ограничиваем допустимые значения вектора конкретным численным диапазоном с помощью std::clamp

Во многих приложениях мы получаем из некоторого источника численные данные. Прежде чем у нас получится построить для них график или обработать их как-то иначе, может понадобиться нормализовать их, поскольку значения, сгенерированные случайным образом, чаще всего заметно отличаются друг от друга.

Обычно это значит, что нужно выполнить вызов `std::transform` для структуры данных, которая содержит случайные значения, а также вызвать простую функцию *масштабирования*. Но если мы не знаем, насколько большими или маленькими являются значения, то нужно сначала пройти по данным, чтобы найти правильные *измерения* для функции масштабирования.

Библиотека STL содержит полезные функции, которые можно применить для решения данной задачи: `std::minmax_element` и `std::clamp`. Эти функции можно использовать в совокупности с некоторыми лямбда-выражениями.

Как это делается

В этом примере мы нормализуем значения вектора из примера диапазона чисел двумя способами, использовав методы `std::minmax_element` и `std::clamp`.

1. Как и всегда, сначала включим следующие заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Реализуем функцию, принимающую минимальное и максимальное значения диапазона, а также новый максимум, что позволит проецировать значения из старого диапазона в более мелкий. Объект функции принимает подобные значения и возвращает другой объект функции, который выполняет именно такое преобразование. Для простоты этим значением будет `0`, поэтому независимо от того, какое смещение имели старые данные, нормализованные значения всегда будут соответствовать порядку нуля. Для повышения читабельности проигнорируем такой факт: значения `max` и `min` могут быть одинаковыми, что приведет к делению на ноль.

```
static auto norm (int min, int max, int new_max)
{
    const double diff (max - min);
```

```
    return [=] (int val) {
        return int((val - min) / diff * new_max);
    };
}
```

3. Еще один конструктор объектов функций с именем `clampval` возвращает объект функции, который захватывает значения `max` и `min` и вызывает функцию `std::clamp`, чтобы поместить получаемые значения в заданный диапазон:

```
static auto clampval (int min, int max)
{
    return [=] (int val) -> int {
        return clamp(val, min, max);
    };
}
```

4. Данные, которые мы будем нормализовывать, представляют собой вектор, содержащий разнообразные значения. Они могут быть, например, данными о температуре, высоте ландшафта или стоимости акций, изменяющимися с течением времени:

```
int main()
{
    vector<int> v {0, 1000, 5, 250, 300, 800, 900, 321};
```

5. Чтобы нормализовать эти данные, нужно иметь *наивысшее* и *наименьшее* значения. Здесь поможет функция `std::minmax_element`. Она возвращает пару итераторов, указывающих именно на эти два значения:

```
    const auto [min_it, max_it] (
        minmax_element(begin(v), end(v)));
```

6. Скопируем все значения из первого вектора во второй. Создадим экземпляр второго вектора и подготовим его к тому, чтобы он принял столько новых элементов, сколько содержится в первом векторе:

```
    vector<int> v_norm;
    v_norm.reserve(v.size());
```

7. С помощью функции `std::transform` скопируем значения из первого вектора во второй. При копировании элементов преобразуем их благодаря вспомогательной функции нормализации. Минимальное и максимальное значения старого вектора равны `0` и `1000`. Минимальное и максимальное значения после нормализации равны `0` и `255`:

```
    transform(begin(v), end(v), back_inserter(v_norm),
              norm(*min_it, *max_it, 255));
```

8. Прежде чем реализовать вторую стратегию по нормализации, выведем полученный результат:

```
    copy(begin(v_norm), end(v_norm),
         ostream_iterator<int>{cout, ", "});
    cout << '\n';
```

9. Снова используем тот же нормализованный вектор, чтобы продемонстрировать работу другой вспомогательной функции `clampval`, которая *сжимает* старый диапазон к диапазону, в котором минимальное значение равно `0`, а максимальное — `255`:

```
transform(begin(v), end(v), begin(v_norm),
          clampval(0, 255));
```

10. После вывода этих значений на экран пример будет закончен:

```
    copy(begin(v_norm), end(v_norm),
         ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```

11. Скомпилируем и запустим программу. Поскольку значения элементов диапазона теперь попадают в диапазон от `0` до `255`, можно использовать их, например, как показатели яркости для цветовых кодов RGB:

```
$ ./reducing_range_in_vector
0, 255, 1, 63, 76, 204, 229, 81,
0, 255, 5, 250, 255, 255, 255, 255,
```

12. На основе полученных данных можно построить следующие графики (рис. 5.2). Как видите, подход, когда мы *делим* значения на разность между максимальным и минимальным значениями, является линейным преобразованием оригинальных данных. *Сжатый* график теряет некоторый объем информации. Обе вариации могут оказаться полезными в разных ситуациях.

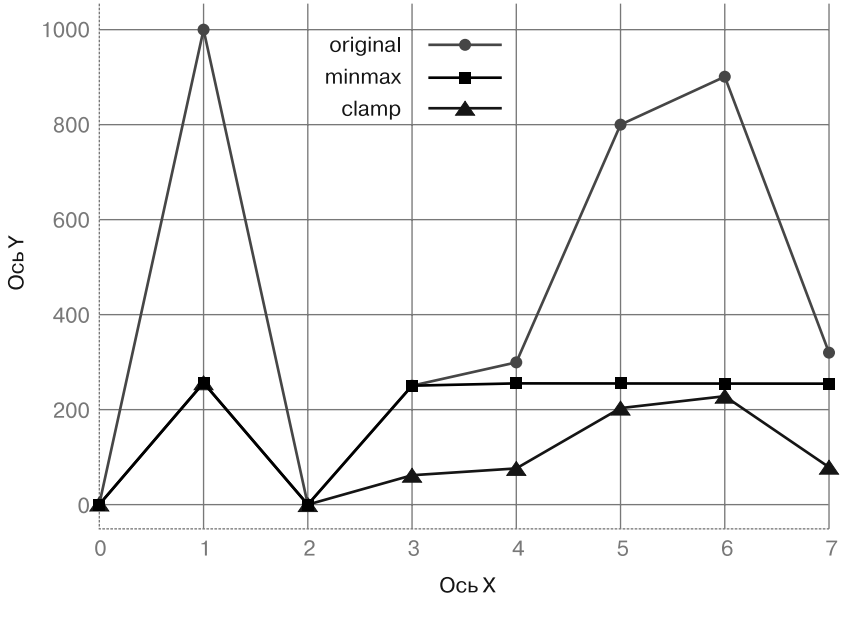

Рис. 5.2

Как это работает

Помимо `std::transform` мы использовали два алгоритма.

`std::minmax_element` принимает начальный и конечный итераторы входного диапазона. Он проходит в цикле по диапазону данных и записывает в процессе максимальное и минимальное значения. Эти значения возвращаются в виде пары, которую мы затем применяем для функции масштабирования.

Функция `std::clamp`, напротив, не работает с диапазоном данных. Она принимает три значения: входное, минимальное и максимальное. Результатом работы этой функции явится входное значение, обрезанное так, что будет находиться между указанными минимумом и максимумом. Кроме того, можно воспользоваться конструкцией `max(min_val, min(max_val, x))` вместо `std::clamp(x, min_val, max_val)`.

Находим шаблоны в строках с помощью функции std::search и выбираем оптимальную реализацию

Поиск строки — несколько иная задача, нежели поиск *одного* объекта в указанном диапазоне данных. С одной стороны, строка, конечно же, представляет собой итерабельный диапазон данных (состоящий из символов). С другой — поиск строки в строке означает поиск одного диапазона данных в *другом*. Для этого нам понадобится выполнить несколько сравнений для потенциальной позиции, содержащей совпадение, так что нужен какой-то другой алгоритм.

`std::string` уже содержит функцию `find`, которая решает именно эту задачу; тем не менее в этом разделе мы сконцентрируемся на функции `std::search`. Несмотря на то что она применяется по большей части для строк, ее можно использовать для контейнеров всех видов. Самая интересная особенность `std::search` заключается в том, что, начиная с C++17, у нее есть дополнительный интерфейс, позволяющий легко заменять алгоритм поиска. Эти алгоритмы оптимизированы, и пользователь может выбрать любой из них в зависимости от варианта применения. Вдобавок мы могли бы реализовать собственные алгоритмы поиска и подключить их в функцию `std::search`.

Как это делается

В этом примере мы воспользуемся новой функцией `std::search` для строк и опробуем ее разные вариации для объектов класса `searcher`.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

```
#include <iterator>
#include <functional>

using namespace std;
```

2. Мы будем выводить на экран подстроки из позиций, которые вернет алгоритм поиска, поэтому реализуем соответствующую вспомогательную функцию:

```
template <typename Itr>
static void print(Itr it, size_t chars)
{
    copy_n(it, chars, ostream_iterator<char>{cout});
    cout << '\n';
}
```

3. Строка *вида lorem-ipsum* сгодится в качестве примера, мы будем искать в ней подстроку. В нашем случае таковой выступит сочетание `"elitr"`:

```
int main()
{
    const string long_string {
        "Lorem ipsum dolor sit amet, consetetur"
        " sadipscing elitr, sed diam nonumy eirmod"};
    const string needle {"elitr"};
```

4. Старый интерфейс `std::search` принимает начальный/конечный итераторы для строки, внутри которой мы будем искать конкретную подстроку. Он возвращает итератор, указывающий на подстроку, которую смог найти. Если поиск не удался, то он вернет конечный итератор:

```
    {
        auto match (search(begin(long_string), end(long_string),
                           begin(needle), end(needle)));
        print(match, 5);
    }
```

5. Версия функции `std::search`, предлагаемая в C++17, принимает не две пары итераторов, а одну (начальный/конечный) и объект *поиска*. Объект `std::default_searcher` принимает пару итераторов для подстроки, которую мы ищем в более крупной строке:

```
    {
        auto match (search(begin(long_string), end(long_string),
            default_searcher(begin(needle), end(needle))));
        print(match, 5);
    }
```

6. Суть этого изменения заключается в том, что так проще сменить алгоритм поиска. Объект `std::boyer_moore_searcher` использует поисковый алгоритм Бойера — Мура, чтобы выполнять поиск несколько быстрее:

```
    {
        auto match (search(begin(long_string), end(long_string),
            boyer_moore_searcher(begin(needle),
```

```
                                    end(needle))));
        print(match, 5);
    }
```

7. В библиотеке STL версии C++17 появились три разные реализации поискового объекта. Третья реализация использует *алгоритм Бойера — Мура — Хорспула*.

```
    {
        auto match (search(begin(long_string), end(long_string),
            boyer_moore_horspool_searcher(begin(needle),
                                          end(needle))));
        print(match, 5);
    }
}
```

8. Скомпилируем и запустим программу. Если все работает правильно, то мы должны увидеть несколько одинаковых строк:

```
$ ./pattern_search_string
elitr
elitr
elitr
elitr
```

Как это работает

Мы воспользовались четырьмя разными способами применения функции `std::search`, чтобы получить одинаковые результаты. Какая функция больше подходит для конкретной ситуации?

Предположим, что наша большая строка, в которой мы будем искать шаблон, называется `s`, а сам шаблон — `p`. Тогда выражения `std::search(begin(s), end(s), begin(p), end(p));` и `std::search(begin(s), end(s), default_searcher(begin(p), end(p));` станут делать одно и то же.

Другие поисковые объекты функций реализуют более сложные алгоритмы поиска:

- `std::default_searcher` — выполняет переадресацию к устаревшей реализации `std::search`;
- `std::boyer_moore_searcher` — использует поисковый алгоритм *Бойера — Мура*;
- `std::boyer_moore_horspool_searcher` — по аналогии применяет алгоритм *Бойера — Мура — Хорспула*.

Что делает другие алгоритмы такими особенными? Алгоритм Бойера — Мура разрабатывали, опираясь на конкретную идею: шаблон поиска сравнивается со строкой, начиная с *конца* шаблона и двигаясь справа налево. Если символ в строке для поиска *отличается* от символа шаблона на перекрывающихся позициях и *даже не встречается* в шаблоне, то становится очевидно, что последний можно сместить в строке поиска на количество позиций, *равное его длине*. Взгляните на рис. 5.3, где это происходит в шаге 1. Если же символ, для которого в данный момент

выполняется сравнение, отличается от символа шаблона на этой позиции, но при этом *есть* в шаблоне, то алгоритм знает, на сколько символов нужно сместить шаблон вправо, чтобы правильно выровнять его относительно этого символа, а затем начинается новая итерация сравнения справа налево. На рис. 5.3 это происходит в шаге 2. Таким образом, алгоритм Бойера — Мура может опустить множество *ненужных* операций сравнения, в отличие от исходной реализации поиска.

Рис. 5.3

Конечно, этот алгоритм мог бы стать алгоритмом поиска по умолчанию, если бы вам не пришлось идти на *компромиссы*. Он работает быстрее алгоритма по умолчанию, но его следует использовать для структур данных быстрого поиска, чтобы определить, какие символы содержатся в шаблоне поиска и насколько они смещены. Компилятор выберет их сложную реализацию в зависимости от типов данных, хранящихся в шаблоне (они могут варьироваться от ассоциативных массивов, основанных на хеше, для сложных типов до примитивных справочных таблиц для типов наподобие `char`). В конечном счете это значит следующее: реализация поиска по умолчанию будет более быстрой при условии, что поисковая строка не слишком велика. Если сам поиск занимает много времени, то использование алгоритма Бойера — Мура может привести к повышению производительности в измерении *постоянного коэффициента*.

Алгоритм *Бойера — Мура — Хорспула* является упрощением описанного алгоритма Бойера — Мура. В нем нет *правила о плохом символе*, что приводит к сдвигу всей длины шаблона, если искомый символ не встречается в найденной строке. Компромисс в принятии такого решения заключается в том, что первый алгоритм работает *несколько медленнее*, чем второй, но в процессе ему требуется *меньше структур данных*.

Не пытайтесь определить, какой алгоритм будет работать быстрее в конкретных случаях. Всегда измеряйте производительность кода с помощью диапазонов данных, типичных для ваших пользователей, и основывайте свое решение на результатах этих измерений.

Делаем выборку данных из крупных векторов

При взаимодействии с очень большими объемами численных данных иногда возникают ситуации, когда их нужно обработать за оптимальное время. В одних случаях можно воспользоваться *выборкой*, чтобы снизить общий объем данных для дальнейшей обработки, что ускорит выполнение всей программы. В других это можно сделать с целью сократить объем операций не для обработки, а для сохранения или передачи данных.

Исходная реализация выборки представляет собой рассмотрение только *N-ных* точек. Это приемлемо в большинстве случаев, но для обработки сигналов, например, может привести к математическому феномену, который называется *искажением*. Если расстояние между пробными точками отличается на небольшую случайную величину, то искажение можно сократить. На рис. 5.4 продемонстрирован крайний случай только для того, чтобы показать эту точку: хотя исходный сигнал выражается в виде синусоидальной волны, треугольные точки на графике являются пробными и взяты с одинаковым шагом *100*. К сожалению, сигнал имеет одинаковое значение по оси *Y* во всех точках! График, получаемый путем соединения этих точек, выглядит как идеальная прямая *горизонтальная линия*. Квадратные точки, однако, показывают, что мы получим, если будем брать образец каждые `100 + random(-15, +15)` точек. Полученный сигнал все еще значительно отличается от оригинального, но уже не пропадает полностью, как это было в случае с выборкой с фиксированным шагом.

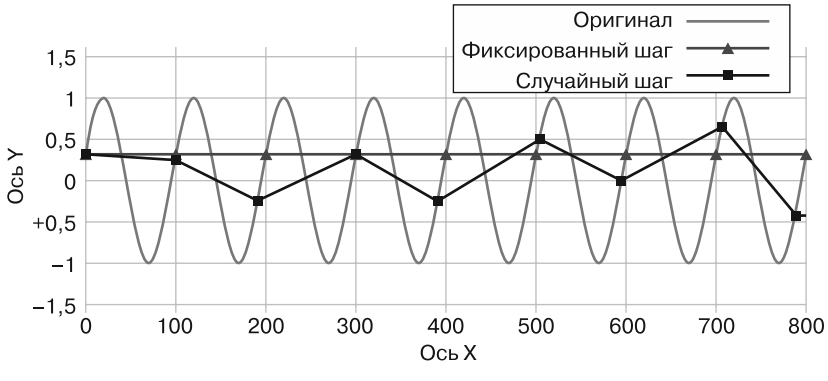

Рис. 5.4

Функция `std::sample` не добавляет случайные значения к пробным точкам с фиксированным шагом, она выбирает полностью случайные точки; поэтому в рамках данного примера работает несколько по-другому.

Как это делается

В этом примере мы создадим выборку для очень большого вектора, содержащую случайные данные. Они имеют нормальное распределение. После завершения выборки полученные результаты все еще будут показывать нормальное распределение, что мы и проверим.

1. Сначала включим все заголовочные файлы, которые будем применять, а также объявим об использовании пространства имен `std`, чтобы сэкономить немного времени на вводе текста:

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>
#include <iterator>
#include <map>
#include <iomanip>

using namespace std;
```

2. С кодом работать будет гораздо проще, если мы сконфигурируем конкретные характеристики нашего алгоритма с помощью константных переменных. Укажем размер большого вектора случайных чисел, а также количество точек:

```
int main()
{
    const size_t data_points   {100000};
    const size_t sample_points {100};
```

3. Большой вектор, заполненный случайными числами, должен наполняться с помощью генератора случайных чисел, выдающего числа, которые будут иметь нормальное распределение. Любое нормальное распределение характеризуется математическим ожиданием и квадратическим отклонением:

```
    const int    mean {10};
    const size_t dev  {3};
```

4. Теперь настроим генератор случайных чисел. Сначала создадим экземпляр `random_device` и вызовем его один раз, чтобы получить исходное значение для конструктора генератора случайных чисел. Далее создадим объект распределения, который применяет нормальное распределение к полученным случайным числам:

```
    random_device rd;
    mt19937 gen {rd()};
    normal_distribution<> d {mean, dev};
```

5. Создадим экземпляр вектора, содержащего целые числа, и заполним его случайными числами. Это можно сделать с помощью алгоритма `std::generate_n`, который вызовет объект функции генератора, чтобы передать его возвращаемое значение в наш вектор благодаря итератору `back_inserter`. Объект функции генератора оборачивает выражение `d(gen)`, получающее случайное число от случайного устройства и передает его в объект распределения:

```
    vector<int> v;
    v.reserve(data_points);
    generate_n(back_inserter(v), data_points,
        [&] { return d(gen); });
```

6. Теперь создадим еще один вектор, который содержит гораздо меньший диапазон точек:

```
vector<int> samples;
v.reserve(sample_points);
```

7. Алгоритм `std::sample` работает аналогично алгоритму `std::copy`, но при этом принимает два дополнительных параметра — *количество точек*, получаемых из входного диапазона данных, и объект *генератора случайных чисел*, к которому он будет обращаться, чтобы получить случайные позиции пробных точек:

```
sample(begin(v), end(v), back_inserter(samples),
       sample_points, mt19937{random_device{}()});
```

8. С выборкой мы закончили. Остальная часть кода посвящена отображению данных. Входные данные имеют нормальное распределение, и если алгоритм выборки отработал хорошо, то полученный вектор тоже должен иметь такое же распределение. Чтобы увидеть, насколько распределение отклоняется от нормального, выведем на экран *гистограмму* значений:

```
map<int, size_t> hist;

for (int i : samples) { ++hist[i]; }
```

9. Наконец, пройдем в цикле по всем элементам, чтобы вывести нашу гистограмму на экран:

```
for (const auto &[value, count] : hist) {
    cout << setw(2) << value << " "
         << string(count, '*') << '\n';
}
}
```

10. После компиляции и запуска программы мы видим, что полученный вектор имеет характеристики нормального распределения (рис. 5.5):

```
$ ./sampling_vectors
 1 *
 3 *
 4 **
 5 *****
 6 *****
 7 *******
 8 **************
 9 **************
10 ******************
11 *********
12 ********
13 *******
14 ***
15 ***
16 ***
```

Рис. 5.5

Как это работает

Алгоритм `std::sample` — это новый алгоритм, который появился в версии C++17. Его сигнатура выглядит следующим образом:

```
template<class InIterator, class OutIterator,
         class Distance, class UniformRandomBitGenerator>
OutIterator sample(InIterator first, InIterator last,
                   SampleIterator out, Distance n,
                   UniformRandomBitGenerator&& g);
```

Входной диапазон данных обозначается итераторами `first` и `last`, в то время как `out` — итератор вывода. Эти итераторы выполняют ту же функцию, что и для функции `std::copy`; элементы копируются из одного диапазона в другой. Алгоритм `std::sample` является особенным с той точки зрения, что копирует только часть входного диапазона, поскольку делает выборку только `n` элементов. Он использует равномерное распределение внутри системы, поэтому каждая точка на графике во входном диапазоне данных может быть выбрана с одинаковой вероятностью.

Выполняем перестановки во входных последовательностях

При тестировании кода, который должен работать с последовательностями входных данных, где порядок аргументов не так важен, полезно проверять, получаем ли мы одинаковый результат во *всех* возможных перестановках для этих входных данных. Такой тест может, например, проверять, корректно ли работает ваш собственный алгоритм *сортировки*.

Независимо от того, зачем нужны все перестановки для какого-то диапазона значений, `std::next_permutation` позволит это реализовать. Можно вызвать эту функцию для изменяемого диапазона, и она изменит *порядок* его элементов на следующую *лексикографическую перестановку*.

Как это делается

В данном примере мы напишем программу, которая считывает несколько строк, содержащих слова, из стандартного потока ввода, а затем применим функцию `std::next_permutation`, чтобы сгенерировать и вывести на экран все перестановки для этих строк.

1. Как и обычно, сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

```
#include <iterator>
#include <algorithm>

using namespace std;
```

2. Мы начнем с вектора строк, который заполним из стандартного потока ввода. Затем *отсортируем* вектор:

```
int main()
{
    vector<string> v {istream_iterator<string>{cin}, {}};
    sort(begin(v), end(v));
```

3. Далее выведем содержимое вектора на консоль. Затем вызовем функцию `std::next_permutation`. Она систематически перемешивает содержимое вектора, чтобы сгенерировать перестановки этих элементов, которое мы снова выведем на экран. Вызов `next_permutation` вернет значение `false`, когда будет сгенерирована *последняя* перестановка.

```
    do {
        copy(begin(v), end(v),
             ostream_iterator<string>{cout, ", "});
        cout << '\n';
    } while (next_permutation(begin(v), end(v)));
}
```

4. Скомпилируем и запустим функцию, передав ей какие-нибудь данные:

```
$ echo "a b c" | ./input_permutations
a, b, c,
a, c, b,
b, a, c,
b, c, a,
c, a, b,
c, b, a,
```

Как это работает

Алгоритм `std::next_permutation` выглядит несколько странным. Это потому, что он принимает только пару итераторов (начальный и конечный) и возвращает значение `true`, если может найти следующую перестановку. В противном случае он возвращает значение `false`. Но что вообще означает выражение *следующая перестановка*?

Алгоритм, с помощью которого функция `std::next_permutation` находит следующий лексикографический порядок элементов, работает таким образом.

1. Найдем самое большое значение индекса `i`, для которого выполняется условие `v[i - 1] < v[i]`. Если такого значения нет, то вернем значение `false`.
2. Теперь найдем самое большое значение индекса `j`, для которого выполняются условия `j >= i` и `v[j] > v[i - 1]`.

3. *Меняем местами* элементы на позициях `j` и `i - 1`.
4. Изменим порядок элементов, находящихся между позицией `i` и концом диапазона данных, на противоположный.
5. Вернем значение `true`.

Отдельные порядки перестановки, которые мы получим в результате вызова этой функции, всегда будут появляться в одинаковой последовательности. Чтобы найти все возможные перестановки, мы поначалу отсортировали массив. Это было сделано потому, что если бы ввели строку `"c b a"`, например, то алгоритм завершил бы работу *немедленно*, так как данная строка представляет собой последний возможный лексикографический порядок элементов.

Инструмент для слияния словарей

Представьте, что у вас имеется отсортированный список элементов, у кого-то другого появляется *другой* такой же список и вы хотите обменяться этими списками друг с другом. Самая лучшая идея заключается в том, чтобы объединить оба списка. Их комбинацию нужно отсортировать, поскольку так будет проще найти конкретные элементы.

Данная операция также называется *слиянием*. Чтобы слить воедино два отсортированных диапазона данных, можно создать новый диапазон и заполнить его элементами из обоих списков. Для каждого перемещения элемента нужно сравнить два элемента наших входных диапазонов данных с целью выбрать *самый маленький* из оставшихся. В противном случае выходной диапазон не будет отсортирован. Этот процесс показан на рис. 5.6.

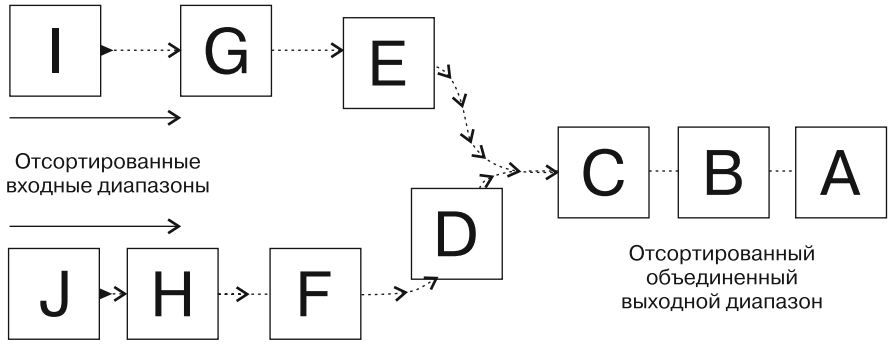

Рис. 5.6

Алгоритм `std::merge` позволяет решить именно эту задачу, поэтому не придется тратить время зря. В данном разделе мы увидим, как пользоваться этим алгоритмом.

Как это делается

В этом примере мы создадим небольшой словарь, в котором английские слова соотносятся с их переводом на немецкий язык, и поместим их в структуры `std::deque`. Программа считает словари из файла и стандартного потока ввода, а затем выведет один большой объединенный словарь в стандартный поток вывода.

1. В этот раз мы включим очень много заголовочных файлов, а также объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <deque>
#include <tuple>
#include <string>
#include <fstream>

using namespace std;
```

2. Запись словаря должна состоять из симметричной пары, содержащей строки на обоих языках.

```
using dict_entry = pair<string, string>;
```

3. Выведем данные пары на консоль и считаем их с пользовательского ввода, поэтому нужно перегрузить операторы `<<` и `>>`:

```
namespace std {
ostream& operator<<(ostream &os, const dict_entry p)
{
    return os << p.first << " " << p.second;
}
istream& operator>>(istream &is, dict_entry &p)
{
    return is >> p.first >> p.second;
}
}
```

4. Вспомогательная функция, принимающая любые объекты потока ввода, поможет создать словарь. Она встраивает экземпляр контейнера `std::deque`, состоящий из записей словаря, которые будут считываться из потока ввода до тех пор, пока тот не опустеет. Прежде чем вернуть контейнер, отсортируем его:

```
template <typename IS>
deque<dict_entry> from_instream(IS &&is)
{
    deque<dict_entry> d {istream_iterator<dict_entry>{is}, {}};
    sort(begin(d), end(d));
    return d;
}
```

5. Создадим два отдельных словаря на основе разных потоков ввода. Один будет получен из файла `dict.txt` — мы предполагаем, что он существует. Он содержит пары слов, размещенные построчно. Другой поток — это стандартный поток ввода.

```
int main()
{
    const auto dict1 (from_instream(ifstream{"dict.txt"}));
    const auto dict2 (from_instream(cin));
```

6. С помощью вспомогательной функции `from_instream` мы уже отсортировали оба словаря и теперь можем передать их алгоритму `std::merge`. Он принимает два входных диапазона данных, определенных с помощью пар итераторов ввода/вывода. Вывод будет показан на консоли пользователя.

```
    merge(begin(dict1), end(dict1),
          begin(dict2), end(dict2),
          ostream_iterator<dict_entry>{cout, "\n"});
}
```

7. Мы уже можем скомпилировать программу, но перед запуском следует создать файл `dict.txt`, содержащий какие-нибудь строки. Заполним его английскими словами и их переводом на немецкий язык:

```
car       auto
cellphone handy
house     haus
```

8. Теперь запустим программу и передадим в стандартный поток ввода пары английских и немецких слов. В результате мы получим объединенный и отсортированный словарь, который содержит данные из обоих источников. Можно создать для него новый файл словаря.

```
$ echo "table tisch fish fisch dog hund" | ./dictionary_merge
car auto
cellphone handy
dog hund
fish fisch
house haus
table tisch
```

Как это работает

Алгоритм `std::merge` принимает две пары начальных и конечных итераторов, обозначающие входные диапазоны данных. Эти диапазоны должны быть *отсортированы*. Пятый параметр — итератор вывода, который принимает поступающие элементы во время слияния.

Кроме того, существует вариант алгоритма, который называется `std::inplace_merge`. Он делает то же самое, что и предыдущий, но не нуждается в итераторе

вывода, поскольку работает *на месте*, о чем можно догадаться из имени. Он принимает три параметра: *начальный* итератор, итератор *для середины* и *конечный* итератор. Эти итераторы должны ссылаться на данные в одной структуре. Итератор для середины является одновременно конечным итератором для первого диапазона данных и начальным итератором для второго диапазона. Это значит, что алгоритм работает с одним диапазоном данных, который на самом деле состоит из двух диапазонов, размещенных один за другим, например `{A, C, B, D}`. Первый поддиапазон — `{A, C}`, а второй — `{B, D}`. Алгоритм `std::inplace_merge` может объединить оба диапазона в одной структуре данных, что приведет к результату `{A, B, C, D}`.

6 Сложные случаи использования алгоритмов STL

В этой главе:

- реализация класса префиксного дерева с использованием алгоритмов STL;
- реализация генератора подсказок при поиске с помощью префиксных деревьев;
- реализация формулы преобразования Фурье с применением численных алгоритмов STL;
- определение ошибки суммы двух векторов;
- реализация отрисовщика множества Мандельброта в ASCII;
- создание собственного алгоритма `split`;
- создание полезных алгоритмов на основе стандартных — `gather`;
- удаление лишних пробельных символов между словами;
- компрессия и декомпрессия строк.

Введение

В предыдущей главе мы рассмотрели базовые алгоритмы STL и выполнили с их помощью простые задания, чтобы понять, как работать с типичным интерфейсом библиотеки: большая часть ее алгоритмов в качестве входных и выходных параметров принимает один или более диапазонов данных в виде пар итераторов. Они зачастую также принимают функции-предикаты, пользовательские функции сравнения или же функции преобразования. В конечном счете они в основном возвращают итераторы, поскольку их можно передать другим алгоритмам.

Хотя программисты стремятся делать алгоритмы STL минимального размера, в то же время интерфейсы они стараются разрабатывать максимально обобщенными. Это позволяет использовать код повторно, но он не всегда хорошо выглядит. Опытный разработчик C++, знающий все алгоритмы, быстрее прочитает код

других людей, если они пытались выразить большинство своих идей с помощью алгоритмов STL. Мозг программиста скорее проанализирует название хорошо известного алгоритма, чем поймет сложный цикл, выполняющий ту же задачу несколько иным образом.

К этому моменту вы уже научились использовать структуры данных STL настолько интуитивно, что можете обходиться без указателей, необработанных массивов и других устаревших структур. Следующим шагом будет более глубокое изучение алгоритмов STL, чтобы вы поняли, как обойтись без сложных циклов, выражая их в терминах популярных алгоритмов STL. Это позволит значительно повысить ваш уровень, поскольку код станет более коротким, удобочитаемым и обобщенным, а также не будет привязан к структурам данных. Вы практически всегда можете избежать написания циклов вручную и взять код алгоритма из пространства имен и `std`, но иногда это приводит к тому, что ваш код начинает выглядеть *странно*. Мы не станем разбираться, какой код выглядит странно, а какой — нет, просто рассмотрим возможные варианты.

В этой главе мы применим алгоритмы STL необычным способом, чтобы исследовать новые горизонты и увидеть, как решать отдельные задачи с помощью современного C++. Кроме того, мы реализуем собственные алгоритмы, которые можно будет легко объединить с существующими структурами данных и другими алгоритмами, разработанными аналогичным способом. Затем мы *объединим* имеющиеся алгоритмы STL, чтобы получить *новые* алгоритмы, которых еще не существует. Такие объединенные алгоритмы позволяют создавать более сложные алгоритмы, но при этом они остаются относительно короткими и читабельными. Еще мы узнаем, почему именно алгоритмы STL считаются «аккуратными» и пригодными для многократного использования. Мы сможем принимать наилучшие решения, только рассмотрев *все* варианты.

Реализуем класс префиксного дерева с использованием алгоритмов STL

Так называемая структура данных префиксного *дерева* представляет собой интересный способ хранить данные так, чтобы по ним было легко выполнить поиск. При разбиении предложений на списки слов вы зачастую можете объединить первые несколько слов, одинаковых в каждом предложении.

Взглянем на рис. 6.1, где предложения `hi how are you` и `hi how do you do` сохранены в древоподобной структуре. В этом случае одинаковыми являются слова `hi how`, а затем предложения различаются и разветвляются, как дерево.

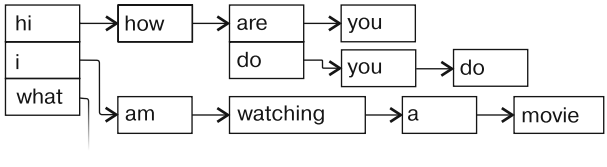

Рис. 6.1

Поскольку структура данных префиксного дерева объединяет общие префиксы, она также называется *деревом префиксов*. Такую структуру нетрудно реализовать с помощью средств, предлагаемых библиотекой STL. Этот раздел посвящен реализации собственного класса префиксного дерева.

Как это делается

В данном примере мы реализуем собственное дерево префиксов с помощью структур данных и алгоритмов, предлагаемых в библиотеке STL.

1. Включим все заголовочные файлы применяемых частей библиотеки STL, а также объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <optional>
#include <algorithm>
#include <functional>
#include <iterator>
#include <map>
#include <vector>
#include <string>

using namespace std;
```

2. Вся программа посвящена префиксному дереву, для которого нужно реализовать собственный класс. В нашей реализации данное дерево, по сути, является рекурсивным ассоциативным массивом, содержащим ассоциативные массивы. Каждый узел дерева содержит подобный массив, в котором соотносятся объект, имеющий тип `T`, и следующий узел префиксного дерева:

```
template <typename T>
class trie
{
    map<T, trie> tries;
```

3. Код для добавления новых последовательностей элементов выглядит довольно просто. Пользователь предоставляет пару итераторов (начальный и конечный), и мы проходим по ним рекурсивно. Если он ввел данные `{1, 2, 3}`, то мы ищем значение `1` в поддереве, а затем ищем значение `2` в следующем поддереве, чтобы получить поддерево для значения `3`. Какое-то из этих поддеревьев, ранее не существовавшее, будет добавлено с помощью оператора `[]` контейнера `std::map`.

```
public:
    template <typename It>
    void insert(It it, It end_it) {
        if (it == end_it) { return; }
        tries[*it].insert(next(it), end_it);
    }
```

4. Для удобства определим также отдельные функции, они дают пользователю возможность получить контейнер элементов, которые будут автоматически опрошены на предмет итераторов:

```
template <typename C>
void insert(const C &container) {
    insert(begin(container), end(container));
}
```

5. Чтобы позволить пользователю написать конструкцию `my_trie.insert({"a", "b", "c"});`, мы должны немного помочь компилятору вывести все типы из данной строки, поэтому добавляем функцию, которая перегружает интерфейс `insert` параметром типа `initializer_list`:

```
void insert(const initializer_list<T> &il) {
    insert(begin(il), end(il));
}
```

6. Мы также хотим видеть содержимое дерева, поэтому нужна функция `print`. Для вывода содержимого дерева на экран можно выполнить поиск в глубину. На пути от корневого узла к первому листу мы записываем все элементы с полезной нагрузкой, которые мы уже встречали. Таким образом, мы получим полную последовательность, как только достигнем листа, и вывести ее на экран будет нетрудно. Мы видим, что достигли листа, когда функция `tries.empty()` возвращает значение `true`. После рекурсивного вызова функции `print` мы снова выталкиваем последний элемент с полезной нагрузкой.

```
void print(vector<T> &v) const {
    if (tries.empty()) {
        copy(begin(v), end(v),
             ostream_iterator<T>{cout, " "});
        cout << '\n';
    }
    for (const auto &p : tries) {
        v.push_back(p.first);
        p.second.print(v);
        v.pop_back();
    }
}
```

7. Рекурсивная функция `print` передает ссылку на выводимый на экран список элементов, но пользователь должен вызывать ее без всяких параметров. Поэтому определяем функцию `print` без параметров, в которой создается вспомогательный объект списка:

```
void print() const {
    vector<T> v;
    print(v);
}
```

8. Теперь, когда мы научились создавать деревья и выводить их на экран, может понадобиться выполнить поиск по поддеревьям. Идея заключается в следующем: если дерево содержит последовательности наподобие `{a, b, c}` `{a, b, d, e}` и мы передаем ему последовательность `{a, b}` для поиска, то поиск вернет поддерево, которое содержит части `{c}` и `{d, e}`. При обнаружении поддерева возвращаем ссылку `const` на нее. Существует вероятность того, что такого поддерева не существует, если дерево не содержит искомой последовательности. В подобных случаях все равно нужно *что-то* вернуть. Здесь пригодится функция `std::optional`, поскольку можно вернуть *пустой* необязательный объект при отсутствии совпадения.

```
template <typename It>
optional<reference_wrapper<const trie>>
subtrie(It it, It end_it) const {
    if (it == end_it) { return ref(*this); }
    auto found (tries.find(*it));
    if (found == end(tries)) { return {}; }
    return found->second.subtrie(next(it), end_it);
}
```

9. Аналогично методу `insert` предоставляем версию метода `subtrie` с одним параметром, которая автоматически принимает итераторы из входного контейнера:

```
    template <typename C>
    auto subtrie(const C &c) {
        return subtrie(begin(c), end(c));
    }
};
```

10. На этом все. Воспользуемся нашим новым классом `trie` в функции `main`, создав экземпляр класса `trie`, работающего с объектами класса `std::string`, и заполнив его каким-нибудь содержимым:

```
int main()
{
    trie<string> t;
    t.insert({"hi", "how", "are", "you"});
    t.insert({"hi", "i", "am", "great", "thanks"});
    t.insert({"what", "are", "you", "doing"});
    t.insert({"i", "am", "watching", "a", "movie"});
```

11. Сначала выведем на экран все дерево:

```
    cout << "recorded sentences:\n";
    t.print();
```

12. Затем получим поддерево для всех входных предложений, которые начинаются со слова `"hi"`, и выведем их на экран:

```
    cout << "\npossible suggestions after \"hi\":\n";
    if (auto st (t.subtrie(initializer_list<string>{"hi"}));
        st) {
```

```
            st->get().print();
        }
    }
```

13. Компиляция и запуск программы покажет, что мы действительно получим всего два предложения, начинающихся со слова `"hi "`, если запросим именно это поддерево:

```
$ ./trie
recorded sentences:
hi how are you
hi i am great thanks
i am watching a movie
what are you doing
possible suggestions after "hi":
how are you
i am great thanks
```

Как это работает

Что интересно, код для *вставки* последовательности слов выглядит короче и проще, чем код *поиска* заданного слова в поддереве. Поэтому сначала рассмотрим код вставки:

```
template <typename It>
void trie::insert(It it, It end_it) {
    if (it == end_it) { return; }
    tries[*it].insert(next(it), end_it);
}
```

Пара итераторов `it` и `end_it` представляет собой вставляемую последовательность слов. Элемент `tries[*it]` выполняет поиск первого слова последовательности в поддереве, а затем с помощью конструкции `.insert(next(it), end_it)` мы перезапускаем ту же функцию для найденного нижнего поддерева, переместив итератор на одно слово *вперед*. Строка `if (it == end_it) { return; }` нужна для прерывания рекурсии. Пустое выражение `return` не делает *ничего*, что на первый взгляд кажется странным. Все операции вставки выполняются с привлечением выражения `tries[*it]`. Оператор `[]` контейнера `std::map` либо возвращает существующий элемент для заданного ключа, либо *создает* элемент с таким ключом. Связанное значение (соотнесенным типом в нашем примере является тип `trie`) создается с помощью конструктора по умолчанию. Таким образом, при поиске неизвестных слов мы *неявно создаем* новую ветвь дерева.

Поиск в поддереве выглядит более сложным, поскольку мы не можем *выразить* многие функции неявно:

```
template <typename It>
optional<reference_wrapper<const trie>>
subtrie(It it, It end_it) const {
    if (it == end_it) { return ref(*this); }
```

```
    auto found (tries.find(*it));
    if (found == end(tries)) { return {}; }

    return found->second.subtrie(next(it), end_it);
}
```

Данный код, по сути, строится вокруг выражения `auto found (tries.find(*it));`. Вместо того чтобы искать следующий по глубине узел дерева с помощью оператора (`[]`), мы применяем метод `find`. Если бы мы использовали для поиска оператор `[]`, то дерево *создавало бы* отсутствующие элементы — совсем *не то*, что нужно! (Кстати, попробуйте сделать это. Метод класса является константным, вследствие чего такой подход невозможен. Это поможет вам избежать некоторых ошибок.)

Еще одной непонятной деталью является возвращаемый тип `optional<reference_wrapper<const trie>>`. В качестве оболочки мы выбрали `std::optional`, поскольку есть вероятность, что такого поддерева во входной последовательности нет. Если мы передаем только последовательность `"hello my friend"`, то последовательность `"goodbye my friend"` не будет существовать. В таких случаях мы просто возвращаем `{}`, что передает вызывающей стороне пустой необязательный объект. Это все еще не объясняет, почему мы используем `reference_wrapper` вместо `optional<const trie &>`. Идея заключается в том, что необязательному экземпляру, имеющему переменную-член с типом `trie&`, нельзя повторно присвоить значение и поэтому код не скомпилируется. Реализация ссылки с помощью `reference_wrapper` приведет к тому, что у вас появится возможность присваивать значения объектам повторно.

Создаем генератор поисковых подсказок с помощью префиксных деревьев

Когда вы вводите некие символы в поисковик, интерфейс зачастую пытается определить, как будет выглядеть весь поисковый запрос. Эти догадки зачастую основываются на популярных запросах. Иногда подобные догадки выглядят довольно забавно, поскольку оказывается, что люди вводят в поисковик странные запросы (рис. 6.2).

Рис. 6.2

В этом разделе мы используем класс `trie`, который реализовали в предыдущем примере, и создадим небольшой генератор подсказок, всплывающих при поиске.

Как это делается

В данном примере мы реализуем консольное приложение, которое принимает некие входные данные, а затем пробует определить, что именно пользователь хочет найти, основываясь на небольшой текстовой базе данных.

1. Как и всегда, сначала указываем, что включаем заголовочные файлы, а также объявляем об использовании пространства имен `std`:

```
#include <iostream>
#include <optional>
#include <algorithm>
#include <functional>
#include <iterator>
#include <map>
#include <list>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;
```

2. Воспользуемся реализацией из предыдущего примера:

```
template <typename T>
class trie
{
    map<T, trie> tries;
public:
    template <typename It>
    void insert(It it, It end_it) {
        if (it == end_it) { return; }
        tries[*it].insert(next(it), end_it);
    }
    template <typename C>
    void insert(const C &container) {
        insert(begin(container), end(container));
    }
    void insert(const initializer_list<T> &il) {
        insert(begin(il), end(il));
    }
    void print(list<T> &l) const {
        if (tries.empty()) {
            copy(begin(l), end(l),
                 ostream_iterator<T>{cout, " "});
            cout << '\n';
        }
        for (const auto &p : tries) {
            l.push_back(p.first);
            p.second.print(l);
            l.pop_back();
        }
```

```
    }
    void print() const {
        list<T> l;
        print(l);
    }
    template <typename It>
    optional<reference_wrapper<const trie>>
    subtrie(It it, It end_it) const {
        if (it == end_it) { return ref(*this); }
        auto found (tries.find(*it));
        if (found == end(tries)) { return {}; }
      return found->second.subtrie(next(it), end_it);
     }
    template <typename C>
    auto subtrie(const C &c) const {
        return subtrie(begin(c), end(c));
    }
};
```

3. Добавим небольшую вспомогательную функцию, которая выводит на экран строку, приглашающую пользователя ввести какой-нибудь текст:

```
static void prompt()
{
    cout << "Next input please:\n > ";
}
```

4. В функции `main` открываем текстовый файл, который играет роль нашей текстовой базы данных. Мы считываем данный файл строка за строкой и передаем эти строки в дерево:

```
int main()
{
    trie<string> t;
    fstream infile {"db.txt"};
    for (string line; getline(infile, line);) {
        istringstream iss {line};
        t.insert(istream_iterator<string>{iss}, {});
    }
```

5. Теперь, когда мы создали дерево на основе содержимого текстового файла, нужно реализовать интерфейс, который позволит пользователю отправлять запросы. Приглашаем пользователя ввести какой-нибудь текст и ожидаем его действий:

```
    prompt();
    for (string line; getline(cin, line);) {
        istringstream iss {line};
```

6. Имея введенный текст, мы делаем запрос к дереву, чтобы получить поддерево. Если у нас есть такая входная последовательность в текстовом файле, то можем вывести на экран возможное продолжение поискового запроса, как делают дру-

гие поисковики. При отсутствии соответствующего поддерева просто скажем об этом пользователю:

```
        if (auto st (t.subtrie(istream_iterator<string>{iss}, {}));
            st) {
            cout << "Suggestions:\n";
            st->get().print();
        } else {
            cout << "No suggestions found.\n";
        }
```

7. Затем снова выведем текст приглашения и подождем следующей строки от пользователя. На этом все.

```
        cout << "---------------\n";
        prompt();
    }
}
```

8. Прежде чем запустить программу, следует чем-то заполнить файл `db.txt`. В нем может быть все что угодно, его даже не нужно сортировать. Каждая строка текста будет одной последовательностью дерева.

```
do ghosts exist
do goldfish sleep
do guinea pigs bite
how wrong can you be
how could trump become president
how could this happen to me
how did bruce lee die
how did you learn c++
what would aliens look like
what would macgiver do
what would bjarne stroustrup do
...
```

9. До запуска программы нужно создать файл `db.txt`. Его содержимое может выглядеть следующим образом:

```
hi how are you
hi i am great thanks
do ghosts exist
do goldfish sleep
do guinea pigs bite
how wrong can you be
how could trump become president
how could this happen to me
how did bruce lee die
how did you learn c++
what would aliens look like
what would macgiver do
what would bjarne stroustrup do
what would chuck norris do
```

```
why do cats like boxes
why does it rain
why is the sky blue
why do cats hate water
why do cats hate dogs
why is c++ so hard
```

10. Компиляция и запуск программы с последующим вводом некоторых входных данных будут выглядеть так:

```
$ ./word_suggestion
Next input please:
 > what would
Suggestions:
aliens look like
bjarne stroustrup do
chuck norris do
macgiver do
----------------
Next input please:
 > why do
Suggestions:
cats hate dogs
cats hate water
cats like boxes
----------------
Next input please:
 >
```

Как это работает

Принцип работы префиксного дерева был показан в предыдущем примере, но способ его заполнения и создания запросов к нему выглядит несколько странно. Давайте подробнее рассмотрим фрагмент кода, который заполняет пустое дерево на основе содержимого текстовой базы данных:

```
fstream infile {"db.txt"};
for (string line; getline(infile, line);) {
    istringstream iss {line};
    t.insert(istream_iterator<string>{iss}, {});
}
```

Цикл заполняет строку `line` содержимым текстового файла строка за строкой. Затем мы копируем строку в объект `istringstream`. Из этого объекта можно создать итератор `istream_iterator`, который нам еще пригодится, поскольку наше дерево принимает не только экземпляр контейнера для поиска поддеревьев, но и итераторы. Таким образом, нет нужды создавать вектор или список слов и можно непосредственно принять строку. Избежать последней части ненужных выделений памяти позволит *перемещение* содержимого строки в `iss`. К сожале-

нию, объект типа `std::istringstream` не предоставляет конструктор, который принимает *перемещаемые* значения типа `std::string`. Тем не менее он *скопирует* свою входную строку.

При чтении пользовательских входных данных, которые нужно найти в дереве, мы применяем точно такую же стратегию, но не задействуем *файловый* поток ввода. Вместо этого мы прибегаем к `std::cin`. В нашем случае это сработает аналогично, поскольку `trie::subtrie` работает для итераторов точно так же, как и `trie::insert`.

Дополнительная информация

Можно добавить в каждый узел дерева *переменные-счетчики*. Это позволит определить, *как часто* префикс встречается в некоторых входных данных. На основе этой информации можно *отсортировать* предположения по частоте их встречаемости, что и делают поисковики. Подсказки, возникающие при наборе текста на смартфоне, также реализованы подобным образом.

Предлагаю вам создать этот вариант генератора подсказок в качестве самостоятельного упражнения.

Реализуем формулу преобразования Фурье с применением численных алгоритмов STL

Преобразование Фурье — это очень важная и очень известная формула в области обработки сигналов. Она была открыта примерно 200 лет назад, но с появлением компьютеров количество вариантов ее использования значительно увеличилось. Она применяется при сжатии аудио/изображений/видео, в аудиофильтрах, медицинских устройствах отображения, приложениях для телефонов, которые определяют, какая песня сейчас играет, и т. д.

Поскольку количество возможных вариантов применения данной формулы довольно велико (и не только преобразования Фурье), STL разрабатывалась так, чтобы приносить пользу и при выполнении вычислений. Преобразование Фурье — лишь один из многих примеров подобных вычислений, при этом не самый простой. Сама формула выглядит так (рис. 6.3):

$$\hat{s}_k = \sum_{j=0}^{N-1} s_j \cdot e^{-i2\pi\frac{kj}{N}}$$

Рис. 6.3

Преобразование, описываемое этой формулой, по сути, описывает *сумму*. Каждый элемент суммы является произведением точки графика вектора входного сигнала и выражения `exp(-2 * i * ...)`. Вычисления, стоящие за данной

формулой, могут озадачить всех, кто незнаком с комплексными числами (и тех, кому просто не нравится математика), но *освоить* пример можно и не понимая эту формулу полностью. Если взглянуть на нее поближе, то увидим, что все точки графика сигнала (`N` элементов) суммируются с помощью переменной цикла `j`. Переменная `k` — еще одна переменная цикла, поскольку преобразование Фурье вычисляет не одно значение, а целый вектор. В данном векторе каждая точка графика представляет собой интенсивность и фазу определенной частоты повторяющейся волны. При реализации этой формулы с помощью циклов получится примерно такой код:

```
csignal fourier_transform(const csignal &s) {
    csignal t(s.size());
    const double pol {-2.0 * M_PI / s.size()};

    for (size_t k {0}; k < s.size(); ++k) {
        for (size_t j {0}; j < s.size(); ++j) {
            t[k] += s[j] * polar(1.0, pol * k * j);
        }
    }
    return t;
}
```

Тип `csignal` может быть вектором, содержащим комплексные числа. Для работы с такими числами в STL существует класс `std::complex`. Функция `std::polar`, по сути, выполняет часть *exp(–i * 2 * ...)*.

Эта версия работает хорошо, но давайте реализуем преобразование Фурье с помощью инструментов, доступных в STL.

Как это делается

В данном примере мы реализуем преобразование Фурье, а затем трансформируем с его помощью некоторые сигналы.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <complex>
#include <vector>
#include <algorithm>
#include <iterator>
#include <numeric>
#include <valarray>
#include <cmath>

using namespace std;
```

2. Точка графика сигнала представляет собой комплексное число, которое будет выражено с помощью типажа `std::complex`, специализированного для типа

`double`. Таким образом, псевдоним типа `cmplx` будет расшифровываться в виде двух связанных значений типа `double`, которые представляют *действительную* и *мнимую* части комплексного числа. Весь сигнал представляет собой вектор, содержащий подобные элементы; назовем этот тип `csignal`:

```
using cmplx   = complex<double>;
using csignal = vector<cmplx>;
```

3. Чтобы проитерировать по возрастающей численной последовательности, мы возьмем *численный итератор* из соответствующего примера. Переменные `k` и `j` и формулы преобразования будут итерировать по подобным последовательностям.

```
class num_iterator {
    size_t i;
public:
    explicit num_iterator(size_t position) : i{position} {}
    size_t operator*() const { return i; }
    num_iterator& operator++() {
        ++i;
        return *this;
    }
    bool operator!=(const num_iterator &other) const {
        return i != other.i;
    }
};
```

4. Функция преобразования Фурье будет принимать сигнал и возвращать новый. Последний представляет собой преобразование Фурье, выполненное для входного сигнала. Обратное преобразование Фурье выполняется аналогично прямому, поэтому предоставим необязательный параметр булева типа, который указывает на направление преобразования. Обратите внимание: наличие подобных параметров зачастую указывает на плохой стиль программирования, особенно если в сигнатуре функции их несколько. Здесь мы для краткости применили всего один параметр булева типа.

   Первое, что нужно сделать, — это выделить память для нового вектора сигналов, имеющего размер исходного сигнала:

```
csignal fourier_transform(const csignal &s, bool back = false)
{
    csignal t (s.size());
```

5. В формуле имеются два множителя, которые всегда выглядят одинаково. Поместим их в отдельные переменные:

```
    const double pol {2.0 * M_PI * (back ? -1.0 : 1.0)};
    const double div {back ? 1.0 : double(s.size())};
```

6. Алгоритм `std::accumulate` отлично подходит для выполнения формул, которые складывают элементы. Мы воспользуемся им для диапазона увеличивающихся

численных значений. На основе этих значений можно сформировать отдельные слагаемые для каждого шага. Алгоритм `std::accumulate` на каждом шаге вызывает бинарную функцию. Первым параметром данной функции будет текущее значение части суммы, которая уже была подсчитана на предыдущих шагах, а второй параметр — следующее значение диапазона. Мы выполняем поиск значения сигнала `s` в текущей позиции и умножаем его на комплексный множитель, `pol`. Затем возвращаем новую частичную сумму. Бинарная функция обернута в *другое* лямбда-выражение, так как мы станем использовать разные значения переменной `j` при каждом вызове алгоритма `accumulate`. Поскольку этот алгоритм цикла двумерный, внутреннее лямбда-выражение применяется для внутреннего цикла, а внешнее — для внешнего.

```
auto sum_up ([=, &s] (size_t j) {
    return [=, &s] (cmplx c, size_t k) {
        return c + s[k] *
            polar(1.0, pol * k * j / double(s.size()));
    };
});
```

7. Внутренняя часть преобразования Фурье теперь выполняется алгоритмом `std::accumulate`. Для каждой позиции алгоритма, кратной `j`, подсчитываем сумму всех слагаемых для позиций $i = 0... N$. Эта идея оборачивается в лямбда-выражение, которое мы будем выполнять для каждой точки графика полученного вектора преобразования Фурье:

```
auto to_ft ([=, &s](size_t j){
    return accumulate(num_iterator{0},
                      num_iterator{s.size()},
                      cmplx{},
                      sum_up(j))
        / div;
});
```

8. До этого момента мы не выполняли код самого преобразования Фурье. Мы лишь подготовили множество вспомогательного кода, который сейчас и задействуем. Вызов `std::transform` сгенерирует значения $j = 0... N$ для внешнего цикла. Преобразованные значения будут помещены в вектор `t`, который мы и вернем вызывающей стороне:

```
    transform(num_iterator{0}, num_iterator{s.size()},
              begin(t), to_ft);
    return t;
}
```

9. Реализуем отдельные функции, которые позволяют создать объекты функций для генерации сигналов. Первая из них представляет собой генератор косинусоидального сигнала. Она возвращает лямбда-выражение, способное сгенерировать косинусоидальный сигнал на основе заданной длины периода. Сам сигнал

может иметь произвольную длину, но его длина периода будет фиксированной. Длина периода *N* означает, что сигнал повторит себя спустя *N* шагов. Лямбда-выражение не принимает никаких параметров. Можно постоянно вызывать его, и для каждого вызова оно будет возвращать точку графика сигнала для следующего момента времени.

```
static auto gen_cosine (size_t period_len){
    return [period_len, n{0}] () mutable {
        return cos(double(n++) * 2.0 * M_PI / period_len);
    };
}
```

10. Вторым сигналом будет прямоугольная волна. Она колеблется между значениями `-1` и `+1` и не имеет других значений. Формула выглядит сложной, но она попросту преобразует линейное увеличивающееся значение `n` в `+1` или `-1`, а изменяющаяся длина периода равна `period_len`.

    Обратите внимание: в этот раз мы инициализируем `n` значением, не равным `0`. Таким образом, наша прямоугольная волна начинается в фазе, где ее выходные значения начинаются с `+1`.

```
static auto gen_square_wave (size_t period_len)
{
    return [period_len, n{period_len*7/4}] () mutable {
        return ((n++ * 2 / period_len) % 2) * 2 - 1.0;
    };
}
```

11. Сгенерировать сам сигнал с помощью указанных генераторов можно, выделив память для нового вектора и заполнив его значениями, сгенерированными на основе повторяющихся вызовов функции-генератора. Это делает функция `std::generate`. Она принимает пару итераторов (начальный и конечный) и функцию-генератор. Для каждой корректной позиции итератора она выполняет операцию `*it = gen()`. Обернув данный код в функцию, мы легко сможем сгенерировать векторы сигналов.

```
template <typename F>
static csignal signal_from_generator(size_t len, F gen)
{
    csignal r (len);
    generate(begin(r), end(r), gen);
    return r;
}
```

12. В самом конце нужно вывести на экран полученные сигналы. Можно легко вывести сигнал, скопировав его значения в итератор вывода потока, но сначала следует преобразовать данные, поскольку точки графиков наших сигналов представляют собой пары комплексных значений. К этому моменту требуется

только действительная часть каждой точки графика; так что помещаем значения в вызов `std::transform`, который извлекает лишь эту часть:

```
static void print_signal (const csignal &s)
{
    auto real_val ([](cmplx c) { return c.real(); });
    transform(begin(s), end(s),
              ostream_iterator<double>{cout, " "}, real_val);
    cout << '\n';
}
```

13. Мы реализовали формулу Фурье, но у нас еще нет сигналов для преобразования. Создаем их в функции `main`. Сначала определим стандартную длину сигнала, которой будут соответствовать все создаваемые сигналы:

```
int main()
{
    const size_t sig_len {100};
```

14. Теперь сгенерируем сигналы, преобразуем их и выведем на экран — это произойдет на трех следующих шагах. Первый шаг — генерация косинусоидального и прямоугольного сигналов. Они имеют одинаковые длину сигнала и длину периода:

```
    auto cosine      (signal_from_generator(sig_len,
           gen_cosine(     sig_len / 2)));
    auto square_wave (signal_from_generator(sig_len,
           gen_square_wave(sig_len / 2)));
```

15. Теперь у нас есть сигналы, представляющие собой косинусоидальную функцию и прямоугольную волну. Чтобы сгенерировать третий сигнал, который будет находиться между ними, возьмем сигнал прямоугольной волны и определим его преобразование Фурье (сохраним его в векторе `trans_sqw`). Преобразование Фурье для прямоугольной волны имеет характерную форму, мы несколько изменим ее. Все элементы с позиций от `10` до (`signal_length - 10`) имеют значение `0.0`. Остальные элементы остаются *неизменными*. Трансформация этого измененного преобразования Фурье обратно к представлению времени сигнала даст другой сигнал. В конце мы увидим, как он выглядит.

```
    auto trans_sqw (fourier_transform(square_wave));
    fill (next(begin(trans_sqw), 10), prev(end(trans_sqw), 10), 0);
    auto mid (fourier_transform(trans_sqw, true));
```

16. Теперь у нас есть три сигнала: `cosine`, `mid` и `square_wave`. Для каждого из них теперь выведем сам сигнал и его преобразование Фурье. На выходе программы увидим шесть очень длинных строк, содержащих значения типа `double`:

```
    print_signal(cosine);
    print_signal(fourier_transform(cosine));
    print_signal(mid);
    print_signal(trans_sqw);
    print_signal(square_wave);
    print_signal(fourier_transform(square_wave));
}
```

17. Компиляция и запуск программы приведут к тому, что экран консоли будет заполнен множеством численных значений. Если мы построим график для полученного результата, то увидим следующее изображение (рис. 6.4).

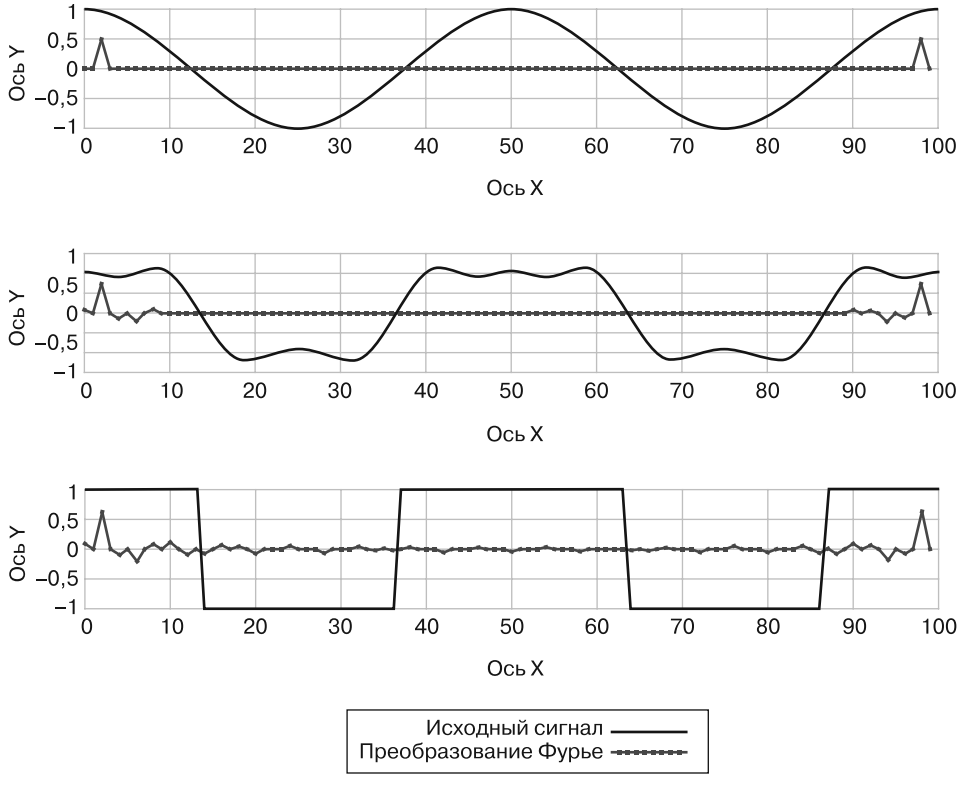

Рис. 6.4

Как это работает

Программа состоит из двух сложных фрагментов. Один из них — само преобразование Фурье, а другой — генерация сигналов с помощью изменяемых лямбда-выражений.

Сначала сконцентрируемся на преобразовании Фурье. Основа реализации формулы, созданной с применением циклов (которой мы не пользовались, а лишь рассмотрели во введении), выглядит так:

```
for (size_t k {0}; k < s.size(); ++k) {
    for (size_t j {0}; j < s.size(); ++j) {
        t[k] += s[j] * polar(1.0, pol * k * j / double(s.size()));
    }
}
```

С помощью алгоритмов STL `std::transform` и `std::accumulate` мы написали код, который можно подытожить, используя следующий псевдокод:

```
transform(num_iterator{0}, num_iterator{s.size()}, ...
    accumulate((num_iterator0}, num_iterator{s.size()}, ...
        c + s[k] * polar(1.0, pol * k * j / double(s.size()));
```

Мы получим точно такой же результат, что и в случае с циклом. Это, вероятно, пример ситуации, когда строгое следование алгоритмам STL не приводит к повышению качества кода. Тем не менее данная реализация алгоритма не знает о выбранной структуре данных. Она также будет работать со списками (однако в нашем случае это не будет иметь особого смысла). Еще одним преимуществом является тот факт, что алгоритмы C++17 STL легко *распараллелить* (данный вопрос мы рассмотрим в другой главе книги). А вот обычные циклы нужно реструктурировать, чтобы включить поддержку многопроцессорного режима (если только мы не используем внешние библиотеки наподобие *OpenMP*, в них циклы реструктурируются за нас).

Еще одна сложная часть — генерация сигналов. Еще раз взглянем на `gen_cosine`:

```
static auto gen_cosine (size_t period_len)
{
    return [period_len, n{0}] () mutable {
        return cos(double(n++) * 2.0 * M_PI / period_len);
    };
}
```

Каждый экземпляр лямбда-выражения представляет собой объект функции, который изменяет свое состояние при каждом вызове. Его состояние описывается переменными `period_len` и `n`. Последняя изменяется с каждым вызовом. Сигнал имеет различные значения в разные моменты времени, выражение `n++` описывает увеличивающиеся моменты времени. Чтобы получить сам вектор сигнала из выражения, мы создали вспомогательную функцию `signal_from_generator`:

```
template <typename F>
static auto signal_from_generator(size_t len, F gen)
{
    csignal r (len);
    generate(begin(r), end(r), gen);
    return r;
}
```

Эта вспомогательная функция выделяет память для вектора сигнала с заданной длиной и вызывает метод `std::generate`, что позволяет заполнить точки его графика. Для каждого элемента вектора `r` он один раз вызывает объект функции `gen`, который представляет собой самоизменяющийся объект функции; его можно создать с помощью `gen_cosine`.

К сожалению, способ решения задачи с помощью STL не позволяет сделать код более элегантным. Ситуация может измениться, если библиотека ranges будет включена в клуб STL (надо надеяться, что это случится в C++20).

Определяем ошибку суммы двух векторов

Существует несколько способов определения численной *ошибки* между целевым и реальным значениями. Измерение разницы между сигналами, состоящими из множества точек графика, обычно подразумевает использование циклов и вычитание соответствующих точек графика и т. д.

Существует простая формула для определения этой ошибки между сигналами `a` и `b` (рис. 6.5).

$$e = \sum_{i=0}^{N-1} (a_i - b_i)^2$$

Рис. 6.5

Для каждого значения *i* мы вычисляем *a[i] – b[i]*, возводим разность в квадрат (таким образом, получаем возможность сравнить положительные и отрицательные значения) и, наконец, складываем эти значения. Опять же мы могли бы просто воспользоваться циклом, но ради интереса сделаем это с помощью алгоритма STL. Плюс данного подхода заключается в том, что мы не зависим от структуры данных. Наш алгоритм будет работать для векторов и для спископодобных структур данных, для которых нельзя выполнить прямое индексирование.

Как это делается

В этом примере мы создадим два сигнала и посчитаем для них ошибку суммы.

1. Как и обычно, сначала приводим выражения `include`. Затем объявляем об использовании пространства имен `std`:

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <vector>
#include <iterator>

using namespace std;
```

2. Определим ошибку суммы двух сигналов. Таковыми выступят синусоидальная волна и ее копия, только оригинал будет сохранен в векторе, содержащем переменные типа `double`, а копия — в векторе, включающем переменные типа `int`. Поскольку копирование значения из переменной типа `double` в переменную типа `int` приводит к потере той его части, что стоит после десятичной точки, мы *потеряем* какие-то данные. Назовем содержащий переменные типа `double` вектор `as`, что расшифровывается как *analog signal* (аналоговый сигнал),

а вектор, который содержит значения типа `int`, — `ds`, что значит *digital signal* (цифровой сигнал). Ошибка суммы позднее покажет, насколько велики потери данных.

```
int main()
{
    const size_t sig_len {100};
    vector<double> as (sig_len); // a для аналогового сигнала
    vector<int>    ds (sig_len); // d для цифрового сигнала
```

3. Чтобы сгенерировать сигнал синусоидальной волны, реализуем небольшое лямбда-выражение, имеющее *изменяемое* значение счетчика `n`. Его можно вызывать по мере необходимости, и при каждом вызове оно будет возвращать значение для следующей временной точки синусоидальной волны. Вызов `std::generate` заполняет вектор сигнала, а вызов `std::copy` копирует все значения из вектора переменных типа `double` в вектор переменных типа `int`.

```
    auto sin_gen ([n{0}] () mutable {
        return 5.0 * sin(n++ * 2.0 * M_PI / 100);
    });
    generate(begin(as), end(as), sin_gen);
    copy(begin(as), end(as), begin(ds));
```

4. Сначала выведем на экран сигналы, чтобы позднее можно было построить для них график:

```
    copy(begin(as), end(as),
         ostream_iterator<double>{cout, " "});
    cout << '\n';
    copy(begin(ds), end(ds),
         ostream_iterator<double>{cout, " "});
    cout << '\n';
```

5. Теперь перейдем к самой ошибке суммы. Используем метод `std::inner_product`, поскольку его можно легко адаптировать для определения разницы между двумя соответствующими элементами вектора сигналов. Он будет итерировать по обоим диапазонам данных, выбирать элементы в соответствующих позициях диапазонов, рассчитывать их разность и складывать результат.

```
    cout << inner_product(begin(as), end(as), begin(ds),
                          0.0, std::plus<double>{},
                          [](double a, double b) {
                              return pow(a - b, 2);
                          })
         << '\n';
}
```

6. Компиляция и запуск программы приведут к выводу двух длинных строк, содержащих сигналы, и третьей строки, в которой показывается единственное значение — разность сигналов. Эта разность равна `40.889`. Если бы мы рассчитывали ошибку непрерывно, сначала для первой пары элементов, затем — для

первых двух пар, затем — для первых трех и т. д., то получили бы накопленную кривую ошибок. Ее можно увидеть на построенном графике (рис. 6.6).

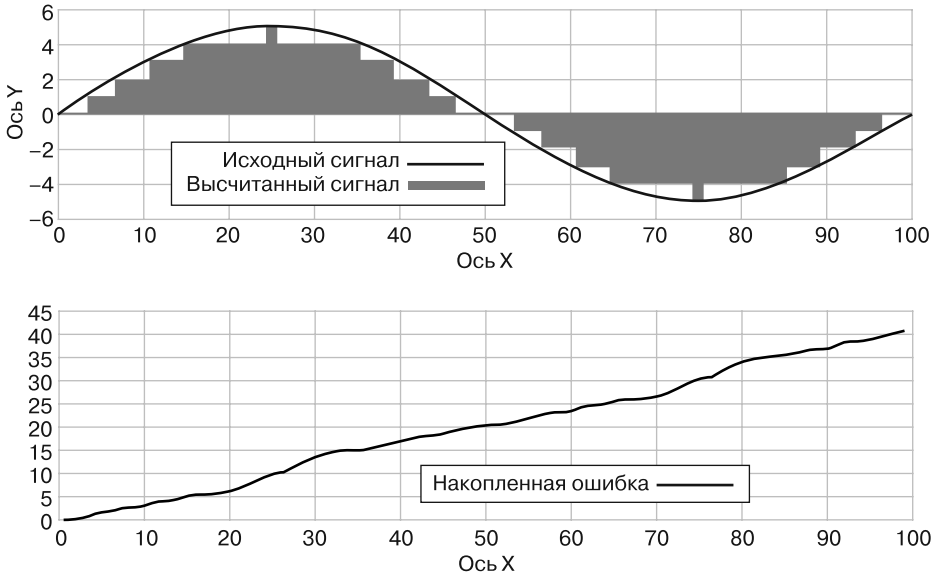

Рис. 6.6

Как это работает

В данном примере мы решили задачу прохода в цикле по двум векторам, получения разности между их соответствующими значениями, возведения их в квадрат и суммирования этих значений с помощью одного вызова `std::inner_product`. В то же время единственным кодом, который мы написали, стало лямбда-выражение `[](double a, double b) { return pow(a - b, 2); }`, принимающее разность аргументов и возводящее ее в квадрат.

Взглянув на потенциальную реализацию метода `std::inner_product`, можно увидеть, как и почему это работает:

```
template<class InIt1, class InIt2, class T, class F, class G>
T inner_product(InIt1 it1, InIt1 end1, InIt2 it2, T val,
                F bin_op1, G bin_op2)
{
    while (it1 != end1) {
        val = bin_op1(val, bin_op2(*it1, *it2));
        ++it1;
        ++it2;
    }
    return value;
}
```

Алгоритм принимает пару итераторов (начальный и конечный) для первого диапазона данных, а также начальный итератор для второго диапазона. В нашем случае они представляют собой векторы, для которых нужно определить ошибку суммы. Следующий символ — исходное значение `val`. Мы инициализировали его значением `0.0`. Затем алгоритм принимает две бинарные функции, которые называются `bin_op1` и `bin_op2`.

К этому моменту мы понимаем, что данный алгоритм очень похож на `std::accumulate`. Единственное отличие состоит в том, что последний работает только для *одного* диапазона данных. Если мы заменим выражение `bin_op2(*it1, *it2)` на `*it`, то перейдем к алгоритму `accumulate`. Поэтому можно считать `std::inner_product` версией алгоритма `std::accumulate`, которая *упаковывает* пары входных значений.

В нашем случае функция-*упаковщик* выглядит как `pow(a - b, 2)`. Для другой функции, `bin_op1`, мы выбрали `std::plus<double>`, поскольку хотим сложить сразу все значения, возведенные в квадрат.

Реализуем отрисовщик множества Мандельброта в ASCII

В 1975 году математик Бенуа Мандельброт (Benoît Mandelbrot) придумал термин «*фрактал*». Это математическое множество с интересными математическими свойствами, которое в конечном счете выглядит как произведение искусства. Фракталы в приближении выглядят так, будто *повторяются бесконечно*. Одним из самых популярных фракталов является множество Мандельброта, его вы можете увидеть на рис. 6.7.

Рис. 6.7

Рисунок множества Мандельброта можно сгенерировать путем повторения специальной формулы (рис. 6.8).

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

Рис. 6.8

Переменные *z* и *c* являются *комплексными* числами. Множество Мандельброта содержит все значения *c*, для которых формула *сходится*, если они применяются достаточно часто (рис. 6.7). Одни значения сходятся раньше, другие — позже, так что их можно раскрасить разными цветами. Некоторые из них не сходятся вообще — они отмечены черным.

В STL предусмотрен полезный класс `std::complex`. Попробуем реализовать эту формулу, не используя явные циклы, только ради того, чтобы получше узнать STL.

Как это делается

В этом примере мы выведем на консоль такое же изображение, которое было показано на рис. 6.8, в формате ASCII.

1. Сначала включим все заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <complex>
#include <numeric>
#include <vector>

using namespace std;
```

2. Множество Мандельброта и формула работают с комплексными числами. Поэтому определим псевдоним типа `cmplx` так, что он имеет типаж `std::complex`, специализированный для значений типа `double`:

```
using cmplx = complex<double>;
```

3. Вы можете скомпоновать весь код для отрисовки изображения для множества Мандельброта с помощью ASCII примерно за 20 строк кода, но мы реализуем каждый логический шаг отдельно, а затем соберем все воедино. Первый шаг — реализация функции, которая переводит координаты из целых чисел в числа с плавающей точкой. Изначально мы имеем столбцы и строки для всех позиций символов на консоли. Нужно получить координаты с типом `complex` для системы координат множества Мандельброта. Для этого реализуем функцию, которая принимает параметры, описывающие геометрию системы координат пользовательского окна, а также систему, к которой нужно их привести. Эти значения служат для построения лямбда-выражения, которое будет возвращено

позднее. Лямбда-выражение принимает координату `int` и возвращает координату `double`.

```
static auto scaler(int min_from, int max_from,
                   double min_to, double max_to)
{
    const int    w_from   {max_from - min_from};
    const double w_to     {max_to - min_to};
    const int    mid_from {(max_from - min_from) / 2 + min_from};
    const double mid_to   {(max_to - min_to) / 2.0 + min_to};
    return [=] (int from) {
        return double(from - mid_from) / w_from * w_to + mid_to;
    };
}
```

4. Теперь можно преобразовать точки в одном измерении, но множество Мандельброта существует в двумерной системе координат. Чтобы выполнить преобразование из одной системы координат `(x, y)` в другую, объединим `scaler_x` и `scaler_y`, а также создадим экземпляр типа `cmplx` на основе их выходных данных.

```
template <typename A, typename B>
static auto scaled_cmplx(A scaler_x, B scaler_y)
{
    return [=](int x, int y) {
        return cmplx{scaler_x(x), scaler_y(y)};
    };
}
```

5. После получения возможности преобразовывать координаты в правильные измерения можно реализовать множество Мандельброта. Функция, которую мы реализуем, сейчас ничего не знает о концепции консольных окон или линейного тангенциального преобразования, поэтому можно сконцентрироваться на математике, описывающей множество Мандельброта. Возводим в квадрат значение `z` и добавляем к нему значение `c` в цикле до тех пор, пока его значение по модулю меньше `2`. Для некоторых координат это не происходит никогда, так что прерываем цикл, если будет превышено максимальное количество итераций `max_iterations`. В конечном счете мы вернем количество итераций, которое успели выполнить до того, как сойдется значение по модулю.

```
static auto mandelbrot_iterations(cmplx c)
{
    cmplx z {};
    size_t iterations {0};
    const size_t max_iterations {1000};
    while (abs(z) < 2 && iterations < max_iterations) {
        ++iterations;
        z = pow(z, 2) + c;
    }
    return iterations;
}
```

6. Теперь можно начать с функции `main`, где определим измерения консоли и создадим объект функции `scale`, который будет масштабировать значения наших координат для обеих осей:

```
int main()
{
    const size_t w {100};
    const size_t h {40};
    auto scale (scaled_cmplx(
        scaler(0, w, -2.0, 1.0),
        scaler(0, h, -1.0, 1.0)
    ));
```

7. Чтобы выполнить линейный перебор всего изображения, напишем еще одну функцию преобразования, которая принимает одномерную координату `i`. На ее основе она определяет координаты `(x, y)`, используя предполагаемую длину строки. После разбиения переменной `i` на количество строк и колонок она преобразует их с помощью нашей функции `scale` и возвращает комплексную координату:

```
    auto i_to_xy ([=](int i) { return scale(i % w, i / w); });
```

8. Сейчас можно преобразовать одномерные координаты (с типом `int`) с помощью двумерных координат (с типом `(int, int)`) во множество координат Мандельброта (с типом `cmplx`), а затем рассчитать количество итераций (снова тип `int`). Объединим все это в одну функцию, которая создаст подобную цепочку вызовов:

```
    auto to_iteration_count ([=](int i) {
        return mandelbrot_iterations(i_to_xy(i));
    });
```

9. Теперь подготовим все данные. Предположим, что итоговое изображение ASCII имеет `w` символов в длину и `h` символов в ширину. Его можно сохранить в одномерном векторе, который имеет `w * h` элементов. Мы заполним данный вектор с помощью `std::iota` значениями из диапазона *0 ... (w*h – 1)*. Эти числа можно использовать в качестве входного источника для нашего диапазона данных функции преобразования, который мы инкапсулировали, применяя `to_iteration_count`.

```
    vector<int> v (w * h);
    iota(begin(v), end(v), 0);
    transform(begin(v), end(v), begin(v), to_iteration_count);
```

10. На этом, по сути, все. Теперь у нас есть вектор `v`, который мы инициализировали одномерными координатами и переписали счетчиком итераций для множества Мандельброта. На его основе можно вывести красивое изображение. Можно сделать окно консоли длиной `w` символов, чтобы не выводить на экран символ перевода строки. Но мы можем также *нестандартно использовать* алгоритм `std::accumulate`, чтобы он добавил разрывы строк за нас. Он применяет

бинарную функцию для сокращения диапазона. Предоставим ему бинарную функцию, принимающую итератор вывода (который мы свяжем с терминалом на следующем шаге) и отдельное значение из диапазона. Выведем это значение как символ *, если количество итераций превышает 50. В противном случае выведем пробел. При нахождении в *конце строки* (поскольку переменная-счетчик `n` без остатка делится на `w`) выведем символ разрыва строки:

```
auto binfunc ([w, n{0}] (auto output_it, int x) mutable {
    *++output_it = (x > 50 ? '*' : ' ');
    if (++n % w == 0) { ++output_it = '\n'; }
    return output_it;
});
```

11. Вызывая функцию `std::accumulate` для входного диапазона данных вместе с нашей бинарной функцией `print` и итератором `ostream_iterator`, можно отправить рассчитанное множество Мандельброта в окно консоли:

```
    accumulate(begin(v), end(v), ostream_iterator<char>{cout},
               binfunc);
}
```

12. Компиляция и запуск программы приводят к следующему результату, который выглядит как изначальное детализированное множество Мандельброта в упрощенной форме (рис. 6.9).

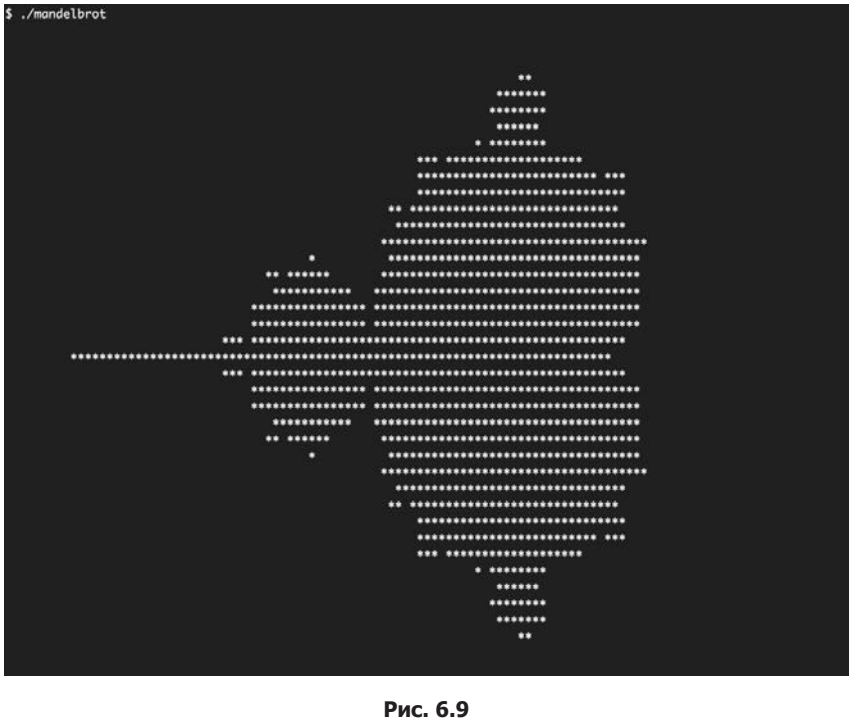

Рис. 6.9

Как это работает

Все расчеты происходят во время вызова `std::transform` для одномерного массива:

```
vector<int> v (w * h);
iota(begin(v), end(v), 0);
transform(begin(v), end(v), begin(v), to_iteration_count);
```

Что же произошло и почему это работает именно так? Функция `to_iteration_count`, по сути, представляет собой цепочку вызовов от `i_to_xy` до `scale` и `mandelbrot_iterations`. На рис. 6.10 показаны этапы преобразования.

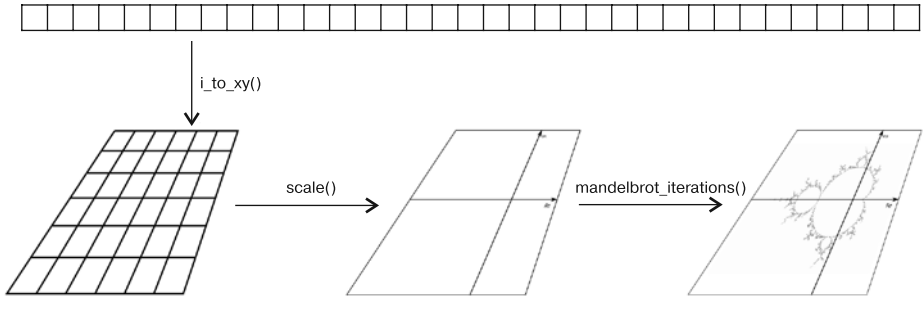

Рис. 6.10

Подобным способом в качестве входного параметра можно использовать индекс одномерного массива и получать количество итераций множества Мандельброта для точки двумерной плоскости, которую представляет точка массива. К счастью, эти три преобразования совершенно не взаимозависимы. Модули подобного кода можно легко протестировать отдельно друг от друга. Таким образом, находить и исправлять ошибки очень легко, как и просто утверждать о корректности кода.

Создаем собственный алгоритм split

В некоторых ситуациях существующих алгоритмов STL недостаточно. Но ничто не запрещает нам реализовать собственный алгоритм. Прежде чем решать конкретную задачу, следует тщательно ее обдумать, чтобы понять: многие задачи можно решить путем обобщения. Если мы будем регулярно добавлять в наши библиотеки новый код по мере решения собственных задач, то можем помочь коллегам-программистам, у которых появятся аналогичные задачи. Идея заключается в том, чтобы знать, когда ваш код является достаточно обобщенным и когда не нужно обобщать его еще больше, в противном случае у нас получится новый язык общего назначения.

В этом примере мы реализуем алгоритм, который назовем `split`. Он может разбить любой диапазон данных, используя в качестве разделителя каждое включение конкретного значения, и скопировать полученный результат в выходной диапазон данных.

Как это делается

В данном примере мы реализуем собственный алгоритм `split` и проверим его работу, разбив на фрагменты строку-пример.

1. Сначала включим некоторые части библиотеки STL и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Алгоритм, показанный в этом разделе, предназначен для разбиения диапазонов данных. Он принимает начальный и конечный итераторы, а также итератор вывода, что поначалу делает его похожим на алгоритмы `std::copy` и `std::transform`. Другими его параметрами являются `split_val` и `bin_func`. Параметр `split_val` — значение, которое мы ищем во входном диапазоне данных; оно представляет собой точку разбиения, по которой мы отделяем входной диапазон данных. Параметр `bin_func` — функция, преобразующая пару итераторов, которые отмечают начало и конец этого поддиапазона. Мы проитерируем по входному диапазону данных с помощью `std::find`, так что будем перескакивать между включениями значений `split_val`. При разбиении отдельной строки на отдельные слова мы станем перескакивать от пробела к пробелу. При нахождении искомого значения останавливаемся, чтобы сформировать фрагмент и передать его в выходной диапазон данных:

```
template <typename InIt, typename OutIt, typename T, typename F>
InIt split(InIt it, InIt end_it, OutIt out_it, T split_val,
           F bin_func)
{
    while (it != end_it) {
        auto slice_end (find(it, end_it, split_val));
        *out_it++ = bin_func(it, slice_end);
        if (slice_end == end_it) { return end_it; }
        it = next(slice_end);
    }
    return it;
}
```

3. Воспользуемся новым алгоритмом. Создадим строку, которую затем разобьем на части. Элементом, отмечающим конец последнего фрагмента и начало следующего, будет дефис `'-'`:

```
int main()
{
    const string s {"a-b-c-d-e-f-g"};
```

4. Когда алгоритм вызывает функцию `bin_func` для пары итераторов, мы хотим создать на его основе новую строку:

```
    auto binfunc ([](auto it_a, auto it_b) {
        return string(it_a, it_b);
    });
```

5. Выходной диапазон данных представляет собой список строк. Мы вызовем алгоритм `split`, который спроектирован так, что похож на другие алгоритмы STL:

```
    list<string> l;
    split(begin(s), end(s), back_inserter(l), '-', binfunc);
```

6. Чтобы увидеть полученное, выведем на экран новый список разбитых строк:

```
    copy(begin(l), end(l), ostream_iterator<string>{cout, "\n"});
}
```

7. Компиляция и запуск программы дадут следующий результат. Он не содержит дефисов и показывает, что включает отдельные слова (которые в нашем примере являются отдельными символами):

```
$ ./split
a
b
c
d
e
f
g
```

Как это работает

Алгоритм `split` работает так же, как и `std::transform`, поскольку принимает начальный и конечный итераторы входного диапазона данных и итератор вывода. Он выполняет какие-то действия с входным диапазоном и в конечном счете присваивает его итератору вывода. Помимо этого, он принимает элемент `split_val` и бинарную функцию. Снова взглянем на полную реализацию, чтобы полностью понять его:

```
template <typename InIt, typename OutIt, typename T, typename F>
InIt split(InIt it, InIt end_it, OutIt out_it, T split_val, F bin_func)
{
    while (it != end_it) {
        auto slice_end (find(it, end_it, split_val));
        *out_it++ = bin_func(it, slice_end);
        if (slice_end == end_it) { return end_it; }
        it = next(slice_end);
    }
    return it;
}
```

Цикл требует выполнять перебор до конца входного диапазона данных. Во время каждой итерации вызов `std::find` используется для поиска следующего элемента

входного диапазона, который равен `split_val`. В нашем случае этот элемент — дефис (`'-'`), поскольку мы хотим разбить входную строку на фрагменты, находящиеся между дефисами. Следующая позиция дефиса теперь сохраняется в `slice_end`. После перебора цикла итератор `it` перемещается на следующий после искомого элемент. Таким образом, цикл перескакивает от дефиса к дефису вместо того, чтобы проходить по отдельным элементам.

В данной комбинации итератор `it` указывает на начало последнего `slice`, а `slice_end` — на конец последней вырезки. Оба итератора отмечают начало и конец поддиапазона данных, который представляет собой ровно одну вырезку между двумя символами дефиса. Для строки `"foo-bar-baz"` это значит, что у нас будет три итерации цикла и всякий раз мы будем получать пару итераторов, которые окружают одно слово. Но нам нужны не итераторы, а подстроки. Бинарная функция `bin_func` помогает их получить. При вызове функции `split` мы передали ей следующую бинарную функцию:

```
[](auto it_a, auto it_b) {
    return string(it_a, it_b);
}
```

Функция `split` пропускает каждую пару итераторов через функцию `bin_func`, прежде чем отправить их в конечный итератор. От функции `bin_func` мы получим строки `"foo"`, `"bar"` и `"baz"`.

Дополнительная информация

Интересной альтернативой реализации нашего алгоритма, разбивающего строки на части, является реализация *итератора*, который делает то же самое. Мы сейчас не будем реализовывать такой итератор, но кратко рассмотрим подобный сценарий.

Этот итератор должен перескакивать между разделителями при каждом инкременте. При разыменовании ему следует создавать объект строки на основе позиции, на которую он сейчас указывает, что можно сделать с помощью бинарной функции `binfunc`, уже применяемой нами ранее.

Если бы наш класс итератора назывался `split_iterator` и мы бы задействовали его вместо алгоритма `split`, то код пользователя выглядел бы так:

```
string s {"a-b-c-d-e-f-g"};
list<string> l;

auto binfunc ([](auto it_a, auto it_b) {
    return string(it_a, it_b);
});

copy(split_iterator{begin(s), end(s), '-', binfunc},{}, back_inserter(l));
```

Недостатком описанного подхода служит тот факт, что реализовать итератор сложнее, чем одну функцию. Кроме того, существует множество узких моментов

в коде итератора, которые могут привести к появлению ошибок, поэтому такое решение требует более серьезного тестирования. С другой стороны, очень легко объединить подобный итератор с другими алгоритмами библиотеки STL.

Создаем полезные алгоритмы на основе стандартных алгоритмов gather

Алгоритм `gather` — очень хороший пример компонуемости алгоритмов STL. Шон Пэрент (Sean Parent), будучи старшим научным сотрудником в компании Adobe Systems, популяризировал данный алгоритм, поскольку он полезен и краток. Способ его реализации идеально подчеркивает идею STL-компоновки алгоритмов.

Алгоритм `gather` работает для диапазонов данных произвольных типов. Он изменяет порядок элементов так, что конкретные элементы собираются вокруг заданной позиции, выбранной вызывающей стороной.

Как это делается

В данном примере мы реализуем алгоритм `gather` и его дополнительную вариацию. После этого посмотрим, как его можно использовать.

1. Сначала добавим все выражения `include`. Затем объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Алгоритм `gather` представляет собой хороший пример компоновки стандартных алгоритмов. Он принимает начальный и конечный итераторы, а также еще один итератор `gather_pos`, который указывает на какую-то позицию между ними. Последний параметр — функция-предикат. С ее помощью алгоритм поместит все элементы, соответствующие заданному условию, в позиции рядом с итератором `gather_pos`. Реализация перемещения элементов выполняется благодаря `std::stable_partition`. Алгоритм `gather` возвращает пару итераторов. Они возвращаются вызовом `stable_partition` и, таким образом, отмечают начало и конец полученного диапазона:

```
template <typename It, typename F>
pair<It, It> gather(It first, It last, It gather_pos, F predicate)
{
    return {stable_partition(first, gather_pos, not_fn(predicate)),
            stable_partition(gather_pos, last, predicate)};
}
```

3. Еще одним вариантом реализации является `gather_sort`. Он работает так же, как и `gather`, но принимает не унарную функцию-предикат, а бинарную функцию сравнения. Это позволит собрать значения вокруг позиции `gather_pos`, они могут быть как самыми маленькими, так и самыми большими:

```
template <typename It, typename F>
void gather_sort(It first, It last, It gather_pos, F comp_func)
{
    auto inv_comp_func ([&](const auto &...ps) {
        return !comp_func(ps...);
    });
    stable_sort(first,      gather_pos, inv_comp_func);
    stable_sort(gather_pos, last,       comp_func);
}
```

4. Воспользуемся этими алгоритмами. Начнем с предиката, который указывает, является ли заданный символьный аргумент символом `'a'`. Мы создадим строку, состоящую из комбинации символов `'a'` и `'_'`:

```
int main()
{
    auto is_a ([](char c) { return c == 'a'; });
    string a {"a_a_a_a_a_a_a_a_a_a"};
```

5. Создадим итератор, который указывает на середину нашей новой строки. Вызовем для нее алгоритм `gather` и посмотрим, что произойдет. В результате вызова символы `'a'` будут собраны в середине строки:

```
    auto middle (begin(a) + a.size() / 2);

    gather(begin(a), end(a), middle, is_a);
    cout << a << '\n';
```

6. Снова вызовем данный алгоритм, но в этот раз итератор `gather_pos` будет указывать не на середину строки, а на ее начало:

```
    gather(begin(a), end(a), begin(a), is_a);
    cout << a << '\n';
```

7. В третьем вызове соберем элементы вокруг конечного итератора:

```
    gather(begin(a), end(a), end(a), is_a);
    cout << a << '\n';
```

8. При последнем вызове алгоритма `gather` снова попробуем собрать все символы в середине. Этот вызов сработает не так, как нам бы того хотелось, и далее мы увидим почему:

```
    // Это работает не так, как вы того ожидаете
    gather(begin(a), end(a), middle, is_a);
    cout << a << '\n';
```

9. Создадим еще одну строку, содержащую символы подчеркивания и числа. Для этой входной последовательности вызовем функцию `gather_sort`. Итератор `gather_pos` находится в середине строки, воспользуемся бинарной функцией сравнения `std::less<char>`:

```
    string b {"_9_2_4_7_3_8_1_6_5_0_"};
    gather_sort(begin(b), end(b), begin(b) + b.size() / 2,
                less<char>{});
    cout << b << '\n';
}
```

10. Компиляция и запуск программы даст следующий интересный результат. Первые три строки выглядят так, как мы и ожидали, но четвертая строка — так, будто алгоритм `gather` *не отработал*.

    В последней строке можно увидеть результат работы функции `gather_short`. Числа выглядят отсортированными в обоих направлениях.

```
$ ./gather
_____aaaaaaaaaa_____
aaaaaaaaaa__________
__________aaaaaaaaaa
__________aaaaaaaaaa
_____9743201568_____
```

Как это работает

Изначально алгоритм `gather` сложно понять, поскольку он очень короткий, но при этом выполняет задачу, которая кажется сложной. Разберем ее по шагам (рис. 6.11).

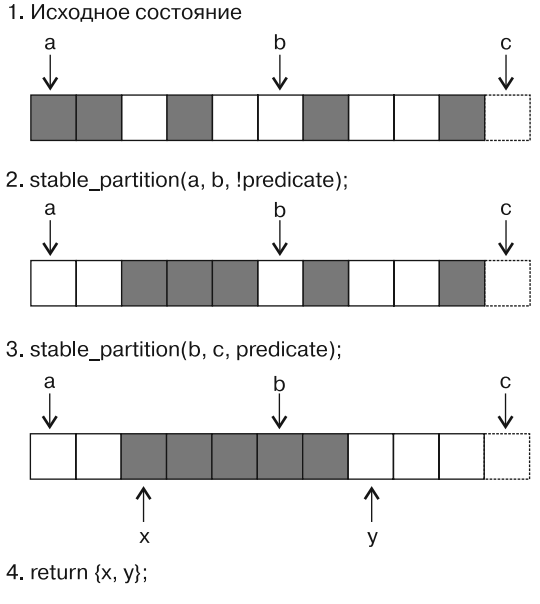

Рис. 6.11

1. В самом начале у нас есть диапазон элементов, для которого мы предоставляем функцию-предикат. На рис. 6.11 все элементы, для которых функция-предикат возвращает значение `true`, окрашены в *серый* цвет. Итераторы `a` и `c` отмечают

весь диапазон, а итератор `b` указывает на *направляющий* элемент. Таковым является элемент, вокруг которого мы хотим *собрать* все серые элементы.

2. Алгоритм `gather` вызывает функцию `std::stable_partition` для диапазона данных `[a, b)` и в то же время использует *инвертированную* версию предиката. Он инвертирует предикат, поскольку функция `std::stable_partition` перемещает все элементы, для которых предикат возвращает значение `true`, в переднюю часть диапазона данных. Нужно, чтобы произошло *противоположное*.
3. Выполняется еще один вызов `std::stable_partition`, однако на сей раз для диапазона данных `[b, c)`, *без* инвертирования предиката. Серые элементы перемещаются в начало входного диапазона данных; это значит, что они перемещаются к направляющему элементу, на который указывает итератор `b`.
4. Теперь элементы собраны вокруг итератора `b` и алгоритм возвращает итераторы, указывающие на начало и конец диапазона данных, содержащего серые элементы.

Мы несколько раз вызвали алгоритм `gather` для одного диапазона данных. Сначала собрали все элементы в середине диапазона данных. Затем собрали их в `begin()` и `end()` этих диапазонов. Эти случаи интересны тем, что всегда приводят *один* из вызовов `std::stable_partition` к работе с *пустым* диапазоном данных, а это влечет *бездействие*.

Для последнего вызова `gather` мы передали параметры (`begin`, `end`, `middle`) и не получили результат. Почему? На первый взгляд это похоже на баг, но на самом деле все не так.

Представьте, что у нас есть диапазон символов `"aabb"` и функция-предикат `is_character_a`, которая возвращает значение `true` для элементов `'a'`, — если мы вызовем ее с третьим итератором, указывающим на середину диапазона символов, то увидим такой же *баг*. Причина заключается в том, что первый вызов `stable_partition` будет работать с диапазоном `"aa"`, а второй — с диапазоном `"bb"`. Эта последовательность вызовов не даст получить результат `"baab"`, на который мы наивно надеялись.

Чтобы получить желаемый результат в последнем случае, можно было бы использовать вызов std::rotate(begin, begin + 1, end);.

Модификация `gather_sort`, по сути, аналогична алгоритму `gather`. Единственное отличие заключается в следующем: она принимает не унарную функцию-предикат, а бинарную функцию сравнения, как и `std::sort`. И вместо того, чтобы дважды вызывать `std::stable_partition`, она дважды вызывает `std::stable_sort`.

Функцию инвертирования сравнения нельзя реализовать с помощью `not_fn`, как мы это делали для алгоритма `gather`, поскольку `not_fn` не работает с бинарными функциями.

Удаляем лишние пробелы между словами

Зачастую полученные от пользователей строки могут иметь самое разное форматирование, и их нужно отредактировать. Например, нужно удалить повторяющиеся пробелы.

В этом разделе мы реализуем быстрый алгоритм удаления таких пробелов, который сохраняет одиночные пробелы. Мы назовем данный алгоритм `remove_multi_whitespace`, его интерфейс будет похож на интерфейсы алгоритмов STL.

Как это делается

В этом примере мы реализуем алгоритм `remove_multi_whitespace` и проверим его работоспособность.

1. Как и всегда, сначала приведем несколько директив `include` и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Реализуем новый алгоритм в стиле STL, который называется `remove_multi_whitespace`. Данный алгоритм удаляет избыточные включения пробелов, но не единичные случаи. Это значит, что строка `"a b"` останется неизменной, но строка `"a  b"` будет сокращена до `"a b"`. Для этого мы применим функцию `std::unique` с пользовательской бинарной функцией-предикатом. Функция `std::unqiue` итерирует по диапазону данных и всегда ищет соседствующие элементы. Затем с помощью функции-предиката она определяет, равны ли искомые элементы. Если да, то удаляет один из них. После вызова этой функции диапазон данных не будет содержать поддиапазонов, в которых одинаковые элементы стоят друг рядом с другом. Функции-предикаты, обычно применяемые в этом контексте, говорят, равны ли два элемента. Мы передадим функции `std::unique` предикат, который укажет, стоят ли рядом два *пробела*, чтобы удалить один из них. Как и в случае `std::unique`, мы принимаем начальный и конечный итераторы, а затем возвращаем итератор, указывающий на новый конец диапазона данных:

```
template <typename It>
It remove_multi_whitespace(It it, It end_it)
{
    return unique(it, end_it, [](const auto &a, const auto &b) {
        return isspace(a) && isspace(b);
    });
}
```

3. На этом все. Создадим строку, которая содержит лишние пробелы.

```
int main()
{
    string s {"fooo     bar    \t   baz"};
    cout << s << '\n';
```

4. Теперь воспользуемся *идиомой erase-remove*, чтобы избавиться от этих пробелов:

```
    s.erase(remove_multi_whitespace(begin(s), end(s)), end(s));

    cout << s << '\n';
}
```

5. Компиляция и запуск программы дадут следующий результат:

```
$ ./remove_consecutive_whitespace
fooo     bar        baz
fooo bar baz
```

Как это работает

Эта сложная задача была решена без привлечения циклов и сравнения элементов вручную. Мы только предоставили функцию-предикат, указывающую, являются ли заданные два символа *пробелами*. Затем передали этот предикат в функцию `std::unique`, и *вуаля*, лишние пробелы испарились. Хотя в этой главе есть и такие примеры, где приходится попотеть, чтобы выразить наши программы в алгоритмах STL, как раз этот алгоритм особенно красив и краток.

Как же работает эта интересная комбинация? Сперва взглянем на возможную реализацию функции `std::unique`:

```
template<typename It, typename P>


It unique(It it, It end, P p)
{
    if (it == end) { return end; }

    It result {it};
    while (++it != end) {
        if (!p(*result, *it) && ++result != it) {
            *result = std::move(*it);
        }
    }
    return ++result;
}
```

Цикл пошагово проходит по элементам диапазона данных до тех пор, пока они не будут удовлетворять условиям предиката. В момент нахождения такого элемента цикл перемещается на один элемент вперед относительно позиции, где предикат сработал в прошлый раз. Та версия функции `std::unique`, которая не принимает дополнительную функцию-предикат, проверяет, равны ли два соседних элемента.

Таким образом, она удаляет *повторяющиеся* символы, например строка `"abbbbbbc"` преобразуется в `"abc"`.

Нужно удалить не просто *все* повторяющиеся символы, но и такие же *пробелы*. Поэтому наш предикат звучит не как *«символы-аргументы равны»*, а как *«символы-аргументы являются пробелами»*.

Последнее, на что нужно обратить внимание: ни функция `std::unique`, ни алгоритм `remove_multi_whitespace` на самом деле не удаляют символы из строки. Они только перемещают символы внутри строки в соответствии с их семантикой и говорят, где находится новый конец строки. Однако нам все еще необходимо выполнить удаление теперь уже ненужных символов, стоящих между новым и старым концом строки. Именно поэтому мы написали следующую строку:

```
s.erase(remove_multi_whitespace(begin(s), end(s)), end(s));
```

Это похоже на *идиому erase-remove*, с которой мы познакомились в теме о векторах и списках.

Компрессия и декомпрессия строк

В этом разделе рассматривается довольно популярная задача, которую предлагают на собеседованиях на должность программиста. Вам нужно написать функцию, которая принимает строку наподобие `"aaaaabbbbbbbccc"` и преобразует ее к виду `"a5b7c3"`. Запись `"a5"` означает, что в строке присутствует пять символов `'a'`, а `"b7"` — что в строке семь символов `'b'`. Это очень простой алгоритм *сжатия*. Для обычного текста он не очень полезен, поскольку обычная речь не такая избыточная, и при ее изложении в виде текста она не станет гораздо короче, воспользуйся мы этой схемой сжатия. Однако алгоритм относительно просто реализовать даже в том случае, когда у нас под рукой есть только доска и фломастеры. Основная хитрость заключается в том, что вы легко можете написать код с ошибками, если программа изначально плохо структурирована. Работать со строками, как правило, не так сложно, но есть вероятность возникновения переполнения буфера в случае применения старомодных функций форматирования.

Попробуем решить эту задачу с помощью средств STL.

Как это делается

В этом примере мы реализуем простые функции `compress` и `decompress` для строк.

1. Сначала включим некоторые библиотеки STL, а затем объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Для нашего дешевого алгоритма сжатия попробуем найти фрагменты текста, содержащие последовательности одинаковых символов, и выполним для них компрессию. Начиная с некой позиции, нужно найти первый символ диапазона, который отличается от текущего. Для этого используем метод `std::find`. Затем возвращаем кортеж, содержащий итератор, указывающий на этот первый элемент, символьную переменную `c`, которая содержится в нашем поддиапазоне, и количество ее включений:

```
template <typename It>
tuple<It, char, size_t> occurrences(It it, It end_it)
{
    if (it == end_it) { return {it, '?', 0}; }
    const char c {*it};
    const auto diff (find_if(it, end_it,
                     [c](char x) { return c != x; }));
    return {diff, c, distance(it, diff)};
}
```

3. Алгоритм `compress` постоянно вызывает функцию `occurrences`. Таким образом, мы переходим от одной группы символов к другой. Строка `r << c << n` помещает символ в поток вывода, следом идет количество включений этого символа в данной части строки. Результатом работы программы является строка, которая автоматически увеличивается по мере работы программы. В конечном счете мы вернем объект типа `string`, содержащий сжатую строку:

```
string compress(const string &s)
{
    const auto end_it (end(s));
    stringstream r;

    for (auto it (begin(s)); it != end_it;) {
        const auto [next_diff, c, n] (occurrences(it, end_it));
        r << c << n;
        it = next_diff;
    }
    return r.str();
}
```

4. Метод `decompress` работает аналогично, но выглядит гораздо проще. Он постоянно пытается получить символьное значение из потока ввода, а затем и следующее за ним число. Из этих двух значений он может создать строку, содержащую столько включений символов, сколько указано в числе. В конечном счете мы снова вернем строку из выходного потока. Кстати говоря, данная функция декомпрессии *небезопасна*. Ее можно легко использовать со злым умыслом. Можете ли вы догадаться как? Мы рассмотрим эту проблему позже.

```
string decompress(const string &s)
{
    stringstream ss{s};
    stringstream r;
    char c;
```

```
    size_t n;
    while (ss >> c >> n) { r << string(n, c); }
    return r.str();
}
```

5. В нашей функции `main` мы создадим простую строку с большим количеством повторений, на которых алгоритм отработает очень хорошо. Выведем на экран сжатую версию, а затем и развернутую. В конечном счете мы должны получить исходную строку.

```
int main()
{
    string s {"aaaaaaaaabbbbbbbbbccccccccccc"};
    cout << compress(s) << '\n';
    cout << decompress(compress(s)) << '\n';
}
```

6. Компиляция и запуск программы дадут следующий результат:

```
$ ./compress
a9b9c11
aaaaaaaaabbbbbbbbbccccccccccc
```

Как это работает

Эта программа, по сути, состоит из двух функций: `compress` и `decompress`.

Функция `decompress` очень проста, поскольку состоит из объявлений переменных, одной строки кода, которая действительно что-то делает, и следующего за ней выражения `return`. Строка кода, выполняющего некие действия, выглядит так:

```
while (ss >> c >> n) { r << string(n, c); }
```

Она постоянно считывает символ `c` и переменную-счетчик `n` из строкового потока `ss`. Класс `stringstream` скрывает от нас возможности, связанные с анализом строки. Отработав успешно, функция возвращает развернутую строку в строковый поток, из которого итоговую строку можно вернуть вызывающей стороне. Если `c = 'a'` и `n = 5`, то выражение `string(n, c)` вернет строку `"aaaaa"`.

Функция `compress` выглядит более сложно. Мы также написали для нее небольшую вспомогательную функцию. Поэтому сначала взглянем на функцию `occurrences`. На рис. 6.12 показано, как она работает.

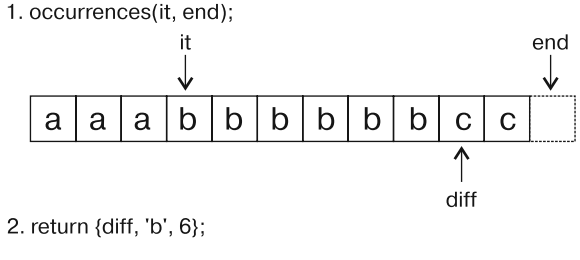

Рис. 6.12

Функция `occurences` принимает два параметра: итератор, указывающий на начало последовательности символов внутри диапазона данных, и конечный итератор этого набора. С помощью `find_if` мы находим первый символ, отличный от символа, на который изначально указывал итератор. На рис. 6.12 данный итератор называется `diff`. Разность между этой новой позицией и старой позицией итератора равна количеству одинаковых элементов (на рис. 6.12 `diff - it` равно `6`). После выполнения подсчетов итератор `diff` можно использовать повторно для выполнения нового поиска. Так что мы помещаем `diff`, символ поддиапазона и длину поддиапазона в кортеж и возвращаем его.

Имея подобную информацию, можно переходить от поддиапазона к поддиапазону и помещать промежуточные результаты в сжатую целевую строку:

```
for (auto it (begin(s)); it != end_it;) {
    const auto [next_diff, c, n] (occurrences(it, end_it));
    r << c << n;
    it = next_diff;
}
```

Дополнительная информация

В шаге 4 мы упоминали, что функция `decompress` небезопасна. Более того, ее легко использовать *со злым умыслом*.

Представьте следующую входную строку: `"a00000"`. Ее сжатие приведет к результату `"a1"`, поскольку в ней содержится только один символ, `'a'`. За ним следует пять символов `'0'`, что даст результат `"05"`. Итоговый результат выглядит как `"a105"`. К сожалению, эта сжатая строка гласит: *«Символ* `'a'`*, повторенный 105 раз»*. Это не имеет ничего общего с нашей исходной строкой. При ее декомпрессии мы получим строку длиной 105 символов вместо строки, состоящей всего из шести символов. Представьте то же самое для более крупных чисел — пользователь может легко *нарушить* использование кучи, поскольку наш алгоритм не готов к таким входным данным.

Чтобы это предотвратить, функция `compress` может, например, отвергать входные данные, содержащие числа, или особым образом их помечать. А алгоритм `decompress` способен принимать еще одно условное выражение, которое ограничивает максимальный размер итоговой строки. Попробуйте выполнить данное упражнение самостоятельно.

7 Строки, классы потоков и регулярные выражения

В этой главе:

- создание, конкатенация и преобразование строк;
- удаление пробелов из начала и конца строк;
- преимущества использования `std::string` без затрат на создание объектов `std::string`;
- чтение значений из пользовательского ввода;
- подсчет всех слов в файле;
- форматирование ваших выходных данных с помощью манипуляторов потока ввода/вывода;
- инициализация сложных объектов из файла вывода;
- заполнение контейнеров с применением итераторов `std::istream`;
- вывод любых данных на экран с помощью итераторов `std::ostream`;
- перенаправление выходных данных в файл для конкретных разделов кода;
- создание пользовательских строковых классов путем наследования `std::char_traits`;
- токенизация входных данных с помощью библиотеки для работы с регулярными выражениями;
- удобный и красивый динамический вывод чисел на экран в зависимости от контекста;
- перехват читабельных исключений для ошибок потока `std::iostream`.

Введение

Данная глава посвящена обработке строк, анализу и выводу на экран произвольных данных. Для этих задач STL предоставляет потоковую библиотеку *для работы с вводом-выводом*. Библиотека состоит из следующих классов, показанных в серых прямоугольниках (рис. 7.1).

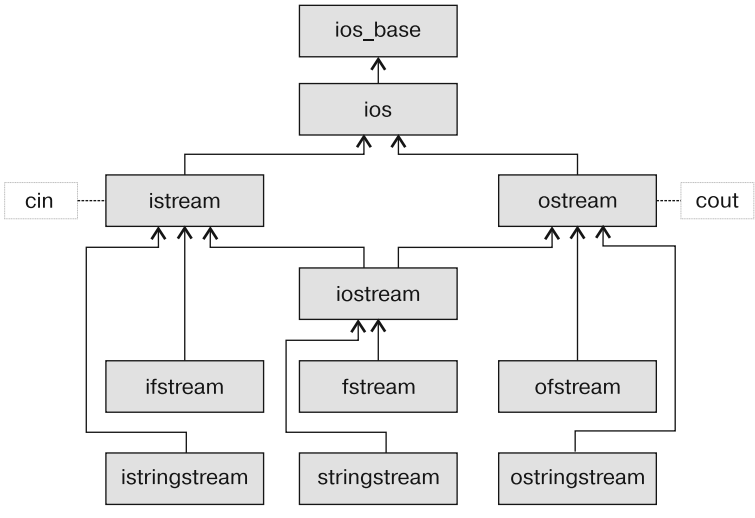

Рис. 7.1

Стрелки показывают схему наследования классов. Рисунок на первый взгляд может показаться непонятным, но в рамках главы мы рассмотрим все классы и познакомимся с ними по очереди. Если мы попробуем обратиться к документации C++ в STL, то не найдем упомянутые классы по *конкретно этим* именам. Причина такова: названия, приведенные на рис. 7.1, мы видим только как программисты приложений. Но они, по сути, являются просто ключевыми словами для классов с префиксом имени класса `basic_` (например, в документации STL проще найти класс `basic_istream`, в отличие от `istream`). Классы для работы с потоками, которые начинаются с префикса `basic_*`, являются шаблонными, они могут быть специализированы для разных типов символов. Классы, показанные на рис. 7.1, специализированы для значений типа `char`. В книге мы будем применять именно эти специализации. Если мы воспользуемся префиксом `w` для упомянутых имен класса, то получим названия `wistream`, `wostream` и т. д. — они являются специализациями для типа `wchar_t` вместо `char`.

В верхней части рис. 7.1 мы видим класс `std::ios_base`. Мы практически никогда не будем использовать непосредственно его; он приведен для полноты картины, поскольку другие классы наследуют от него. Следующая специализация — это `std::ios`, она воплощает идею объекта, сопровождающего поток данных, который может находиться в *исправном* состоянии, в состоянии, когда *закончились* данные (empty of data, EOF) или в другом *ошибочном* состоянии.

Первыми специализациями, которые мы применим на самом деле, являются `std::istream` и `std::ostream`. Префиксы `"i"` и `"o"` расшифровываются как input и output («ввод» и «вывод»). Мы уже видели такие префиксы на раннем этапе программирования на C++ в простейших примерах в объектах `std::cout` и `std::cin` (а также `std::cerr`). Экземпляры этих классов всегда доступны глобально. Мы выполняем вывод данных с помощью `ostream`, а ввод — с использованием `input`.

Класс, который наследует от классов `istream` и `ostream`, называется `iostream`. С его помощью можно выполнять как ввод, так и вывод данных. Зная, как использовать все классы из трио `istream`, `ostream` и `iostream`, вы сможете незамедлительно применить следующие классы:

- классы `ifstream`, `ofstream` и `fstream` наследуют от классов `istream`, `ostream` и `iostream` соответственно, но задействуют их возможности, чтобы перенаправить ввод/вывод в файлы или из них из *файловой системы* компьютера;
- классы `istringstream`, `ostringstream` и `iostringstream` работают по схожему принципу. Они помогают создавать строки в памяти, а затем помещают туда данные или считывают их.

Создание, конкатенация и преобразование строк

Даже те, кто довольно давно пользовался языком C++ , знают о классе `std::string`. Хотя в языке C обработка строк довольно утомительна, особенно при анализе, конкатенации и копировании и т. д., класс `std::string` — это реальный шаг вперед к простоте и безопасности.

Благодаря выходу C++11 теперь даже не нужно копировать строки, когда мы хотим передать право собственности какой-то другой функции или структуре данных, поскольку можем *перемещать* их. Таким образом, в подобных случаях не возникает больших издержек.

По мере выхода новых стандартов класс `std::string` получил несколько новых свойств. Совершенно новой особенностью C++17 является `std::string_view`. Мы немного поработаем с ней (но впереди будет и другой пример, в котором более подробно рассматривается `std::string_view`), чтобы понять, как взаимодействовать с подобным инструментарием в эпоху C++17.

Как это делается

В этом примере мы создадим строки и строковые представления, а затем выполним простые операции конкатенации и преобразования.

1. Как и обычно, сначала включим заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Сначала создадим объекты строк. Самым очевидным способом является инстанцирование объекта класса `string`. Мы контролируем его содержимое, передавая конструктору строку в стиле C (которая после компиляции будет встроена в бинарный файл как статический массив, содержащий символы). Конструктор скопирует ее и сделает содержимым объекта строки `a`. Помимо этого, вместо инициализации строки с помощью строки в стиле C можно применить оператор строкового литерала `""s`. Он создает объект строк динамически. Мы используем его для создания объекта `b`, что позволит применять автоматическое выведение типа.

```
int main()
{
    string a { "a"  };
    auto   b ( "b"s );
```

3. Строки, которые мы только что создали, *копируют* их входные данные из аргумента конструктора в собственный буфер. Чтобы не копировать строку, а просто *сослаться* на нее, можно воспользоваться объектами типа `string_view`. Этот класс также имеет оператор литерала, он вызывается с помощью конструкции `""sv`:

```
    string_view c { "c"   };
    auto        d ( "d"sv );
```

4. Теперь поработаем с нашими строками и строковыми представлениями. Для обоих типов существует перегруженная версия оператора `<<` для класса `std::ostream`, поэтому их удобно выводить на экран:

```
    cout << a << ", " << b << '\n';
    cout << c << ", " << d << '\n';
```

5. В классе `string` имеется перегруженная версия оператора `+`, поэтому можно *сложить* две строки и получить в качестве результата их конкатенацию. Таким образом, выражение `"a" + "b"` даст результат `"ab"`. Конкатенация строк `a` и `b` с помощью данного способа выполняется довольно легко. При необходимости сложить `a` и `c` могут возникнуть некоторые трудности, поскольку `c` — не строка, а экземпляр класса `string_view`. Сначала нужно получить строку из `c`, это делается путем создания новой строки из `c` и сложения ее с `a`. Кто-то может задаться вопросом: «Погодите, зачем копировать `c` в промежуточную строку только для того, чтобы сложить ее с `a`? Можно ли этого избежать, использовав конструкцию `c.data()`?» Идея хороша, но имеет недостаток: экземпляры класса `string_view` не обязаны содержать строки, завершающиеся нулем. Данная проблема может привести к переполнению буфера.

```
    cout << a + b << '\n';
    cout << a + string{c} << '\n';
```

6. Создадим новую строку, содержащую все введенные нами строки и строковые представления. С помощью `std::ostringstream` можно *поместить* любую пере-

менную в объект потока, который ведет себя точно так же, как и `std::cout`, но не выводит данные на консоль. Вместо этого он выводит данные в *строковый буфер*. После того, как мы поместим все переменные в поток, разделив их пробелами с помощью оператора `<<`, можем создать и вывести на экран новый объект строки, задействовав конструкцию `o.str()`.

```
ostringstream o;
o << a << " " << b << " " << c << " " << d;
auto concatenated (o.str());
cout << concatenated << '\n';
```

7. Мы также можем преобразовать эту новую строку путем, например, перевода всех ее символов в верхний регистр. Библиотечная функция C `toupper`, которая соотносит символы в нижнем регистре и символы в верхнем, оставляя другие символы неизменными, уже доступна, и ее можно объединить с функцией `std::transform`, поскольку строка, по сути, представляет собой итерабельный контейнер, содержащий элементы типа `char`.

```
transform(begin(concatenated), end(concatenated),
          begin(concatenated), ::toupper);
cout << concatenated << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат, который полностью оправдывает наши ожидания:

```
$ ./creating_strings
a, b
c, d
ab
ac
a b c d
A B C D
```

Как это работает

Очевидно, строки можно складывать с помощью оператора `+` прямо как числа. Математика здесь не используется, но в итоге мы получаем *сконкатенированные* строки. Чтобы иметь возможность работать с объектами класса `string_view`, сначала нужно выполнить их преобразование к типу `std::string`.

Однако очень важно отметить: при объединении в коде строк и строковых представлений мы никогда не должны предполагать, что хранящаяся в экземпляре класса `string_view` строка завершается *нулевым символом*! Именно поэтому мы используем конструкцию `"abc"s + string{some_string_view}` вместо конструкции `"abc"s + some_string_view.data()`. Кроме того, класс `std::string` предоставляет функцию-член `append`, которая может работать с экземплярами класса `string_view`, но изменяет строку вместо того, чтобы вернуть новую строку, к которой прикреплено содержимое строкового представления.

Класс std::string_view очень полезен, но будьте осторожны при смешивании его со строками и строковыми функциями. Нельзя рассчитывать на то, что эти строки будут оканчиваться нулями, поэтому весь порядок быстро нарушается в стандартном строковом окружении. К счастью, зачастую предусмотрены подходящие перегруженные версии функций, которые могут обрабатывать подобные ситуации.

Сложную конкатенацию строк с форматированием и т. д. не следует выполнять шаг за шагом для экземпляров строк. Классы `std::stringstream`, `std::ostringstream` и `std::istringstream` подходят для этого гораздо лучше, так как позволяют более качественно управлять памятью при объединении строк и предоставляют все возможности по форматированию, характерные для потоков. Для данного раздела мы выбрали класс `std::ostringstream`, поскольку собираемся создать строку вместо того, чтобы ее анализировать. Экземпляр класса `std::istringstream` можно создать на основе существующей строки, которая при необходимости преобразуется в переменные других типов. Если мы хотим объединить обе возможности, то нам подойдет класс `std::stringstream`.

Удаляем пробелы из начала и конца строк

Полученные из пользовательского ввода строки зачастую содержат лишние пробелы. В одном из предыдущих примеров мы удаляли повторяющиеся пробелы между словами.

Взглянем на пробелы, расположенные вокруг строк, и удалим их. Класс `std::string` содержит удобные вспомогательные функции, которые нам помогут.

После рассмотрения данного примера, в котором показано, как работать с простыми строковыми объектами, взгляните также и на следующий пример. Там вы увидите, как избежать ненужного копирования или изменения данных при работе с классом std::string_view.

Как это делается

В этом примере мы напишем вспомогательную функцию, которая определяет наличие пробелов в начале и конце строки и возвращает ее копию, но уже без таких пробелов. Затем немного протестируем ее.

1. Как и обычно, сначала идут заголовочные файлы и директива `using`:

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

using namespace std;
```

2. Наша функция будет принимать константную ссылку на существующую строку. А возвращать станет новую строку, из начала и конца которой удалены лишние пробелы:

```
string trim_whitespace_surrounding(const string &s)
{
```

3. Класс `std::string` предоставляет две удобные функции, которые будут очень полезны. Первая функция — это `string::find_first_not_of`, она принимает строку, содержащую все символы, которые мы хотим опустить. В нашем случае таковыми являются символы пробела `' '`, табуляции `'\t'` и перехода на новую строку `'\n'`. Функция возвращает позицию первого символа, не совпадающего с переданными. При наличии в строке только пробелов она вернет значение `string::npos`. Это значит следующее: если мы удалим все пробелы, то останется только пустая строка. Так что в подобных случаях просто возвращайте пустую строку:

```
    const char whitespace[] {" \t\n"};
    const size_t first (s.find_first_not_of(whitespace));
    if (string::npos == first) { return {}; }
```

4. Теперь мы знаем, где должна начинаться новая строка, но пока неизвестно, где она заканчивается. Поэтому воспользуемся другой полезной функцией строки: `string::find_last_not_of`. Она вернет позицию последнего символа, не являющегося пробелом.

```
    const size_t last (s.find_last_not_of(whitespace));
```

5. С помощью функции `string::substr` мы теперь можем вернуть часть строки, окруженной пробелами, в которой их не будет. Эта функция принимает два параметра: *позицию* от начала строки и *количество символов* после данной позиции.

```
    return s.substr(first, (last - first + 1));
}
```

6. На этом все. Напишем функцию `main`, в которой создадим строку, окружающую предложение всеми видами пробелов, чтобы обрезать ее:

```
int main()
{
    string s {" \t\n string surrounded by ugly"
              " whitespace \t\n "};
```

7. Выведем на экран необрезанную и обрезанную версии строки. Окружив строку скобками, можно показать все пробелы, которые находились в ней до обрезки.

```
    cout << "{" << s << "}\n";
    cout << "{"
         << trim_whitespace_surrounding(s)
         << "}\n";
}
```

8. Компиляция и запуск программы дадут ожидаемый результат:

```
$ ./trim_whitespace
{
  string surrounded by ugly whitespace
   }
{string surrounded by ugly whitespace}
```

Как это работает

В этом разделе мы применили функции `string::find_first_not_of` и `string::find_last_not_of`. Обе принимают строку, созданную в стиле C, в виде списка символов, которые нужно проигнорировать при поиске другого символа. Если у нас есть экземпляр строки, содержащий строку `"foo bar"`, и мы вызовем для него функцию `find_first_not_of("bfo ")`, то она вернет значение `5`, поскольку символ `'a'` — первый символ, который не входит в строку `"bfo"`. Порядок символов в строке-аргументе неважен.

Существуют подобные функции с инвертированной логикой, однако мы не использовали их в этом примере:

`string::find_first_of and string::find_last_of.`

По аналогии с функциями, основанными на итераторах, нужно проверять, возвращают ли эти функции реальную позицию в строке или же значение, которое указывает, что позиция символа, отвечающего условию, *не была найдена*. Если такой позиции нет, то они возвращают значение `string::npos`.

На основе позиций символов, полученных из этих функций, мы в рамках вспомогательной функции создаем подстроку, не содержащую пробелов в начале и конце, с помощью функции `string::substring`. Она принимает относительное смещение и длину строки, а затем возвращает новый экземпляр строки, для которого выделен собственный фрагмент памяти, содержащий только эту подстроку. Например, вызов `string{"abcdef"}.substr(2, 2)` вернет новую строку `"cd"`.

Преимущества использования std::string без затрат на создание объектов std::string

Класс `std::string` очень полезен, поскольку значительно упрощает работу со строками. Его недостаток заключается в том, что при необходимости передать подстроку нужно передавать указатель и переменную, содержащую длину подстроки, два итератора или копию подстроки. Мы делали это в предыдущем примере, когда удаляли лишние пробелы из строки, вернув копию подстроки, которая не содержит их.

Если мы хотим передать строку или подстроку в библиотеку, которая не предоставляет поддержку класса `std::string`, то можем передать только необработанный указатель на строку, что несколько разочаровывает, поскольку этот способ использовался еще во времена C. Как и в случае с выделением подстроки, необработанный

указатель не несет информации о длине строки. Таким образом, кто-то должен будет реализовать связку указателя и длины строки.

Говоря упрощенно, такой конструкцией как раз и является класс `std::string_view`. Он доступен, начиная с версии C++17, и предоставляет способ объединения указателя на некую строку и ее размера. Он воплощает идею наличия ссылочного типа для массивов данных.

Представим, что разрабатываем функции, которые ранее в качестве параметров принимали объекты типа `std::string`, но при этом не изменяли их так, чтобы экземплярам класса `string` потребовалось повторно выделять память, содержащую реальные данные. Мы могли бы использовать тип `std::string_view` для повышения совместимости с библиотеками, которые не знают об STL. Можно позволить другим библиотекам предоставлять `string_view` для строк, содержащих полезные данные, скрытые за сложной реализацией типа `string`, а затем применять их в нашем коде, совместимом с STL. Таким образом, класс `string_view` ведет себя как минималистичный и полезный интерфейс, пригодный для использования многими библиотеками.

Еще одной приятной особенностью является тот факт, что класс `string_view` может применяться как ссылка на подстроки больших объектов класса `string`. Существует много возможностей грамотно использовать эту особенность. В данном разделе мы поработаем с классом `string_view`, чтобы получить представление о его преимуществах и недостатках. Кроме того, увидим, как можно скрыть пробелы в начале и конце строки путем адаптации строковых представлений, а не изменения или копирования самой строки. Этот метод позволяет избежать ненужного копирования или изменения данных.

Как это делается

В этом примере мы реализуем функцию, которая работает с особенностями класса `string_view`, а затем увидим, сколько разных типов можем ей передать.

1. Сначала указываем заголовочные файлы и директивы `using`:

```
#include <iostream>
#include <string_view>

using namespace std;
```

2. Реализуем функцию, которая принимает в качестве единственного аргумента объект типа `string_view`:

```
void print(string_view v)
{
```

3. Прежде чем сделать что-то с входной строкой, удалим все пробелы из ее начала и конца. Мы будем изменять не строку, а ее *представление*, сузив его до значащей части. Функция `find_first_not_of` найдет первый символ строки, который не является пробелом (`' '`), символом табуляции (`'\t'`) или символом перехода

на новую строку (`'\n'`). С помощью функции `remove_prefix` мы переместим внутренний указатель класса `string_view` на первый символ, не являющийся пробелом. В том случае, если строка содержит только пробелы, функция `find_first_not_of` вернет значение `npos`, которое равно `size_type(-1)`. Поскольку `size_type` — беззнаковая переменная, мы получим очень большое число. Поэтому выберем меньшее число из полученных: `words_begin` или размер строкового представления:

```
const auto words_begin (v.find_first_not_of(" \t\n"));
v.remove_prefix(min(words_begin, v.size()));
```

4. То же самое сделаем с пробелами в конце строки. Функция `remove_suffix` уменьшает переменную, показывающую размер строкового представления:

```
const auto words_end (v.find_last_not_of(" \t\n"));
if (words_end != string_view::npos) {
    v.remove_suffix(v.size() - words_end - 1);
}
```

5. Теперь можно вывести на экран строковое представление и его длину:

```
    cout << "length: " << v.length()
         << " [" << v << "]\n";
}
```

6. В функции `main` воспользуемся новой функцией `print`, передав ей разные типы аргументов. Сначала передадим ей во время выполнения строку `char*` из указателя `argv`. Во время выполнения программы он будет содержать имя нашего исполняемого файла. Затем передадим ей пустой объект `string_view`. Далее передадим ей символьную строку, созданную в стиле C, а также строку, образованную с помощью литерала `""sv`, который динамически создаст объект типа `string_view`. И наконец, передадим ей объект класса `std::string`. Положительный момент заключается в следующем: ни один из данных аргументов не изменяется и не копируется, чтобы вызвать функцию `print`. Не происходит выделения памяти в куче. Для большого количества строк и/или для длинных строк это очень эффективно.

```
int main(int argc, char *argv[])
{
    print(argv[0]);
    print({});
    print("a const char * array");
    print("an std::string_view literal"sv);
    print("an std::string instance"s);
```

7. Мы не протестировали функцию удаления пробелов. Передадим ей строку, которая содержит множество пробелов в начале и в конце:

```
    print(" \t\n foobar \n \t ");
```

8. Еще одна приятная особенность класса `string_view`: он позволяет создавать строки, *не завершающиеся нулевым символом*. Если мы введем строку, напри-

мер "abc", которая не будет заканчиваться нулем, то функция print сможет безопасно ее обработать, поскольку объект класса string_view также содержит размер строки, на которую указывает:

```
    char cstr[] {'a', 'b', 'c'};
    print(string_view(cstr, sizeof(cstr)));
}
```

9. Компиляция и запуск программы дадут следующий результат. Все строки обработаны корректно. Строка, которую мы заполнили большим количеством пробелов в начале и конце, также была корректно отфильтрована, а строка abc, не имеющая завершающего нулевого символа, корректно выведена на экран, не вызвав переполнения буфера:

```
$ ./string_view
length: 17 [./string_view]
length: 0 []
length: 20 [a const char * array]
length: 27 [an std::string_view literal]
length: 23 [an std::string instance]
length: 6 [foobar]
length: 3 [abc]
```

Как это работает

Мы только что увидели следующее: можно вызвать функцию, принимающую аргумент типа string_view, который способен содержать все, что похоже на строку, — а именно символы, сохраненные непрерывным способом. Во время вызовов нашей функции print не было выполнено *ни одной операции копирования*.

Интересно отметить, что в вызове print(argv[0]) строковое представление автоматически определило длину строки, поскольку данная строка по соглашению завершается нулевым символом. С другой стороны, никто не может предполагать, что можно определить длину поля данных объекта типа string_view путем подсчета символов до тех пор, пока не встретится нулевой символ. Поэтому нужно всегда соблюдать осторожность при работе с указателями на данные строкового представления с помощью string_view::data(). Обычные строковые функции предполагают, что строка будет завершаться нулевым символом, и поэтому использование необработанных указателей способно привести к переполнению буфера. Всегда лучше применять интерфейсы, которые ожидают передачи строкового представления.

За исключением этой особенности, у нас имеется роскошный интерфейс, с которым мы уже знакомы благодаря классу std::string.

Задействуйте класс std::string_view для передачи строк или подстрок, чтобы избежать копирования или выделения памяти кучей, продолжая при этом привычно использовать строковые классы. Но помните: класс std::string_view не предполагает, что строки завершаются нулевым символом.

Считываем значения из пользовательского ввода

Во многих примерах нашей книги значения поступают из входного источника, которым является стандартный поток ввода или файл, а затем с ними выполняются некие действия. В этот раз мы сконцентрируемся лишь на чтении и обработке ошибок, что становится важным, если чтение каких-то данных из потока завершилось *неудачей* и нужно обработать возникшую ситуацию вместо того, чтобы завершать работу программы.

В следующем примере мы лишь считаем данные от пользователя, но тщательно разобрав этот процесс, вы научитесь читать данные из любого другого потока. Пользовательские данные можно считать с помощью `std::cin` — по сути, это объект потока ввода, как и экземпляры классов `ifstream` и `istringstream`.

Как это делается

В этом примере мы считаем пользовательские данные в разные переменные и увидим, как обрабатывать ошибки, а также научимся выполнять более сложную токенизацию входных данных, чтобы разбить их на полезные фрагменты.

1. На сей раз нам понадобится только `iostream`. Так что включим этот единственный заголовочный файл и объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>

using namespace std;
```

2. Пригласим пользователя ввести два числа. Поместим их в переменные типов `int` и `double`. Пользователь может разделить их пробелами. Например, `1 2.3` — это корректные входные данные.

```
int main()
{
    cout << "Please Enter two numbers:\n> ";
    int x;
    double y;
```

3. Анализ и проверка ошибок выполняются одновременно в условной части блока `if`. Выведем числа на экран только в том случае, если их можно проанализировать и они имеют смысл:

```
    if (cin >> x >> y) {
        cout << "You entered: " << x
             << " and " << y << '\n';
```

4. Если по какой-то причине анализ завершился ошибкой, то мы скажем пользователю об этом. Объект потока `cin` теперь находится в *состоянии ошибки*

и не будет давать другие данные до тех пор, пока мы не избавимся от этого состояния. Чтобы иметь возможность проанализировать новые входные данные после ошибки, вызываем метод `cin.clear()` и отбрасываем все входные данные, полученные ранее. Отбрасывание выполняется с помощью `cin.ignore`, где мы указываем, что нужно отбросить максимальное количество символов до тех пор, пока не встретим символ новой строки (который тоже будет отброшен). После этого символа снова начинаются интересные входные данные:

```
    } else {
        cout << "Oh no, that did not go well!\n";
        cin.clear();
        cin.ignore(
            std::numeric_limits<std::streamsize>::max(),
            '\n');
    }
```

5. Теперь запросим еще какие-нибудь входные данные. Мы позволим пользователю вводить имена. Они могут состоять из нескольких слов, разделенных пробелами, поэтому символ пробела не подходит в качестве разделителя. Так что воспользуемся `std::getline`, принимающим объектом потока наподобие `cin`, ссылкой на строку, в которую будут скопированы входные данные, и символом-разделителем (таковым выступит запятая `','`). Задействуя вместо `cin` конструкцию `cin >> ws` в качестве параметра потока для `getline`, можно дать `cin` команду отбросить пробелы, стоящие перед именами. На каждом шаге цикла выводим текущее имя, но если оно пустое, то завершим цикл:

```
    cout << "now please enter some "
            "comma-separated names:\n> ";
    for (string s; getline(cin >> ws, s, ',');) {
        if (s.empty()) { break; }
        cout << "name: \"" << s << "\"\n";
    }
}
```

6. Компиляция и запуск программы дадут следующий результат (для него предполагается введение только корректных значений). Мы ввели числа `"1 2"`, они были преобразованы корректно, а затем ввели некоторые имена, также корректно выведенные на экран. Пустое имя, полученное в виде двух запятых, расположенных рядом друг с другом, завершает цикл:

```
$ ./strings_from_user_input
Please Enter two numbers:
> 1 2
You entered: 1 and 2
now please enter some comma-separated names:
> john doe,  ellen ripley,       alice,    chuck norris,,
name: "john doe"
name: "ellen ripley"
name: "alice"
name: "chuck norris"
```

7. При повторном запуске программы и вводе некорректных цифр в самом начале мы увидим, что программа корректно выбирает другую ветвь, отбрасывает некорректные данные и продолжает работу с помощью слушателя имен. Попробуйте использовать строки `cin.clear()` и `cin.ignore(...)`, чтобы увидеть, как это пересекается с кодом чтения имен:

```
$ ./strings_from_user_input
Please Enter two numbers:
> a b
Oh no, that did not go well!
now please enter some comma-separated names:
> bud spencer, terence hill,,
name: "bud spencer"
name: "terence hill"
```

Как это работает

В этом разделе мы выполнили сложную операцию получения входных данных. Первое, на что следует обратить внимание: получение данных и проверка на ошибки всегда происходили одновременно.

Результатом выполнения выражения `cin >> x` является ссылка на `cin`. Таким образом можно создавать конструкции наподобие `cin >> x >> y >> z >> ...`. В то же время можно преобразовать данные к булевым значениям, используя их в булевых контекстах, таких как условие `if`. Булево значение говорит нам, была ли корректной последняя операция чтения. Вот почему можно создавать конструкции, аналогичные выражению `if (cin >> x >> y){...}`.

Если мы, например, попробуем считать целое число, но во входных данных следующим является токен `"foobar"`, то преобразование этого значения к типу `int` выполнить нельзя и объект потока входит в *ошибочное состояние*. Это критически важно только для попытки преобразования, но не для всей программы. Можно совершенно спокойно сбросить ее и попробовать сделать что-то еще. В нашем примере мы попробовали считать список имен после потенциально ошибочной попытки считать два числа. При неудачной попытке считывания числа мы используем функцию `cin.clear()` с целью вернуть `cin` в рабочее состояние. Но после этого его внутренний курсор все еще находится в той позиции, где лежат данные, которые мы ввели вместо чисел. Чтобы отбросить старые входные данные и очистить канал для ввода имен, мы применили очень длинное выражение, `cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');`. Необходимо удалять все, что находилось в буфере, поскольку он нужен абсолютно пустым, когда мы запросим у пользователя список имен.

Следующий цикл также может показаться странным на первый взгляд:

```
for (string s; getline(cin >> ws, s, ',');) { ... }
```

В условной части цикла мы использовали функцию `getline`. Она принимает объект потока ввода, ссылку на строку в качестве выходного параметра, а также символ-разделитель. По умолчанию таковым является символ перехода на новую

строку. Здесь мы указали, что в роли этого символа выступает запятая (`,`), чтобы все имена из списка наподобие `"john, carl, frank"` считывались отдельно.

Пока все идет по плану. Но что означает предоставление функции `cin >> ws` в качестве объекта потока? Это заставляет `cin` отбросить все пробелы, которые стоят перед следующим символом, не являющимся пробелом, а также в конце строки. Если бы мы не применили `ws`, то из строки `"john, carl, frank"` получили бы подстроки `"john"`, `" carl"` и `" frank"`. Заметили лишние пробелы для имен `carl` и `frank`? Они пропадают, поскольку мы использовали `ws`.

Подсчитываем все слова в файле

Предположим, мы считали текстовый файл и хотим определить количество слов в тексте. Словом мы называем диапазон символов, расположенный между пробелами. Как же решить эту задачу?

Например, можно подсчитать количество пробелов, поскольку между словами должны быть пробелы. В предложении `"John has a funny little dog."` пять символов пробела, поэтому можно сказать, что оно состоит из шести слов.

Но как быть с предложением, содержащим разные виды пробелов, например: `" John has \t a\nfunny little dog ."`? В нем содержится слишком много ненужных пробелов и не только. Из других примеров данной книги вы уже знаете, как удалить эти лишние пробелы. Так что сначала можно было бы предварительно обработать строку, преобразовав ее в обычное предложение, а затем применить стратегию подсчета пробелов. Это выполнимо, но существует *гораздо более простой* способ. Почему бы не воспользоваться теми возможностями, которые предоставляет STL?

Помимо нахождения элегантного решения этой проблемы, мы позволим пользователю решать, для каких именно входных данных считать количество слов — для данных из стандартного потока ввода или из текстового файла.

Как это делается

В этом примере мы напишем короткую функцию, которая считает количество слов из входного буфера и позволяет пользователю выбирать, откуда именно во входном буфере появятся данные.

1. Включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

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

using namespace std;
```

2. Функция `wordcount` принимает поток ввода, например `cin`. Она создает итератор `std::input_iterator`, который токенизирует строки потока, а затем передает их в `std::distance`. Параметр `distance` принимает в качестве аргументов два итератора и пытается определить, сколько именно операций инкремента нужно выполнить, чтобы переместиться с одной позиции итератора в другую. Для *итераторов с произвольным доступом* это делается легко, поскольку они реализуют операцию математической разности (`operator-`). Такие итераторы можно вычитать друг из друга, как и другие указатели. Итератор `istream_iterator`, однако, является *однонаправленным*, и его нужно сдвигать вперед до тех пор, пока он не станет равен итератору `end`. В конечном счете количество шагов будет равно количеству слов.

```
template <typename T>
size_t wordcount(T &is)
{
    return distance(istream_iterator<string>{is}, {});
}
```

3. В нашей функции `main` мы позволяем пользователю выбрать, откуда придет поток ввода — из `std::cin` или из файла:

```
int main(int argc, char **argv)
{
    size_t wc;
```

4. Если пользователь запустит программу в оболочке и укажет имя файла (например, `$ ./count_all_words some_textfile.txt`), то мы сможем получить его из параметра командной строки `argv` и открыть, чтобы передать новый поток ввода в функцию `wordcount`:

```
    if (argc == 2) {
        ifstream ifs {argv[1]};
        wc = wordcount(ifs);
```

5. В случае запуска пользователем программы без параметров мы предполагаем, что входные данные появятся из стандартного потока ввода:

```
    } else {
        wc = wordcount(cin);
    }
```

6. На этом все, просто выведем количество слов, сохраненное в переменной `wc`:

```
    cout << "There are " << wc << " words\n";
};
```

7. Скомпилируем и запустим программу. Сначала передадим данные из стандартного потока ввода. Можно либо выполнить вызов `echo`, передав туда несколько слов, либо запустить программу и ввести несколько слов с клавиатуры. Во втором случае можно прервать ввод нажатием комбинации клавиш Ctrl+D. Так выглядит вызов `echo`:

```
$ echo "foo bar baz" | ./count_all_words
There are 3 words
```

8. Если запустить программу и передать ей в качестве входного файла ее файл с исходным кодом, то она подсчитает количество слов, которые содержатся в нем:

```
$ ./count_all_words count_all_words.cpp
There are 61 words
```

Как это работает

Здесь особо нечего добавить, большую часть программы мы уже объяснили при реализации, поскольку сама программа очень короткая. Единственный факт, который можно немного пояснить: `std::cin` и `std::ifstream` взаимозаменяемы. `cin` имеет тип `std::istream`, а `std::ifstream` наследует от `std::istream`. Взгляните на диаграмму наследования, приведенную в начале этой главы (см. рис. 7.1). Таким образом, они полностью взаимозаменяемы, даже во время работы программы.

Поддерживайте модульность кода путем использования абстракций потока. Это позволит отвязать фрагменты исходного кода друг от друга и облегчит тестирование исходного кода, поскольку можно внедрить любой другой соответствующий тип потока.

Форматируем ваши выходные данные с помощью манипуляторов потока ввода-вывода

Во многих случаях недостаточно просто вывести строки и числа. Иногда числа нужно вывести в десятичной системе счисления, иногда — в шестнадцатеричной, а иногда — в восьмеричной. В одних ситуациях перед шестнадцатеричными числами префикс `"0x"` нужен, а в других — нет.

При выводе чисел с плавающей точкой существует множество моментов, на которые можно повлиять. Нужно ли всегда выводить их с одинаковой точностью? Надо ли выводить их вообще? Или, может быть, требуется научное представление?

Помимо представления и системы счисления нужно также представить пользователю выходные данные в «аккуратном» виде. Некоторые выходные данные можно поместить, например, в таблицы, чтобы сделать их максимально читабельными.

Все это можно сделать с помощью потоков вывода. Некоторые из этих настроек важны и при преобразовании входных значений. В данном примере мы познакомимся с так называемыми *манипуляторами ввода-вывода*. Часть их могут показаться сложными, так что рассмотрим их подробнее.

Как это делается

В этом примере мы выведем на экран числа, используя совершенно разные настройки форматов, чтобы ознакомиться с манипуляторами ввода-вывода.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <locale>

using namespace std;
```

2. Далее определим вспомогательную функцию, выводящую на экран целое число в разных стилях. Она принимает ширину отступа и символ-заполнитель, которым по умолчанию является `' '`:

```
void print_aligned_demo(int val,
                        size_t width,
                        char fill_char = ' ')
{
```

3. С помощью `setw` можно задать минимальное количество символов, которое нужно вывести. Например, при вводе числа `123` с шириной `6` получим `" 123"` или `" 123"`. Можно управлять тем, с какой стороны будут вставлены символы-заполнители, используя `std::left`, `std::right` и `std::internal`. При выводе чисел в десятичной форме манипулятор `internal` выглядит похожим на `right`. Но если мы выведем значение `0x1`, например с шириной `6` и манипулятором `internal`, то получим `"0x 6"`. Манипулятор `setfill` определяет, какой именно символ будет применен для заполнения. Мы попробуем разные стили.

```
    cout << "================\n";
    cout << setfill(fill_char);
    cout << left << setw(width) << val << '\n';
    cout << right << setw(width) << val << '\n';
    cout << internal << setw(width) << val << '\n';
}
```

4. В функции `main` начинаем использовать реализованную нами функцию. Сначала выведем на экран значение `12345` с шириной `15`. Сделаем это дважды, но во второй раз применим заполнитель `'_'`.

```
int main()
{
    print_aligned_demo(123456, 15);
    print_aligned_demo(123456, 15, '_');
```

5. Затем выведем значение `0x123abc` с такой же шириной. Однако прежде, чем это сделать, применим `std::hex` и `std::showbase` с целью указать объекту потока вывода `cout`, что он должен выводить числа в шестнадцатеричном формате и до-

бавлять к ним префикс "0x", поскольку тогда их нельзя будет интерпретировать по-другому:

```
cout << hex << showbase;
print_aligned_demo(0x123abc, 15);
```

6. Сделаем то же самое и для oct, что укажет cout использовать восьмеричную систему счисления при выводе чисел. showbase все еще активен, поэтому 0 будет добавлен к каждому выводимому числу:

```
cout << oct;
print_aligned_demo(0123456, 15);
```

7. В случае использования hex и uppercase мы увидим, что символ 'x' в конструкции "0x" будет выведен в верхнем регистре. Сочетание 'abc' в конструкции '0x123abc' также будет в верхнем регистре:

```
cout << "A hex number with upper case letters: "
     << hex << uppercase << 0x123abc << '\n';
```

8. Если мы хотим снова вывести число 100 в десятичной системе счисления, то нужно запомнить, что мы переключили поток в режим hex. С помощью dec вернем его в обычное состояние:

```
cout << "A number: " << 100 << '\n';
cout << dec;
cout << "Oops. now in decimal again: " << 100 << '\n';
```

9. Мы также можем сконфигурировать формат вывода булевых значений. По умолчанию значение true выводится как 1, а false — как 0. С помощью boolalpha можно задать ему текстовое представление:

```
cout << "true/false values: "
     << true << ", " << false << '\n';
cout << boolalpha
     << "true/false values: "
     << true << ", " << false << '\n';
```

10. Взглянем на переменные с плавающей точкой типов float и double. Если нужно вывести число наподобие 12.3, то увидим 12.3. При наличии такого числа, как 12.0, поток вывода отбросит десятичную точку. Это можно изменить, использовав showpoint. С его помощью десятичная точка будет отображаться всегда:

```
cout << "doubles: "
     << 12.3 << ", "
     << 12.0 << ", "
     << showpoint << 12.0 << '\n';
```

11. Представление чисел с плавающей точкой может иметь модификаторы scientific или fixed. Первый означает следующее: число *нормализовано* к такому виду, что видна только первая цифра после десятичной точки, а затем выводится экспонента, на которую нужно умножить число, чтобы получить его реальный размер. Например, значение 300.0 будет выглядеть как "3.0E2", поскольку

300 равно 3.0 * 10^2. Модификатор fixed позволяет вернуться к представлению с десятичной точкой:

```
    cout << "scientific double: " << scientific
         << 123000000000.123 << '\n';
    cout << "fixed      double: " << fixed
         << 123000000000.123 << '\n';
```

12. Помимо нотации мы можем решить, какую точность будут иметь числа с плавающей точкой. Создадим очень маленькое значение и выведем его, указав точность десять знаков после запятой, а затем повторим вывод, но укажем точность один знак:

```
    cout << "Very precise double: "
         << setprecision(10) << 0.0000000001 << '\n';
    cout << "Less precise double: "
         << setprecision(1)  << 0.0000000001 << '\n';
}
```

13. Компиляция и запуск программы дают следующий длинный результат. Первые четыре блока выходных данных получены от вспомогательной функции print с помощью модификаторов setw и left/right/internal. Затем мы работали с регистрами представлений системы счисления, представлениями булевых чисел, а также форматированием чисел с плавающей точкой. Вам стоит поработать со всеми этими возможностями, чтобы получше ознакомиться с ними.

```
$ ./formatting
================
123456
         123456
         123456
================
123456_________
_________123456
_________123456
================
0x123abc
       0x123abc
0x       123abc
================
0123456
        0123456
        0123456
A hex number with upper case letters: 0X123ABC
A number: 0X64
Oops. now in decimal again: 100
true/false values: 1, 0
true/false values: true, false
doubles: 12.3, 12, 12.0000
scientific double: 1.230000E+11
fixed      double: 123000000000.123001
Very precise double: 0.0000000001
Less precise double: 0.0
```

Как это работает

Все эти `stream` выражения, `<< foo << bar`, иногда довольно длинные и способны запутать читателя: ему может быть непонятно, что именно делает каждое из них. Поэтому взглянем на таблицу, в которой приведены существующие модификаторы форматов (табл. 7.1). Эти модификаторы нужно помещать в выражение `input_stream >> modifier` или `output_stream << modifier`, в этом случае они будут влиять на входные или выходные данные.

Таблица 7.1

Символ	Значение
setprecision(int n)	Определяет точность при выводе или преобразовании значений с плавающей точкой
showpoint/ noshowpoint	Включает или отключает десятичную точку для чисел с плавающей точкой, даже если они не имеют десятичных разрядов
fixed/scientific/ hexfloat/defaultfloat	Числа могут быть выведены в фиксированном (самый интуитивный) или научном представлении. Для этих режимов используются модификаторы fixed и scientific. hexfloat активизирует оба режима и форматирует числа с плавающей точкой в шестнадцатеричном представлении с плавающей точкой. Модификатор defaultfloat отключает оба режима
showpos/noshowpos	Включает или отключает префикс '+ ' для положительных значений с плавающей точкой
setw(int n)	Считывает или записывает ровно n символов. При чтении обрезает входные данные. При выводе данных на экран применяется заполнитель, если выходные данные короче n символов
setfill(char c)	При использовании заполнителя (см. setw) заполняет выходные данные символьными значениями c. По умолчанию применяются пробелы (' ')
internal/left/right	left и right управляют тем, с какой стороны появляется заполнитель для вывода информации с фиксированной шириной (см. setw). Модификатор internal помещает символы-заполнители между целыми числами и знаком минуса, префиксом hex и шестнадцатеричным значением или валютой и численным значением
dec/hex/oct	Целочисленные значения могут быть выведены и преобразованы в десятичной, шестнадцатеричной и восьмеричной системах счисления
setbase(int n)	Эта численная функция аналогична dec/hex/oct, если передать в нее числа 10/16/8. Другие числа сбрасывают выбор системы счисления в 0. Это приводит к тому, что вы снова видите десятичные числа, или к преобразованию данных на основе префикса
quoted(string)	Выводит строку в кавычках или преобразует данные, стоящие в кавычках, а затем отбрасывает кавычки. Передаваемая строка может быть объектом класса String или строкой, созданной в стиле C

Продолжение ⇩

Таблица 7.1 *(продолжение)*

Символ	Значение
boolalpha/noboolalpha	Выводит на экран или преобразует булевы значения в буквенное представление или из него, а не в строки 1/0
showbase/noshowbase	Включает или отключает отображение префиксов системы счисления при выводе или преобразовании чисел. Для шестнадцатеричной системы счисления используется префикс 0x; для восьмеричной — 0
uppercase/nouppercase	Включает или отключает применение верхнего регистра или символы алфавита при выводе чисел с плавающей точкой или шестнадцатеричных значений

Самый лучший способ познакомиться с этими модификаторами — изучить все их многообразие и немного поработать с ними.

При взаимодействии, однако, мы уже могли заметить, что большинство модификаторов являются *стойкими*. Термин «стойкий» означает следующее: после применения они станут влиять на входные/выходные данные *до тех пор*, пока не будут сброшены. Единственные нестойкие модификаторы в табл. 7.1 — `setw` и `quoted`. Они влияют только на следующий элемент входных/выходных данных. Это важно знать, поскольку при выводе неких данных с определенным форматированием нужно сбрасывать настройки форматирования объекта потока, так как следующий блок выходных данных, создаваемый несвязанным кодом, может выглядеть странно. Это же верно и для преобразования входных данных, где правильный ход программы может быть нарушен из-за неверных настроек манипулятора ввода/вывода.

Мы не использовали следующие манипуляторы, поскольку они никак не связаны с форматированием, но для полноты картины рассмотрим и их (табл. 7.2).

Таблица 7.2

Символ	Значение
skipws/noskipws	Включает или отключает возможность пропускать лишние пробелы
unitbuf/nounitbuf	Включает или отключает промежуточный буфер, который сбрасывает данные после любой другой операции вывода
ws	Может использоваться для потоков ввода, чтобы пропускать все пробелы, стоящие в начале потока
ends	Записывает в поток завершающий символ '\0'
flush	Мгновенно сбрасывает все, что находится в выходном буфере
endl	Добавляет символ \n в поток вывода, сбрасывает выходные данные

Среди перечисленных модификаторов стойкими являются только `skipws`/`noskipws` и `unitbuf`/`nounitbuf`.

Инициализируем сложные объекты из файла вывода

Считывать отдельные числа и слова довольно просто, поскольку оператор >> имеет перегруженные версии для всех этих типов, а потоки ввода удобным образом отбрасывают все промежуточные пробелы.

Но что, если перед нами более сложная структура и нужно прочесть ее из потока ввода или же требуется считать строки, состоящие более чем из одного слова (по умолчанию они будут разбиты на отдельные слова из-за того, что пробелы опускаются)?

Для любого типа можно предоставить еще одну перегруженную версию оператора потока ввода >>, и сейчас мы увидим воплощение этого на практике.

Как это делается

В этом примере мы определим пользовательскую структуру данных и предоставим возможности для чтения ее объектов из потоков ввода.

1. Включим некоторые заголовочные файлы и объявим об использовании пространства имен `std` для удобства:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;
```

2. В качестве примера сложного объекта определим структуру `city`. Она будет иметь название, количество населения и географические координаты:

```
struct city {
    string name;
    size_t population;
    double latitude;
    double longitude;
};
```

3. Чтобы считать объект такой структуры из последовательного потока ввода, следует перегрузить оператор потоковой функции >>. В этом операторе сначала опустим все пробелы, стоящие перед текстом, с помощью `ws`, поскольку пробелы не должны засорять название города. Затем считаем целую строку текста. Это подразумевает, что входной файл будет включать строку, в которой записано только название города. Затем, после символа новой строки, будет

следовать список чисел, разделенный запятой, в котором содержится информация о численности населения, а также географическая широта и долгота:

```
istream& operator>>(istream &is, city &c)
{
    is >> ws;
    getline(is, c.name);
    is >> c.population
       >> c.latitude
       >> c.longitude;
    return is;
}
```

4. В нашей функции `main` создаем вектор, в котором может содержаться целый диапазон элементов типа `city`. Заполним его с помощью `std::copy`. Входными данными для вызова `copy` является диапазон `istream_iterator`. Передавая ему тип структуры `city` в качестве параметра шаблона, мы будем использовать перегруженную функцию >>, реализованную только что:

```
int main()
{
    vector<city> l;
    copy(istream_iterator<city>{cin}, {},
         back_inserter(l));
```

5. Чтобы увидеть, прошло ли преобразование правильно, выведем на экран содержимое списка. Форматирование ввода/вывода, `left << setw(15) <<`, приводит к тому, что название города заполняется пробелами, поэтому выходные данные представлены в приятной и удобочитаемой форме:

```
    for (const auto &[name, pop, lat, lon] : l) {
        cout << left << setw(15) << name
             << " population=" << pop
             << " lat=" << lat
             << " lon=" << lon << '\n';
    }
}
```

6. Текстовый файл, из которого наша программа будет считывать данные, выглядит следующим образом. В нем содержится информация о четырех городах: их названия, количество населения и географические координаты:

```
Braunschweig
250000 52.268874 10.526770
Berlin
4000000 52.520007 13.404954
New York City
8406000 40.712784 -74.005941
Mexico City
8851000 19.432608 -99.133208
```

7. Компиляция и запуск программы дадут следующий результат, он соответствует нашим ожиданиям. Попробуйте изменить входной файл, добавляя

ненужные пробелы перед названием городов, чтобы увидеть, как они будут отфильтрованы:

```
$ cat cities.txt  | ./initialize_complex_objects
Braunschweig    population=250000 lat=52.2689 lon=10.5268
Berlin          population=4000000 lat=52.52 lon=13.405
New York City   population=8406000 lat=40.7128 lon=-74.0059
Mexico City     population=8851000 lat=19.4326 lon=-99.1332
```

Как это работает

Мы снова рассмотрели короткий пример. В нем мы лишь создали новую структуру `city`, а затем перегрузили оператор `>>` итератора `std::istream` для данного типа. Это позволило десериализовать элементы типа `city`, полученные из стандартного потока ввода, с помощью `istream_iterator<city>`.

Открытым может оставаться вопрос, связанный с проверкой на ошибки. Поэтому снова рассмотрим реализацию оператора `>>`:

```
istream& operator>>(istream &is, city &c)
{
    is >> ws;
    getline(is, c.name);
    is >> c.population >> c.latitude >> c.longitude;
    return is;
}
```

Мы считываем множество разных элементов. Что произойдет, если один из них даст сбой, а следующий за ним — нет? Означает ли это потенциальное считывание всех следующих элементов со «смещением» в потоке токенов? Нет, этого не произойдет. Если хотя бы один элемент потока ввода не сможет быть преобразован, то объект потока ввода входит в ошибочное состояние и отказывается выполнять дальнейшие преобразования. Это значит, что если, например, `c.population` или `c.latitude` не могут быть преобразованы, то остальные операнды `>>` будут просто отброшены и мы покинем область действия функции оператора с наполовину заполненным объектом.

На вызывающей стороне мы получим оповещение об этом при написании конструкции `if (input_stream >> city_object)`. Такое потоковое выражение неявно преобразуется в булево значение, когда используется как условное выражение. Оно возвращает значение `false`, если объект потока ввода находится в ошибочном состоянии. Зная это, можно сбросить поток и выполнить подходящие операции.

В данном примере мы не писали подобных условий `if` сами, поскольку позволили выполнить десериализацию итератору `std::istream_iterator<city>`. Реализация перегруженной версии оператора `++` для этого итератора также выполняет проверку ошибок во время преобразования. При генерации ошибок преобразование будет приостановлено. В этом состоянии проверка будет возвращать значение `true` при сравнении с конечным итератором, что заставляет алгоритм `copy` завершить работу. Таким образом, мы в безопасности.

Заполняем контейнеры с применением итераторов std::istream

В предыдущем примере вы узнали, как можно собрать сложные структуры из потока ввода, а затем заполнить списки или векторы полученными элементами.

В этот раз мы усложним задачу, заполняя на основе стандартного потока ввода контейнер `std::map`. Проблема заключается в том, что мы не можем просто заполнить отдельную структуру значениями и отправить ее в линейный контейнер наподобие списка или вектора, поскольку в ассоциативных массивах полезная нагрузка распределяется между ключом и значением. Однако, как вы увидите, само решение задачи отличается незначительно.

После изучения этого примера вы сможете легко выполнять сериализацию и десериализацию сложных структур данных.

Как это делается

В этом примере мы определим другую структуру, как делали в прошлом примере, но на сей раз заполним элементами данной структуры ассоциативный массив, что несколько усложняет задачу, поскольку этот контейнер соотносит ключи и значения вместо размещения всех значений в списке.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <map>
#include <iterator>
#include <algorithm>
#include <numeric>

using namespace std;
```

2. Мы хотим создать небольшую базу данных интернет-мемов. Предположим, мем имеет название, описание и год, в который он родился или был создан. Сохраним их в контейнере `std::map`, где название выступит в качестве ключа, а другая информация будет помещена в структуру как значение, связанное с ним:

```
struct meme {
    string description;
    size_t year;
};
```

3. Сначала проигнорируем ключ и просто реализуем перегруженную функцию потокового оператора `>>` для структуры `meme`. Предположим, что описание мема окружено кавычками, за ним следует год. Это будет выглядеть так: `"some description" 2017`. Описание окружено кавычками, так что может содержать

пробелы, поскольку мы знаем, что все символы, стоящие между кавычками, принадлежат описанию. После чтения с помощью конструкции `is >> quoted(m.description)` кавычки автоматически используются как разделители и впоследствии отбрасываются, что очень удобно. Сразу после этого считываем число, которое представляет собой год:

```
istream& operator>>(istream &is, meme &m) {
    return is >> quoted(m.description) >> m.year;
}
```

4. О'кей, теперь примем в расчет название мема в качестве ключа ассоциативного массива. Чтобы вставить мем в массив, нужно создать объект типа `std::pair<тип_ключа, тип_значения>`. Типом ключа является `string`, а типом значения — `meme`. В названии мема также могут находиться пробелы, поэтому мы используем ту же оболочку, которую применяли для описания. `p.first` — это название, а `p.second` — целая структура `meme`, связанная с ним. Данный объект будет передан другой реализации оператора `>>`, которую мы только что создали:

```
istream& operator >>(istream &is,
                     pair<string, meme> &p) {
    return is >> quoted(p.first) >> p.second;
}
```

5. На этом все. Напишем функцию `main`, в которой будет создаваться ассоциативный массив, и заполним его. Поскольку мы переопределили оператор `>>`, итератор `istream_iterator` может работать с данным типом непосредственно. Мы позволим ему десериализовать наши объекты типа `meme`, полученные из стандартного потока ввода, и используем итератор вставки, чтобы поместить их в ассоциативный массив:

```
int main()
{
    map<string, meme> m;
    copy(istream_iterator<pair<string, meme>>{cin},
         {},
         inserter(m, end(m)));
```

6. Прежде чем вывести на экран все, что у нас есть, сначала найдем *самое длинное* название мема в ассоциативном массиве. Для этого воспользуемся алгоритмом `std::accumulate`. Он получит исходное значение `0u` (`u` расшифровывается как unsigned — «беззнаковый») и пройдет по всем элементам ассоциативного массива, чтобы *слить* их воедино. С точки зрения алгоритма `accumulate` слияние обычно означает *сложение*. В нашем случае требуется не численная *сумма*, а размер самой длинной строки. Для получения этого значения предоставим алгоритму вспомогательную функцию `max_func`, которая принимает переменную, содержащую текущий максимальный размер строки (она должна быть беззнаковой, поскольку длина строки знака не имеет), и сравнивает ее с длиной

названия текущего мема, чтобы взять максимальное значение. Это делается для каждого элемента. Итоговое возвращаемое значение функции `accumulate` представляет собой максимальную длину названия мема:

```
auto max_func ([](size_t old_max,
                  const auto &b) {
    return max(old_max, b.first.length());
});
size_t width {accumulate(begin(m), end(m),
                         0u, max_func)};
```

7. Теперь быстро пройдем по ассоциативному массиву и выведем каждый элемент. Чтобы выходные данные смотрелись более «аккуратно», воспользуемся конструкцией `<< left << setw(width)`:

```
    for (const auto &[meme_name, meme_desc] : m) {
        const auto &[desc, year] = meme_desc;
        cout << left << setw(width) << meme_name
             << " : " << desc
             << ", " << year << '\n';
    }
}
```

8. На этом все. Нам нужна небольшая база данных интернет-мемов, так что заполним текстовый файл некоторыми примерами:

```
"Doge" "Very Shiba Inu. so dog. much funny. wow." 2013
"Pepe" "Anthropomorphic frog" 2016
"Gabe" "Musical dog on maximum borkdrive" 2016
"Honey Badger" "Crazy nastyass honey badger" 2011
"Dramatic Chipmunk" "Chipmunk with a very dramatic look" 2007
```

9. Компиляция и запуск программы с базой данных, содержащей примеры мемов, дадут следующий результат:

```
$ cat memes.txt | ./filling_containers
Doge              : Very Shiba Inu. so dog. much funny. wow., 2013
Dramatic Chipmunk : Chipmunk with a very dramatic look, 2007
Gabe              : Musical dog on maximum borkdrive, 2016
Honey Badger      : Crazy nastyass honey badger, 2011
Pepe              : Anthropomorphic frog, 2016
```

Как это работает

В данном примере можно отметить три интересные детали. Первая заключается в том, что на основе данных последовательного символьного потока ввода мы заполняли не вектор или список, а более сложный контейнер `std::map`. Еще одна — мы использовали эти манипуляторы потока. Последняя — вызов `accumulate`, который определяет размер самой длинной строки.

Начнем с работы с ассоциативным массивом. Наша структура `meme` содержит только поле с описанием и год. Название интернет-мема не является частью структуры, поскольку используется в качестве ключа. При вставке чего-нибудь в ассоциативный массив можно предоставить объект типа `std::pair`, имеющий разные типы для ключа и значения. Именно это мы и сделали. Сначала реализовали оператор потока `>>` для структуры `meme`, а затем сделали то же самое для типа `pair<string, meme>`. Затем мы использовали конструкцию `istream_iterator<pair<string, meme>>{cin}`, чтобы получить такие элементы из стандартного потока ввода и передать их в ассоциативный массив с помощью `inserter(m, end(m))`.

При десериализации элементов типа `meme` из потока мы разрешили использовать пробелы в названиях и описаниях. Это легко реализовать, однако они не длиннее одной строки, поскольку мы поместили данные поля в кавычки. Взгляните на пример формата строк: `"Name with spaces" "Description with spaces" 123`.

При работе со строками, заключенными в кавычки как на входе, так и на выходе, поможет `std::quoted`. Если у нас есть строка `s`, то при выводе ее с помощью `cout << quoted(s)` она будет заключена в кавычки. В случае десериализации строки из потока, например используя `cin >> quoted(s)`, мы считаем следующий символ кавычек, заполним строку символами, стоящими после него, и будем делать это до тех пор, пока не увидим следующий символ кавычек, независимо от количества встреченных пробелов.

Последний необычный момент заключается в том, как мы передали функцию `max_func` в наш вызов алгоритма `accumulate`:

```
auto max_func ([](size_t old_max, const auto &b) {
    return max(old_max, b.first.length());
});
size_t width {accumulate(begin(m), end(m), 0u, max_func)};
```

Похоже, что функция `max_func` принимает аргумент типа `size_t` и еще один аргумент с автоматическим типом, который оказывается элементом типа `pair`, взятым из ассоциативного массива. На первый взгляд все выглядит очень странно, поскольку большинство бинарных функций сжатия принимают аргументы идентичных типов, а затем объединяют их с помощью некой операции, как, например, это делает `std::plus`. В нашем случае все выглядит по-другому, поскольку мы не объединяем сами пары. Мы только получаем длину каждой строки, представляющей собой ключ, для каждой пары, *отбрасываем* остальное, а затем сжимаем полученные значения типа `size_t` с помощью функции `max`.

В вызове `accumulate` первый вызов функции `max_func` получает значение `0u`, изначально предоставленное нами в качестве левого аргумента, и ссылку на первый элемент типа `pair` с правой стороны. Это дает возвращаемое значение `max(0u, string_length)`, которое станет левым аргументом для *следующего* вызова, где очередная пара будет представлять собой правый параметр, и т. д.

Выводим любые данные на экран с помощью итераторов std::ostream

Очень легко вывести что-то на экран с помощью потоков вывода, поскольку в STL есть много полезных перегруженных версий оператора `<<` для большинства простых типов. Таким образом, структуры данных, содержащие элементы подобных типов, можно легко вывести на экран, задействовав класс `std::ostream_iterator`, что мы довольно часто делали.

В данном примере мы сконцентрируемся на том, как это сделать для пользовательского типа и что еще можно сделать для управления выводом на экран с помощью выбора типа шаблона без необходимости писать при этом много кода на вызывающей стороне.

Как это делается

В этом примере мы поработаем с итератором `std::ostream_iterator`, объединив его с новым пользовательским классом, и взглянем на его возможности неявного преобразования, что может помочь при выводе данных на экран.

1. Сначала указываем, какие заголовочные файлы включить, а затем объявляем об использовании пространства имен `std`:

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

using namespace std;
```

2. Реализуем функцию преобразования, которая соотносит числа и строки. Она будет возвращать строку `"one"` для значения `1`, `"two"` для значения `2` и т. д.:

```
string word_num(int i) {
```

3. Мы заполним ассоциативный массив, основанный на хешах, этими парами, чтобы получить к ним доступ позже:

```
    unordered_map<int, string> m {
        {1, "one"}, {2, "two"}, {3, "three"},
        {4, "four"}, {5, "five"}, //...
    };
```

4. Теперь можно передать в функцию `find` ассоциативного массива, основанного на хеше, аргумент `i` и вернуть то значение, которое она найдет. Если функция ничего не найдет — например, для заданного числа нет перевода, — то вернем строку `"unknown"`:

```
    const auto match (m.find(i));
    if (match == end(m)) { return "unknown"; }
```

```
    return match->second;
};
```

5. Мы будем работать также со структурой `bork`. Она содержит только одно целое число и неявно создается на основе целого числа. Кроме того, она имеет функцию `print`, которая принимает ссылку на поток вывода и выводит строку `"bork"` столько раз, сколько указано в целочисленной переменной `borks`:

```
struct bork {
    int borks;
    bork(int i) : borks{i} {}

    void print(ostream& os) const {
        fill_n(ostream_iterator<string>{os, " "},
               borks, "bork!"s);
    }
};
```

6. Для использования функции `bork::print` перегрузим оператор `<<` для объектов потока, чтобы они автоматически вызывали функцию `bork::print`, когда объекты типа `bork` попадают в поток вывода:

```
ostream& operator<<(ostream &os, const bork &b) {
    b.print(os);
    return os;
}
```

7. Теперь наконец можем начать реализовывать саму функцию `main`. Изначально просто создаем вектор, содержащий некоторые значения:

```
int main()
{
    const vector<int> v {1, 2, 3, 4, 5};
```

8. Для объектов типа `ostream_iterator` нужен параметр шаблона, который указывает, переменные какого типа они могут выводить. Если мы напишем `ostream_iterator<T>`, то в дальнейшем для вывода данных на экран будет применяться конструкция `ostream& operator(ostream&, const T&)`. Именно это свойство мы и реализовали до типа `bork`. На сей раз просто выводим целые числа, поэтому специализация выглядит как `ostream_iterator<int>`. Для вывода информации на экран мы будем использовать поток `cout`, так что предоставим его в качестве параметра конструктора. Пройдем по вектору в цикле и присвоим каждый элемент `i` разыменованному итератору вывода. Именно так потоковые итераторы используются в алгоритмах STL.

```
    ostream_iterator<int> oit {cout};
    for (int i : v) { *oit = i; }
    cout << '\n';
```

9. Полученный от итератора результат нам подходит, но он выводит числа без каких-либо разделителей. Если мы хотим добавить пробелы-разделители между всеми выведенными элементами, то можем предоставить собственную строку

с пробелами в качестве второго параметра конструктора итератора выводного потока. Данное действие позволит вывести строку `"1, 2, 3, 4, 5, "` вместо строки `"12345"`. К сожалению, мы не можем указать отбросить пробел с запятой после последнего числа, поскольку итератор не знает, что достиг конца строки, до тех пор, пока это не случится.

```
ostream_iterator<int> oit_comma {cout, ", "};
for (int i : v) { *oit_comma = i; }
cout << '\n';
```

10. Присвоение элементов итератору потока вывода для вывода их на экран — один из корректных способов использования итератора, но его создавали не для этого. Идея заключается в том, чтобы применить их вместе с алгоритмами. Самым простым алгоритмом является `std::copy`. Можно предоставить начальный и конечный итераторы вектора в качестве входного диапазона данных, а также итератор вывода потока в качестве итератора вывода. Алгоритм выведет все числа, содержащиеся в векторе. Сделаем это для обоих итераторов вывода, а затем сравним полученный результат с результатом работы циклов, написанных нами ранее:

```
copy(begin(v), end(v), oit);
cout << '\n';
copy(begin(v), end(v), oit_comma);
cout << '\n';
```

11. Помните функцию `word_num`, которая соотносит числа и строки — `1` с `"one "`, `2` с `"two"` и т. д.? Можно использовать для вывода данных и ее. Нужен только оператор вывода потока, имеющий шаблон, специализированный для типа `string`, поскольку мы больше не выводим целые числа. Вместо алгоритма `std::copy` станем использовать алгоритм `std::transform`, поскольку он позволяет применять функцию преобразования для каждого элемента входного диапазона до того, как эти элементы будут скопированы в выходной диапазон.

```
transform(begin(v), end(v),
          ostream_iterator<string>{cout, " "},
          word_num);
cout << '\n';
```

12. В последней строке выходных данных наконец используем структуру `bork`. Мы могли бы передать алгоритму `std::transform` функцию преобразования, но не стали. Вместо этого можно просто создать итератор потока вывода, который специализирован для типа `bork` в вызове `std::copy`. В результате экземпляры типа `bork` будут *неявно* создаваться на основе целых чисел, находящихся в диапазоне данных. Это даст интересные выходные данные:

```
copy(begin(v), end(v),
     ostream_iterator<bork>{cout, "\n"});
}
```

13. Компиляция и запуск программы дадут следующий результат. Первые две строки полностью идентичны следующим двум строкам, как мы и ожидали.

Далее мы получили «аккуратные» строки, содержащие текстовое написание чисел, разделенное пробелами, а после этого — множество строк bork!, для них был использован разделитель "\n" вместо пробелов.

```
$ ./ostream_printing
12345
1, 2, 3, 4, 5,
12345
1, 2, 3, 4, 5,
one two three four five
bork!
bork! bork!
bork! bork! bork!
bork! bork! bork! bork!
bork! bork! bork! bork! bork!
```

Как это работает

Мы увидели, что итератор std::ostream_iterator — всего лишь оболочка для функции вывода данных на экран, имеющая форму и синтаксис итератора. Инкрементирование этого итератора ни к чему не приводит. Его разыменование возвращает лишь прокси-объект, чей оператор присваивания перенаправляет аргумент в поток вывода.

Итераторы потока вывода, специализированные для типа T (например, ostream_iterator<T>), работают для всех типов, для которых предоставлена реализация ostream& operator<<(ostream&, const T&).

Итератор ostream_iterator всегда пытается вызвать оператор << того типа, для которого он был специализирован, с помощью своего параметра шаблона. Он попробует неявно преобразовать типы, если это разрешено. Когда мы итерируем по диапазону элементов типа A, но копируем данные элементы в экземпляры типа output_iterator<B>, то код будет работать при условии, что тип A можно неявно преобразовать к типу B. Мы сделали именно это для структуры bork: экземпляр типа bork можно неявно получить из целочисленного значения. Как следствие, можно легко сгенерировать множество строк "bork! " на консоли пользователя.

Если неявное преобразование выполнить нельзя, можно провести его самостоятельно с помощью алгоритма std::transform, что мы и сделали в комбинации с функцией word_num.

Обратите внимание: как правило, разрешение неявного преобразования для пользовательских типов — плохой стиль программирования, поскольку такие преобразования являются распространенным источником ошибок, которые очень трудно найти. В нашем примере неявный конструктор скорее полезен, чем опасен, поскольку класс применяется только для вывода данных на экран.

Перенаправляем выходные данные в файл для конкретных разделов кода

Поток `std::cout` предоставляет удобный способ вывести на экран все, что мы хотим и когда хотим, поскольку его действительно просто использовать, легко расширять и получать к нему доступ глобально. Даже если мы желаем выводить особые сообщения, например сообщения об ошибках, которые нужно изолировать от обычных сообщений, то можем просто задействовать поток `std::cerr`. Он похож на `cout`, но выводит данные в стандартный канал ошибок, а не в стандартный канал для выходных данных.

При журналировании некоторой информации могут возникать и более сложные требования. Предположим, нужно *перенаправить* выходные данные, получаемые от некой функции, в файл или *заглушить* выходные данные, получаемые от функции, не изменяя ее саму. Возможно, это библиотечная функция и у нас нет доступа к ее исходному коду. Может быть, эта функция никогда не имела возможности записывать данные в файл, но мы хотим, чтобы она это делала.

Перенаправить выходные данные объектов потока можно. Далее мы рассмотрим, как это сделать очень простым и элегантным способом.

Как это делается

В этом примере мы реализуем вспомогательный класс, который решает задачу перенаправления потока и отмены такого перенаправления средствами конструкторов/деструкторов. А затем увидим, как это можно использовать.

1. В этот раз нужны только заголовочные файлы для потоков ввода/вывода и файлового потока. Кроме того, мы объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <fstream>

using namespace std;
```

2. Реализуем класс, содержащий объект файлового потока и указатель на буфер потока. `cout`, представленный как объект потока, имеет внутренний буфер, который можно подменить. В процессе выполнения подобной подмены можно сохранить его исходное состояние, чтобы в будущем иметь возможность *отменить* это изменение. Мы могли бы найти его тип в справочном материале по C++, но также можем использовать `decltype`, чтобы узнать, какой тип возвращает конструкция `cout.rdbuf()`. Данный прием подходит не для всех ситуаций, но в нашем случае мы получим тип указателя:

```
class redirect_cout_region
{
    using buftype = decltype(cout.rdbuf());
    ofstream ofs;
    buftype  buf_backup;
```

3. Конструктор нашего класса принимает в качестве единственного параметра строку `filename`. Данная строка используется для инициализации члена файлового потока `ofs`. После этого можно передать его в `cout` в качестве нового буфера потока. Та же функция, что принимает новый буфер, также возвращает указатель на старый, поэтому можно сохранить его, чтобы в будущем восстановить.

```
public:
    explicit
    redirect_cout_region (const string &filename)
        : ofs{filename},
          buf_backup{cout.rdbuf(ofs.rdbuf())}
    {}
```

4. Конструктор по умолчанию делает то же, что и предыдущий конструктор. Различие заключается вот в чем: он не открывает никаких файлов. Передача созданного по умолчанию буфера файлового потока в поток `cout` приводит к тому, что `cout` в некотором роде *деактивизируется*. Он просто будет *отбрасывать* входные данные, что мы ему передаем. Это также может быть полезно в отдельных ситуациях.

```
    redirect_cout_region()
        : ofs{},
          buf_backup{cout.rdbuf(ofs.rdbuf())}
    {}
```

5. Деструктор просто отменяет наше изменение. Когда объект этого класса выходит из области видимости, буфер потока `cout` возвращается в исходное состояние:

```
    ~redirect_cout_region() {
        cout.rdbuf(buf_backup);
    }
};
```

6. Создадим функцию, *генерирующую множество выходных данных,* чтобы с ней можно было работать в дальнейшем:

```
void my_output_heavy_function()
{
    cout << "some output\n";
    cout << "this function does really heavy work\n";
    cout << "... and lots of it...\n";
    // ...
}
```

7. В функции `main` сначала создадим совершенно обычные выходные данные:

```
int main()
{
    cout << "Readable from normal stdout\n";
```

8. Теперь откроем еще одну область видимости, и первое, что мы в ней сделаем, — создадим экземпляр нового класса с параметром в виде текстового файла.

Файловые потоки открывают файлы в режиме чтения и записи по умолчанию, поэтому создадут для нас данный файл. Любые выходные данные будут перенаправлены в него, однако для вывода данных мы используем `cout`:

```
{
    redirect_cout_region _ {"output.txt"};
    cout << "Only visible in output.txt\n";
    my_output_heavy_function();
}
```

9. После того как мы покинем область видимости, файл будет закрыт и выходные данные станут перенаправляться в стандартный поток вывода. Теперь откроем еще одну область видимости, в которой создадим экземпляр того же класса с помощью конструктора по умолчанию. Таким образом, следующая строка нигде не будет видна, она просто отбросится:

```
{
    redirect_cout_region _;
    cout << "This output will "
            "completely vanish\n";
}
```

10. После того как мы покинем эту область видимости, стандартный поток вывода восстановится и последнюю строку можно будет увидеть на консоли.

```
    cout << "Readable from normal stdout again\n";
}
```

11. Компиляция и запуск программы дадут следующий ожидаемый результат. На консоли будут видны только первая и последняя строки:

```
$ ./log_regions
Readable from normal stdout
Readable from normal stdout again
```

12. Можно увидеть, что был создан новый файл `output.txt`, который содержит выходные данные, полученные из первой области видимости. Выходные данные, полученные из второй области видимости, пропали без следа:

```
$ cat output.txt
Only visible in output.txt
some output
this function does really heavy work
... and lots of it...
```

Как это работает

Каждый объект потока имеет внутренний буфер, для которого он играет роль фронтенда. Такие буферы взаимозаменяемы. Если у нас есть объект потока `s`, а мы хотим сохранить его буфер в переменную `a` и установить новый буфер `b`, то данная

конструкция будет выглядеть так: `a = s.rdbuf(b)`. Восстановить буфер можно следующим образом: `s.rdbuf(a)`.

Именно это мы и сделали в данном примере. Еще один положительный момент заключается в том, что можно *объединять* эти вспомогательные функции `redirect_cout_region`:

```
{
    cout << "print to standard output\n";

    redirect_cout_region la {"a.txt"};
    cout << "print to a.txt\n";
    redirect_cout_region lb {"b.txt"};
    cout << "print to b.txt\n";
}
cout << "print to standard output again\n";
```

Этот код работает, поскольку объекты разрушаются в порядке, обратном порядку их создания. Концепция, лежащая в основе данного шаблона, который использует тесное связывание между созданием и разрушением объектов, называется *«Получение ресурса есть инициализация»* (resource acquisition is initialization, RAII).

Следует упомянуть еще один очень важный момент — обратите внимание на *порядок инициализации* переменных-членов класса `redirect_cout_region`:

```
class redirect_cout_region {
    using buftype = decltype(cout.rdbuf());

    ofstream ofs;
    buftype  buf_backup;

public:
    explicit
    redirect_cout_region(const string &filename)
        : ofs{filename},
          buf_backup{cout.rdbuf(ofs.rdbuf())}
    {}
...
```

Как видите, член `buf_backup` создается из выражения, которое зависит от `ofs`. Очевидно, это значит следующее: `ofs` нужно инициализировать до `buf_backup`. Что интересно, порядок, в котором инициализируются переменные-члены, *не зависит* от порядка элементов списка инициализаторов. Порядок инициализации зависит только от порядка *объявления членов*!

Если одна переменная-член должна быть инициализирована после другой переменной, то их нужно привести именно в этом порядке при объявлении членов класса. Порядок их появления в списке инициализаторов конструктора некритичен.

Создаем пользовательские строковые классы путем наследования std::char_traits

Класс `std::string` очень полезен. Однако, как только пользователям нужен строковый класс, семантика которого несколько отличается от обычной обработки строк, они пишут *собственный* класс `string`. Такая идея редко хороша, поскольку не так просто безопасно обработать строки. К счастью, класс `std::string` — лишь специализированное ключевое слово шаблонного класса `std::basic_string`. Данный класс содержит все сложные средства обработки памяти, но не навязывает никаких правил, как копировать строки, сравнивать их и т. д. Эти правила импортируются в `basic_string`, для чего принимается шаблонный параметр, который содержит класс `traits`.

В этом примере мы увидим, как создавать собственные классы типажей и, соответственно, как создавать пользовательские строки, не реализуя повторно все их возможности.

Как это делается

В этом примере мы реализуем два разных пользовательских строковых класса: `lc_string` и `ci_string`. Первый создает на основе любых входных данных строки в нижнем регистре. Второй строки не преобразует, но может выполнить сравнение строк независимо от регистра.

1. Включим несколько необходимых заголовочных файлов, а затем объявим об использовании пространства имен `std` по умолчанию:

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

using namespace std;
```

2. Затем снова реализуем функцию `std::tolower`, которая уже определена в `<cctype>`. Уже существующая функция работает хорошо, но она не имеет модификатора `constexpr`. Однако отдельные строковые функции имеют этот модификатор, начиная с C++17, и мы хотим иметь возможность использовать их для нашего собственного строкового класса-типажа. Функция соотносит символы в верхнем регистре с символами в нижнем регистре и оставляет другие символы неизменными:

```
static constexpr char tolow(char c) {
    switch (c) {
    case 'A'...'Z': return c - 'A' + 'a';
    default:        return c;
    }
}
```

3. Класс `std::basic_string` принимает три шаблонных параметра: тип основного символа, класс-типаж символа и тип `allocator`. В этом разделе мы изменяем только класс-типаж символа, поскольку он определяет поведение строк. Для повторной реализации только того, что должно отличаться от типичного поведения строк, мы явно наследуем от стандартного класса-типажа:

```
class lc_traits : public char_traits<char> {
public:
```

4. Наш класс принимает входные строки, но преобразует их в строки в нижнем регистре. Существует функция, которая делает это посимвольно, так что можно поместить здесь нашу функцию `tolow`. Она имеет модификатор `constexpr`, именно поэтому мы и реализовали самостоятельно функцию `tolow` с таким же модификатором `constexpr`:

```
    static constexpr
    void assign(char_type& r, const char_type& a ) {
        r = tolow(a);
    }
```

5. Еще одна функция обрабатывает копирование целой строки в отдельный участок памяти. Мы используем вызов `std::transform`, чтобы скопировать все символы из исходной строки в строку — место назначения, и в то же время соотносим каждый символ с его версией в нижнем регистре:

```
    static char_type* copy(char_type* dest,
                           const char_type* src,
                           size_t count) {
        transform(src, src + count, dest, tolow);
        return dest;
    }
};
```

6. Еще один типаж помогает создать строковый класс, эффективно преобразующий строки в нижний регистр. Напишем еще один типаж, который не изменяет полученную строку, но позволяет выполнять сравнение строк независимо от регистра. Снова унаследуем от существующего стандартного класса-типажа для символов и в этот раз переопределим некоторые члены функции:

```
class ci_traits : public char_traits<char> {
public:
```

7. Функция `eq` указывает, равны ли два символа. Реализуем такую же функцию, но будем сравнивать версии символов в нижнем регистре. При использовании этой функции символ `'A'` равен символу `'a'`.

```
    static constexpr bool eq(char_type a, char_type b) {
        return tolow(a) == tolow(b);
    }
```

8. Функция `lt` указывает, меньше ли значение `a` значения `b`. Применим корректный логический оператор, но только после того, как оба символа будут преобразованы к нижнему регистру:

```
static constexpr bool lt(char_type a, char_type b) {
    return tolow(a) < tolow(b);
}
```

9. Последние две функции работали для посимвольного ввода, а следующие две будут работать для строк. Функция `compare` работает по аналогии со старой функцией `strncmp`. Она возвращает значение `0` при условии, что обе строки равны от начала до позиции, указанной в переменной `count`. Если они отличаются, то функция возвращает отрицательное или положительное число, которое указывает, какая из строк была меньше с точки зрения лексикографии. Определение разности между обоими символами в каждой позиции должно, конечно же, происходить для символов в нижнем регистре. Положительный момент заключается в том, что весь код цикла является частью функции с модификатором `constexpr`, начиная с C++14.

```
static constexpr int compare(const char_type* s1,
                             const char_type* s2,
                             size_t count) {
    for (; count; ++s1, ++s2, --count) {
        const char_type diff (tolow(*s1) - tolow(*s2));
        if      (diff < 0) { return -1; }
        else if (diff > 0) { return +1; }
    }
    return 0;
}
```

10. Последняя функция, которую нужно реализовать для нашего строкового класса, не зависящего от регистра, — это функция `find`. Для заданной входной строки `p` и ее длины `count` она определяет позицию символа `ch`. Затем возвращает указатель на первое включение данного символа или `nullptr`, если такого символа нет. Сравнение в этой функции должно выполняться с использованием функции `tolow`, чтобы поиск не зависел от регистра. К сожалению, мы не можем применить функцию `std::find_if`, поскольку она не имеет модификатора `constexpr`, нужно писать цикл самостоятельно.

```
    static constexpr
    const char_type* find(const char_type* p,
                          size_t count,
                          const char_type& ch) {
        const char_type find_c {tolow(ch)};
        for (; count != 0; --count, ++p) {
            if (find_c == tolow(*p)) { return p; }
        }
        return nullptr;
    }
};
```

11. О'кей, с типажами мы закончили. Теперь можно определить два новых строковых типа. `lc_string` означает *lower-case string* (строка в нижнем регистре). `ci_string` расшифровывается как *case-insensitive string* (строка, не зависящая от регистра). Оба класса отличаются от класса `std::string` своими классами-типажами для символов:

```
using lc_string = basic_string<char, lc_traits>;
using ci_string = basic_string<char, ci_traits>;
```

12. Чтобы позволить потокам вывода принимать эти новые классы для вывода на экран, нужно перегрузить потоковый оператор `<<`:

```
ostream& operator<<(ostream& os, const lc_string& str) {
    return os.write(str.data(), str.size());
}
ostream& operator<<(ostream& os, const ci_string& str) {
    return os.write(str.data(), str.size());
}
```

13. Теперь наконец можно начать реализовывать саму программу. Создадим экземпляр обычной строки, строки в нижнем регистре и строки, не зависящей от регистра, и сразу же выведем их на экран. Строки в нижнем регистре будут соответствовать своим названиям — их символы будут иметь нижний регистр:

```
int main()
{
    cout << "   string: "
         << string{"Foo Bar Baz"} << '\n'
         << "lc_string: "
         << lc_string{"Foo Bar Baz"} << '\n'
         << "ci_string: "
         << ci_string{"Foo Bar Baz"} << '\n';
```

14. Чтобы протестировать строку, не зависящую от регистра, можно создать две строки, которые, по сути, равны, но регистры отдельных символов различаются. При выполнении сравнения, не зависящего от регистра, они должны показаться равными:

```
    ci_string user_input {"MaGiC PaSsWoRd!"};
    ci_string password   {"magic password!"};
```

15. Сравним их и выведем на экран сообщение об их совпадении, если это так:

```
    if (user_input == password) {
        cout << "Passwords match: \"" << user_input
             << "\" == \"" << password << "\"\n";
    }
}
```

16. Компиляция и запуск программы дадут ожидаемый результат. При выводе на экран одинаковых строк с разными типами мы получили одинаковые результаты,

но в строке типа `lc_string` все символы указаны в нижнем регистре. Сравнение двух строк, различающихся лишь регистром некоторых символов, прошло успешно и дало правильный результат:

```
$ ./custom_string
   string: Foo Bar Baz
lc_string: foo bar baz
ci_string: Foo Bar Baz
Passwords match: "MaGiC PaSsWoRd!" == "magic password!"
```

Как это работает

Вся работа по созданию подклассов и повторной реализации функций, конечно же, для новичков выглядит несколько странно. Откуда появились все сигнатуры функций, из которых мы *волшебным образом* узнали о том, что именно нужно реализовать повторно?

Сначала взглянем, откуда появился класс `std::string`:

```
template <
    class CharT,
    class Traits    = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
    >
class basic_string;
```

Класс `std::string`, по сути, является `std::basic_string<char>`, и данная конструкция разворачивается к виду `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`. О'кей, что означает это длинное описание типа? Идея заключается вот в чем: можно создать строку, которая работает не только с однобайтовыми элементами типа `char`, но и с другими, более крупными типами. Это позволяет создавать строковые типы, способные работать с более широкими диапазонами символов, нежели типичный диапазон символов американской таблицы ASCII. Нам сейчас не нужно обращать на это внимание.

Класс `char_traits<char>`, однако, содержит алгоритмы, которые необходимы классу `basic_string` для корректной работы. Класс `char_traits<char>` умеет сравнивать, искать и копировать символы и строки.

Класс `allocator<char>` также является классом трактовок, но его особая задача заключается в обработке процессов выделения и освобождения памяти. Сейчас для нас это неважно, поскольку поведение по умолчанию удовлетворяет наши потребности.

Если мы хотим, чтобы строковый класс вел себя иначе, то можем попробовать повторно использовать по максимуму все возможности, предоставляемые классами `basic_string` и `char_traits`. Именно это и произошло. Мы реализовали два подкласса `char_traits`, которые называются `case_insentitive` и `lower_caser`, и сконфигурировали с их помощью два совершенно новых строковых типа, применяя их в качестве замены стандартному типу `char_traits`.

Чтобы исследовать другие возможности по подстройке под себя класса basic_string, обратитесь к документации C++ STL для std::char_traits и взгляните на другие его функции, которые можно переопределить.

Токенизация входных данных с помощью библиотеки для работы с регулярными выражениями

При преобразовании строк сложными способами или разбиении их на фрагменты могут помочь *регулярные выражения*. Они встроены во многие языки программирования, поскольку очень полезны.

Если вы еще не знакомы с регулярными выражениями, то можете, например, прочесть о них в «Википедии». Это определенно расширит ваш кругозор, так как нетрудно заметить, насколько эти выражения полезны при анализе любых текстов. С их помощью можно, например, проверить корректность адреса электронной почты или IP-адреса, найти и извлечь подстроки из больших строк согласно сложному шаблону и т. д.

В этом примере мы извлечем все ссылки из HTML-документа и выведем их на экран пользователя. Код решения данной задачи будет очень коротким, поскольку язык C++ STL поддерживает регулярные выражения, начиная с версии C++11.

Как это делается

В примере мы определим регулярное выражение, которое обнаруживает ссылки, и применим его к файлу HTML, чтобы аккуратно вывести все ссылки, найденные в этом файле.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iterator>
#include <regex>
#include <algorithm>
#include <iomanip>

using namespace std;
```

2. Позднее сгенерируем итерабельный диапазон данных, который состоит из строк. Эти строки всегда будут составлять пары из ссылки и ее описания. Так что напишем небольшую вспомогательную функцию, аккуратно выводящую их на экран:

```
template <typename InputIt>
void print(InputIt it, InputIt end_it)
{
    while (it != end_it) {
```

3. На каждом шаге цикла дважды инкрементируем итератор и берем копии ссылки и ее описания, которые он содержит. Между двумя операциями разыменования итератора добавляем еще один охранный блок `if`, из соображений безопасности проверяющий, достигли ли мы конца итерабельного диапазона данных:

```
        const string link {*it++};
        if (it == end_it) { break; }
        const string desc {*it++};
```

4. Теперь выведем на экран ссылки и их описания в аккуратном виде:

```
        cout << left << setw(28) << desc
             << " : " << link << '\n';
    }
}
```

5. В функции `main` считываем все, что поступает в стандартный поток ввода. Для этого создаем строку из всех данных, поступивших в стандартный поток, с помощью входного потокового итератора. Для предотвращения токенизации, поскольку нам нужны все входные данные, используем модификатор `noskipws`. Этот модификатор деактивизирует пропуск пробелов и токенизацию:

```
int main()
{
    cin >> noskipws;
    const std::string in {istream_iterator<char>{cin}, {}};
```

6. Теперь следует определить регулярное выражение, которое описывает, как с нашей точки зрения должна выглядеть ссылка HTML. Скобки `()` внутри регулярного выражения определяют группы. Они являются частями ссылки, к которым нужно получить доступ — URL и его описание:

```
    const regex link_re {
        "<a href=\"([^\"]*)\"[^<]*>([^<]*)</a>"};
```

7. Класс `sregex_token_iterator` выглядит точно так же, как и класс `istream_iterator`. Мы передадим ему целую строку, представляющую собой итерабельный диапазон данных, и определенное нами регулярное выражение. Третий параметр `{1, 2}` — это список инициализаторов целочисленных значений. Он определяет, что мы хотим итерировать по группам 1 и 2 из полученных им выражений:

```
    sregex_token_iterator it {
        begin(in), end(in), link_re, {1, 2}};
```

8. Теперь у нас есть итератор, который будет возвращать все найденные ссылки и их описания. Мы предоставим его и итератор того же типа, созданный по умолчанию, функции `print`, реализованной нами ранее:

```
    print(it, {});
}
```

9. Компиляция и запуск программы дадут следующий результат. Я запустил программу `curl` для домашней страницы ISO C++, которая просто загружает HTML-страницу из Интернета. Конечно, я мог и написать `cat some_html_file.html | ./link_extraction`. Использованное нами регулярное выражение довольно жестко определяет представление о том, как должны выглядеть ссылки в документе HTML. В качестве самостоятельной работы можете сделать его более обобщенным.

```
$ curl -s "https://isocpp.org/blog" | ./link_extraction
Sign In / Suggest an Article : https://isocpp.org/member/login
Register                     : https://isocpp.org/member/register
Get Started!                 : https://isocpp.org/get-started
Tour                         : https://isocpp.org/tour
C++ Super-FAQ                : https://isocpp.org/faq
Blog                         : https://isocpp.org/blog
Forums                       : https://isocpp.org/forums
Standardization              : https://isocpp.org/std
About                        : https://isocpp.org/about
Current ISO C++ status       : https://isocpp.org/std/status
 (...и многие другие...)
```

Как это работает

Регулярные выражения (или коротко *regex*) очень полезны. Они могут казаться очень сложными, но вам стоит изучить принципы их работы. Короткое регулярное выражение может избавить от необходимости писать множество строк кода, что пришлось бы сделать при выполнении проверки на соответствие вручную.

В данном примере мы сначала создали объект типа регулярных выражений. Мы передали его конструктору строку, которая описывает регулярное выражение. Самое простое регулярное выражение выглядит как `"."`, оно соответствует *любому* символу, поскольку точка — это специальный символ для регулярного выражения. Выражение `"a"` соответствует только символам `'a'`. Выражение `"ab*"` означает «один символ `a`, а затем ноль или больше символов `b`» и т. д. Регулярные выражения — довольно обширная тема, более подробную информацию можно найти в «Википедии» и на других сайтах и в литературе.

Еще раз взглянем на регулярное выражение, соответствующее нашему представлению о ссылках HTML. Простая ссылка HTML может выглядеть как `<a href="some_url.com/foo">A great link</a>`. Нужно получить часть `some_url.com/foo`, а также `A great link`. Мы создали следующее регулярное выражение, которое содержит *группы* для соответствия подстрокам (рис. 7.2).

Полное совпадение всегда является *группой 0*. В данном случае это будет вся строка `<a href ..... </a>`. Заключенная в кавычки часть `href`, которая содержит URL, — это *группа 1*. Скобки в регулярном выражении определяют такие группы. Их у нас две. Еще одна группа — фрагмент текста между тегами `<a ...>` и `</a>`, содержащий описание ссылки.

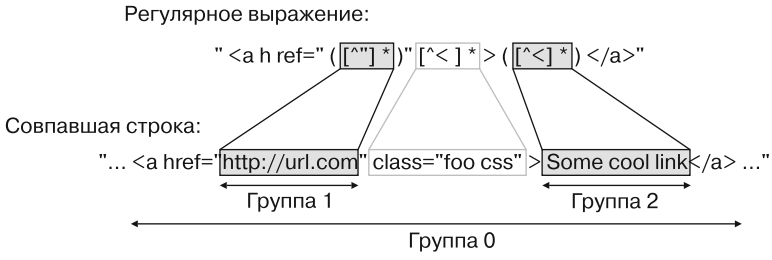

Рис. 7.2

Существует множество функций STL, которые принимают объекты регулярных выражений. Однако мы непосредственно использовали адаптер для итератора, работающего с токенами регулярного выражения. Он представляет собой высокоуровневую абстракцию, применяющую `std::regex_search`, чтобы автоматизировать работу по поиску совпадений. Мы создали его экземпляр следующим образом:

```
sregex_token_iterator it {begin(in), end(in), link_re, {1, 2}};
```

Части `begin` и `end` обозначают нашу входную строку, по которой итератор, работающий с токенами регулярного выражения, будет итерировать и искать все ссылки. `link_re` — это, конечно, сложное регулярное выражение, созданное нами для поиска ссылок. Часть `{1, 2}` — еще один непонятный фрагмент. Он дает итератору команду опускать полное совпадение и возвращать группу 1, затем инкрементировать итератор и возвращать группу 2, а позже, после очередной операции инкремента, находить следующее совпадение в строке. Это разумное поведение освобождает нас от необходимости писать ненужные строки кода.

Давайте рассмотрим еще один пример. Представим регулярное выражение `"a(b*)(c*)"`. Оно будет соответствовать строкам, содержащим символ `a`, после которого идет ноль или больше символов `b`, а затем — ноль или больше символов `c`:

```
const string s {" abc abbccc "};
const regex re {"a(b*)(c*)"};

sregex_token_iterator it {begin(s), end(s), re, {1, 2}};

print( *it ); // выводит b
++it;
print( *it ); // выводит c
++it;
print( *it ); // выводит bb
++it;
print( *it ); // выводит ccc
```

Существует также класс `std::regex_iterator`, который генерирует подстроки, находящиеся *между* совпадениями с регулярными выражениями.

Удобный и красивый динамический вывод чисел на экран в зависимости от контекста

В предыдущем примере было показано, как отформатировать выходные данные с помощью потоков вывода. Во время решения данной задачи мы узнали следующее:

- большая часть манипуляторов являются стойкими, поэтому нужно отменять их действие, чтобы не пересечься с другим несвязанным кодом, который также выводит данные на экран;
- создавать длинные цепочки манипуляторов ввода/вывода, чтобы вывести несколько переменных с конкретным форматированием, может быть утомительно, а код получится не очень читабельным.

По этим причинам многие пользователи не любят потоки ввода/вывода и даже в C++ все еще применяют функцию `print` для форматирования строк.

Ниже мы увидим, как форматировать типы динамически, не прибегая к чрезмерно большому количеству манипуляторов ввода/вывода.

Как это делается

В этом примере мы реализуем класс `format_guard`, способный автоматически отменить любые настройки форматирования. Кроме того, напишем тип-оболочку, который может содержать любые значения, но при выводе будет применять особое форматирование, не загружая код лишними манипуляторами ввода/вывода.

1. Сначала включим некоторые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>

using namespace std;
```

2. Вспомогательный класс, очищающий состояние форматирования для потоков, называется `format_guard`. В его конструкторе сохраняются флаги форматирования, которые были заданы для `std::cout` в момент создания объекта. Его деструктор восстанавливает их состояние на момент вызова конструктора. Это, по сути, отменяет все настройки форматирования, примененные между вызовами данных методов.

```
class format_guard {
    decltype(cout.flags()) f {cout.flags()};
public:
    ~format_guard() { cout.flags(f); }
};
```

3. Еще один небольшой вспомогательный класс называется `scientific_type`. Поскольку это шаблон класса, он может оборачивать любой тип, который будет представлять собой переменную-член. По сути, он ничего не делает.

```
template <typename T>
struct scientific_type {
    T value;
    explicit scientific_type(T val) : value{val} {}
};
```

4. Можно определить собственные настройки форматирования для любого типа, который ранее был обернут в `scientific_type`, поскольку в случае перегрузки оператора потока `>>` потоковая библиотека будет выполнять совершенно другой код при выводе подобных типов. Таким образом, можно выводить научные значения в научном представлении с плавающей точкой, в верхнем регистре и явным префиксом `+`, если они имеют положительные значения. Кроме того, мы используем наш класс `format_guard`, чтобы очистить все настройки при выходе из функции:

```
template <typename T>
ostream& operator<<(ostream &os, const scientific_type<T> &w) {
    format_guard _;
    os << scientific << uppercase << showpos;
    return os << w.value;
}
```

5. В функции `main` сначала поработаем с классом `format_guard`. Откроем новую область видимости, получим экземпляр класса, а затем применим некоторые флаги форматирования к потоку вывода `std::cout`:

```
int main()
{
    {
        format_guard _;
        cout << hex << scientific << showbase << uppercase;
        cout << "Numbers with special formatting:\n";
        cout << 0x123abc << '\n';
        cout << 0.123456789 << '\n';
    }
```

6. После того как выведем некоторые числа с помощью флагов форматирования, покинем область видимости. В результате деструктор класса `format_guard` сбросит настройки форматирования. Чтобы это протестировать, выведем точно такие же числа *снова*. Они должны выглядеть по-другому:

```
    cout << "Same numbers, but normal formatting again:\n";
    cout << 0x123abc << '\n';
    cout << 0.123456789 << '\n';
```

7. Теперь воспользуемся классом `scientific_type`. Выведем на экран три числа с плавающей точкой в ряд. Второе число обернем в класс `scientific_type`.

Соответственно, оно будет выведено с применением нашего особенного стиля форматирования, а числа, стоящие перед ним и после него, будут иметь форматирование по умолчанию. В то же время мы избежим появления *ненужных* символов.

```
    cout << "Mixed formatting: "
         << 123.0 << " "
         << scientific_type{123.0} << " "
         << 123.456 << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат. Первые два числа будут выведены с конкретным форматированием. Следующие два будут иметь форматирование по умолчанию — это показывает, что наш класс `format_guard` работает хорошо. Три числа в последних строках также выглядят соответственно нашим ожиданиям. Число, которое стоит посередине, имеет форматирование класса `scientific_type`, остальные же имеют форматирование по умолчанию:

```
$ ./pretty_print_on_the_fly
Numbers with special formatting:
0X123ABC
1.234568E-01
Same numbers, but normal formatting again:
1194684
0.123457
Mixed formatting: 123 +1.230000E+02 123.456
```

Перехватываем читабельные исключения для ошибок потока std::iostream

Ни в одном из примеров, показанных в данной главе, мы не использовали для обнаружения ошибок *исключения*. Несмотря на такую возможность, можно работать с объектами потоков без исключений — это очень удобно. Если мы попробуем преобразовать десять значений, но где-то в середине будет сгенерирована ошибка, то весь объект потока войдет в ошибочное состояние и перестанет работать. Таким образом, мы не столкнемся с ситуацией, когда преобразуем переменные с ошибочным смещением в потоке. Мы можем выполнять условные преобразования, например, так: `if (cin >> foo >> bar >> ...)`. В случае сбоя этой конструкции мы его обработаем. Использование конструкции `try { ... } catch ...` при преобразовании объектов не кажется очень полезным.

Фактически библиотека для работы с потоками ввода/вывода в C++ существовала уже в те времена, когда язык C++ еще не работал с исключениями. Поддержка исключений была добавлена позже, это объясняет отсутствие их первоклассной поддержки в потоковой библиотеке.

Применение исключений для потоковой библиотеки возможно при конфигурации каждого отдельного объекта потока таким образом, чтобы он генерировал исключение в тот момент, когда входит в ошибочное состояние. К сожалению,

пояснения к ошибкам, лежащие в объектах исключений, которые мы будем отлавливать, не стандартизированы. Это приводит к появлению не очень информативных сообщений об ошибках, их мы рассмотрим в данном разделе. Если вы действительно хотите использовать исключения для объектов потока, то можете *дополнительно* опросить библиотеку языка C для состояния ошибок файловой системы, чтобы получить добавочную информацию.

В данном разделе мы напишем программу, которая может генерировать множество разных сбоев, обработаем их с помощью исключений и увидим, как получить более подробное описание этих ошибок.

Как это делается

В этом примере мы реализуем программу, которая открывает файл (тут может быть сгенерирован сбой), а затем считаем оттуда целое число (здесь тоже возможен сбой). Сделаем это с помощью активизированных исключений, а затем увидим, как их обработать.

1. Сначала включим некоторые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <fstream>
#include <system_error>
#include <cstring>

using namespace std;
```

2. Для использования объектов потока вместе с исключениями нужно их активизировать. Чтобы объект файлового потока генерировал исключение в том случае, если нужный файл не существует или при преобразовании возникли ошибки, следует установить значения некоторых битов, указывающих на сбой, в маске исключения. Если мы затем сделаем нечто вызывающее сбой, это сгенерирует исключение. Активизируя `failbit` и `badbit`, мы включаем генерацию исключений для ошибок файловой системы и преобразования:

```
int main()
{
    ifstream f;
    f.exceptions(f.failbit | f.badbit);
```

3. Теперь можно открыть блок `try` и обратиться к файлу. Если последний был открыт успешно, то попробуем считать из него целое число. При успешном выполнении обоих шагов выведем целое число:

```
    try {
        f.open("non_existant.txt");
        int i;
        f >> i;
        cout << "integer has value: " << i << '\n';
    }
```

4. Если хотя бы в одной из этих ситуаций возникнет ошибка, то будет сгенерирован экземпляр `std::ios_base::failure`. Данный объект имеет функцию-член `what()`, которая должна объяснить, что вызвало генерацию исключения. К сожалению, это сообщение не стандартизировано и не слишком информативно. Однако мы можем хотя бы определить, это проблема с *файловой системой* (например, файл не существует) или же ошибка *преобразования*. Глобальная переменная `errno` существовала с момента создания C++, и она получает значение ошибки, которое мы сейчас можем получить. Функция `strerror` преобразует номер ошибки в строку, понятную для человека. Если код равен `0`, то значит, у нас возникла не ошибка файловой системы.

```
    catch (ios_base::failure& e) {
        cerr << "Caught error: ";
        if (errno) {
            cerr << strerror(errno) << '\n';
        } else {
            cerr << e.what() << '\n';
        }
    }
}
```

5. Компиляция и запуск программы для двух разных сценариев дадут следующий результат. Если нужно открыть существующий файл, но получить из него целое число нельзя, то мы получим сообщение об ошибке `iostream_category`:

```
$ ./readable_error_msg
Caught error: ios_base::clear: unspecified iostream_category error
```

6. Если файл *не существует*, то мы увидим другое сообщение от `strerror(errno)`:

```
$ ./readable_error_msg
Caught error: No such file or directory
```

Как это работает

Мы увидели, как можно добавить исключения для объекта потока `s` с помощью выражения `s.exceptions(s.failbit | s.badbit)`. Таким образом, здесь мы никак не сможем применить для открытия файла, к примеру, конструктор экземпляра `std::ifstream`, если хотим получать исключение всякий раз, когда открыть этот файл окажется невозможно:

```
ifstream f {"non_existant.txt"};
f.exceptions(...); // слишком поздно для генерации исключения
```

И это плохо, поскольку именно благодаря исключениям обработка ошибок получается не столь громоздкой, по сравнению с кодом обработки ошибок для C, который обычно заполнен множеством условий `if`, обрабатывающих ошибки после каждого шага.

Если бы мы попробовали смоделировать разные причины сбоя потоков, то увидели бы, что генерируются одинаковые исключения. Таким образом, можно

только понять, *когда* выдается ошибку, но не узнать о ее конкретном типе. (Это, конечно, неверно для обработки исключений *в целом*, только для потоковой библиотеки STL.) Именно поэтому мы дополнительно проверяем значение глобальной переменной `errno`. Она представляет собой древнюю конструкцию, которая использовалась в те времена, когда в языке C++ не было такого понятия, как исключения.

Если любая функция, связанная с системой, встречает ошибочное состояние, то может установить в качестве значения переменной `errno` нечто отличное от `0` (`0` означает отсутствие ошибок), а затем вызывающая сторона сможет прочесть этот номер ошибки и определить, что он означает. Единственная проблема заключается в следующем: если наше приложение многопоточно и все потоки используют функции, которые могут устанавливать значение данной переменной, то мы не можем выяснить, *кто* именно установил ее значение. Если мы считаем значение `0`, то это не значит, что ошибок нет, — какая-то *другая* системная функция, работающая в *другом потоке*, могла столкнуться с ошибкой. К счастью, этот недостаток был устранен в версии C++11, где каждый поток процесса видит собственную переменную `errno`.

Если не оценивать преимущества и недостатки этого древнего метода обнаружения ошибок, то можно получить полезную дополнительную информацию о том, когда было сгенерировано исключение для таких системных функций, как файловые потоки. Исключения говорят нам, *когда* они были сгенерированы, а `errno` может подсказать, *что именно* случилось, раз ошибка проявилась на системном уровне.

8 Вспомогательные классы

В этой главе:

- преобразование единиц измерения времени с помощью `std::ratio`;
- преобразование между абсолютными и относительными единицами измерения времени с использованием `std::chrono`;
- безопасное извещение о сбое с помощью `std::optional`;
- применение функций для кортежей;
- быстрое создание структур данных с помощью `std::tuple`;
- замена `void*` с использованием `std::any` для повышения безопасности типов;
- хранение разных типов с применением `std::variant`;
- автоматическое управление ресурсами с помощью `std::unique_ptr`;
- автоматическое управление разделяемой памятью кучи с использованием `std::shared_ptr`;
- работа со слабыми указателями на разделяемые объекты;
- упрощение управления ресурсами устаревших API с применением умных указателей;
- открытие доступа к разным переменным — членам одного объекта;
- генерация случайных чисел и выбор правильного генератора случайных чисел;
- генерация случайных чисел и создание конкретных распределений с помощью STL.

Введение

Эта глава посвящена вспомогательным классам, которые очень удобны для решения конкретных задач. Некоторые из них хороши настолько, что мы, вероятно, либо будем их очень часто встречать в любом фрагменте кода C++, либо уже видели в других главах книги.

Первые два примера посвящены измерению *времени*. Кроме того, мы увидим, как выполнять преобразования между разными единицами измерения времени и перепрыгивать с одного момента времени на другой.

В следующих пяти примерах мы рассмотрим типы `optional`, `variant` и `any` (они появились в C++14 и C++17), а также некоторые способы использования кортежей.

Начиная с C++11, у нас появились сложные типы умных указателей, а именно `unique_ptr`, `shared_ptr` и `weak_ptr`, которые очень эффективны при *управлении памятью*, поэтому уделим им особое внимание.

Наконец, мы кратко рассмотрим те части библиотеки STL, которые связаны с генерацией *случайных чисел*. Помимо изучения самых важных характеристик генераторов случайных чисел STL мы также узнаем, как обрабатывать случайные числа, чтобы получить распределения, соответствующие нашим потребностям.

Преобразуем единицы измерения времени с помощью std::ratio

Начиная с C++11 STL включает новые типы и функции для получения, измерения и отображения времени. Эта часть библиотеки существует в пространстве имен `std::chrono`.

В этом примере мы сконцентрируемся на измерении промежутков времени и преобразовании результатов между единицами измерения времени, такими как секунды, миллисекунды и микросекунды. STL поддерживает функции, которые позволяют определять собственные единицы измерения времени и выполнять преобразования между ними.

Как это делается

В данном примере мы напишем небольшую *игру*, приглашающую пользователя ввести конкретное слово. Время, за которое он введет его с клавиатуры, будет измерено и выведено в нескольких единицах измерения.

1. Сначала включим все необходимые заголовочные файлы. Для удобства также объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <chrono>
#include <ratio>
#include <cmath>
#include <iomanip>
#include <optional>

using namespace std;
```

2. Тип `chrono::duration` используется для выражения промежутков времени, сравнимых с долями секунд. Все единицы измерения времени, представленные

в STL, ссылаются на целочисленные специализации. В этом примере мы будем конкретизировать их для типа `double`. В следующем больше сконцентрируемся на существующих определениях единиц времени, уже встроенных в STL.

```
using seconds = chrono::duration<double>;
```

3. Миллисекунда — доля секунды, поэтому определяем эту единицу измерения в виде секунд. Параметр шаблона `ratio_multiply` применяет заранее определенный в STL делитель `milli` к `seconds::period`, что дает нужную долю секунды. Шаблон `ratio_multiply`, по сути, представляет собой функцию метапрограммирования для умножения чисел на эти множители:

```
using milliseconds = chrono::duration<
    double, ratio_multiply<seconds::period, milli>>;
```

4. То же верно и для микросекунд. Миллисекунда — миллидоля секунды, а микросекунда — микродоля секунды:

```
using microseconds = chrono::duration<
    double, ratio_multiply<seconds::period, micro>>;
```

5. Теперь реализуем функцию, которая считает входную строку, отправленную пользователем, и определит, сколько времени ему потребовалось для ее ввода. Она не принимает никаких аргументов и возвращает строку, введенную пользователем, а также время, которое ему потребовалось на ввод, упакованные в одну пару:

```
static pair<string, seconds> get_input()
{
    string s;
```

6. Нужно получить время, в которое пользователь начал вводить строку, а также время, в которое он это делать закончил. Данная операция выглядит следующим образом:

```
    const auto tic (chrono::steady_clock::now());
```

7. Сейчас мы получим данные от пользователя. Если эта операция пройдет безуспешно, то просто вернем кортеж, инициализированный значениями по умолчанию. Вызывающая сторона получит пустую строку:

```
    if (!(cin >> s)) {
        return {{}, {}};
    }
```

8. В случае успеха продолжим работу, сделав еще один снимок текущего времени. Далее вернем входную строку и разницу между временными точками. Обратите внимание: обе временные точки выражены в абсолютном виде, но, вычислив их разность, мы получаем длительность:

```
    const auto toc (chrono::steady_clock::now());
    return {s, toc - tic};
}
```

9. Теперь реализуем саму программу. Запустим цикл, который будет работать до тех пор, пока пользователь не введет корректную строку. На каждом шаге цикла мы просим пользователя ввести строку `"C++17"`, а затем вызываем функцию `get_input`:

```
int main()
{
    while (true) {
        cout << "Please type the word \"C++17\" as"
                " fast as you can.\n> ";
        const auto [user_input, diff] = get_input();
```

10. Далее проверим входные данные. Если они пусты, то интерпретируем это как запрос на завершение программы:

```
        if (user_input == "") { break; }
```

11. Если пользователь корректно введет строку `"C++17"`, то поздравим его, а затем выведем время, которое ему потребовалось на данное действие. Метод `diff.count()` возвращает количество секунд в качестве числа с плавающей точкой. Используй мы оригинальный тип `duration` STL `seconds`, получили бы *округленное* целочисленное значение, а не дробное. Передавая конструктору `milliseconds` или `microseconds` нашу переменную `diff` перед вызовом `count()`, получаем то же значение, преобразованное в другую единицу измерения:

```
        if (user_input == "C++17") {
            cout << "Bravo. You did it in:\n"
                 << fixed << setprecision(2)
                 << setw(12) << diff.count()
                 << " seconds.\n"
                 << setw(12) << milliseconds(diff).count()
                 << " milliseconds.\n"
                 << setw(12) << microseconds(diff).count()
                 << " microseconds.\n";
            break;
```

12. Если пользователь сделал опечатку, то позволим ему повторить попытку:

```
        } else {
            cout << "Sorry, your input does not match."
                    " You may try again.\n";
        }
    }
}
```

13. Компиляция и запуск программы дадут следующий результат. Сначала при наличии опечаток программа попросит пользователя ввести корректное слово. После этого она отобразит время, которое потребовалось на то, чтобы ввести его, в трех единицах измерения.

```
$ ./ratio_conversion
Please type the word "C++17" as fast as you can.
```

```
> c+17
Sorry, your input does not match. You may try again.
Please type the word "C++17" as fast as you can.
> C++17
Bravo. You did it in:
        1.48 seconds.
     1480.10 milliseconds.
  1480099.00 microseconds.
```

Как это работает

Несмотря на то что этот раздел посвящен выполнению преобразований между разными единицами измерения времени, сначала нужно выбрать один из трех доступных объектов часов. Как правило, в пространстве имен `std::chrono` можно выбрать между `system_clock`, `steady_clock` и `high_resolution_clock`. Чем они отличаются? Взглянем на их описание (табл. 8.1).

Таблица 8.1

Часы	Характеристики
system_clock	Представляет собой системные «настенные» часы, показывающие реальное время. Они подходят в том случае, если нужно получить местное время
steady_clock	Данные часы являются монотонными, то есть никогда не будут отставать на какой-то промежуток времени. Это может случиться с другими часами, когда их время корректируется на минимальные значения или даже когда время переключается между зимним и летним
high_resolution_clock	Эти часы имеют самый точный период такта, который только может предоставить STL

Поскольку мы определяли продолжительность промежутка времени между двумя абсолютными точками во времени (они хранятся в переменных `tic` и `toc`), нам не нужно знать, были ли эти точки искажены глобально. Даже если часы спешат или опаздывают на 112 лет 5 часов 10 минут и 1 секунду (или другое значение), это не отражается на *разности между ними*. Единственное, что важно, — после того, как мы сохраняем временную точку `tic`, и до того, как сохраняем временную точку `toc`, для часов нельзя выполнить микронастройку (что случается время от времени во многих системах), поскольку это исказит измерение. Согласно данным требованиям, оптимальным выбором является `steady_clock`. Их реализация может быть основана на счетчике временных меток процессора, который всегда монотонно увеличивается с момента запуска системы.

О'кей, теперь, когда мы выбрали правильный объект `time`, можем сохранить временные точки с помощью функции `chrono::steady_clock::now()`. Функция `now` возвращает значение типа `chrono::time_point<chrono::steady_clock>`. Разность

между двумя такими значениями (`toc - tic`) является *временным промежутком*, или *продолжительностью*, имеющей тип `chrono::duration`.

Поскольку данный тип является основным для текущего раздела, все немного усложняется. Рассмотрим интерфейс шаблонного типа `duration` более пристально:

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

Можно изменить значения параметров `Rep` и `Period`. Значение параметра `Rep` объяснить легко: это всего лишь численный тип переменной, который используется для сохранения значения времени. Для существующих в STL единиц измерения времени таковым обычно выступает тип `long long int`. В данном примере мы выбрали тип `double` и благодаря этому можем сохранять по умолчанию значения в секундах, а затем преобразовывать их в милли- или микросекунды. Если у нас есть промежуток времени, равный `1.2345` секунды и имеющий тип `chrono::seconds`, то значение будет округлено до одной целой секунды. Таким образом, нужно сохранить разность между переменными `tic` и `toc` в переменной типа `chrono::microseconds`, а затем преобразовать его в менее точные единицы. Из-за выбора типа `double` для `Rep` можно выполнять преобразование к более и менее точным единицам и терять минимальный объем точности, что не влияет на наш пример.

Мы использовали `Rep = double` для всех единиц измерения времени, поэтому они отличаются значением параметра `Period`:

```
using seconds      = chrono::duration<double>;
using milliseconds = chrono::duration<double,
    ratio_multiply<seconds::period, milli>>;
using microseconds = chrono::duration<double,
    ratio_multiply<seconds::period, micro>>;
```

Секунды — самая простая в описании единица времени, поскольку можно воспользоваться конструкцией `Period = ratio<1>`, другие же придется подстраивать. Поскольку миллисекунда — одна тысячная секунды, мы умножим `seconds::period` (который представляет собой всего лишь функцию-геттер для параметра `Period`) на `milli` — псевдоним типа `std::ratio<1, 1000>` (`std::ratio<a, b>` — это дробное значение `a/b`). Тип `ratio_multiply`, по сути, является *функцией времени компиляции*, которая представляет собой тип, получаемый в результате умножения одного типа `ratio` на другой.

Это может показаться непонятным, так что рассмотрим пример: команда `ratio_multiply<ratio<2, 3>, ratio<4, 5>>` даст результат `ratio<8, 15>`, поскольку `(2/3) * (4/5) = 8/15`.

Полученные описания типов эквивалентны следующим описаниям:

```
using seconds      = chrono::duration<double, ratio<1, 1>>;
using milliseconds = chrono::duration<double, ratio<1, 1000>>;
using microseconds = chrono::duration<double, ratio<1, 1000000>>;
```

После получения этих типов можно легко выполнять преобразования между ними. При наличии промежутка времени `d` с типом `seconds` можно преобразовать его в тип `milliseconds`, передав в конструктор другого типа — `milliseconds(d)`.

Дополнительная информация

В других учебниках и книгах при преобразовании промежутков времени вы могли столкнуться с `duration_cast`. Если у нас есть промежуток времени типа `chrono::milliseconds` и нужно преобразовать его к типу `chrono::hours`, например, то следует написать конструкцию `duration_cast<chrono::hours>(milliseconds_value)`, поскольку данные единицы измерения зависят от *целочисленных* типов. Преобразование точных единиц времени в менее точные приводит к *потере точности*, именно поэтому и нужен `duration_cast`. Для продолжительностей, основанных на типах `double` или `float`, этого не требуется.

Выполняем преобразование между абсолютными и относительными значениями с использованием std::chrono

До C++11 было довольно сложно получить физическое время и *просто вывести* его на экран, поскольку C++ не имела собственной библиотеки для работы с временем. Требовалось всегда вызывать функции библиотеки C, которая выглядит очень архаично, учитывая, что такие вызовы могут быть инкапсулированы в собственные классы.

Начиная с C++11, в STL можно найти библиотеку `chrono`, она значительно упрощает решение задач, связанных с временем.

В этом примере мы возьмем местное время, выведем его на экран и поработаем с ним, добавляя разные смещения, что очень удобно делать с помощью библиотеки `std::chrono`.

Как это делается

В примере мы сохраним текущее время и выведем его на экран. Кроме того, наша программа будет добавлять разные смещения к сохраненному времени и выводить на экран полученные результаты.

1. Сначала идут типичные директивы `include`, затем мы объявляем об использовании по умолчанию пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <chrono>

using namespace std;
```

2. Выведем на экран абсолютные моменты времени. Они будут иметь форму шаблона типа `chrono::time_point`, поэтому просто перегрузим для него оператор выходного потока. Существуют разные способы вывести на экран дату и/или время для заданного момента. Мы применим только стандартное форматирование `%c`. Можно было бы, конечно, также вывести только время, только дату, только год или что-то еще, приходящее на ум. Все преобразования между разными типами до того, как мы сможем использовать `put_time`, будут выглядеть несколько «неаккуратно», но мы провернем это лишь однажды.

```
ostream& operator<<(ostream &os,
              const chrono::time_point<chrono::system_clock> &t)
{
    const auto tt   (chrono::system_clock::to_time_t(t));
    const auto loct (std::localtime(&tt));
    return os << put_time(loct, "%c");
}
```

3. Для секунд, минут, часов и т. д. в STL существуют описания типов. Сейчас мы добавим тип `days`. Это делается легко; нужно лишь специализировать шаблон `chrono::duration`, сославшись на часы и умножив их на 24, поскольку сутки насчитывают 24 часа.

```
using days = chrono::duration<
    chrono::hours::rep,
    ratio_multiply<chrono::hours::period, ratio<24>>>;
```

4. Чтобы наиболее элегантным способом выразить продолжительность длиной в несколько дней, можно определить собственный пользовательский литерал `days`. Теперь можно написать `3_days`, чтобы создать значение, которое представляет собой три дня.

```
constexpr days operator ""_days(unsigned long long h)
{
    return days{h};
}
```

5. В самой программе сделаем снимок момента времени, который затем просто выведем на экран. Это очень легко и удобно, поскольку мы уже реализовали правильную версию перегруженного оператора.

```
int main()
{
    auto now (chrono::system_clock::now());
    cout << "The current date and time is " << now << '\n';
```

6. Сохранив текущее время в переменной `now`, можем добавить к нему произвольные продолжительности и также вывести их на экран. Добавим к текущему времени 12 часов и выведем результат на экран:

```
    chrono::hours chrono_12h {12};
    cout << "In 12 hours, it will be "
         << (now + chrono_12h)<< '\n';
```

7. Объявляя об использовании по умолчанию пространства имен `chrono_literals`, разблокируем все существующие литералы, описывающие продолжительность, для часов, секунд и т. д. Таким образом, можно изящно вывести на экран, какое время было 12 часов 15 минут назад или семь дней назад.

```
    using namespace chrono_literals;
    cout << "12 hours and 15 minutes ago, it was "
         << (now - 12h - 15min) << '\n'
         << "1 week ago, it was "
         << (now - 7_days) << '\n';
}
```

8. Компиляция и запуск программы дадут следующий результат. Поскольку мы использовали в качестве строки форматирования `%c`, получим довольно полное описание в конкретном формате. Поработав с разными строками формата, можем вывести время в любом формате, который нам нравится. Обратите внимание: здесь мы применяем 24-часовой формат.

```
$ ./relative_absolute_times
The current date and time is Fri May  5 13:20:38 2017
In 12 hours, it will be Sat May  6 01:20:38 2017
12 hours and 15 minutes ago, it was Fri May  5 01:05:38 2017
1 week ago, it was Fri Apr 28 13:20:38 2017
```

Как это работает

Мы получили текущий момент времени из `std::chrono::system_clock`. Этот класс часов STL единственный способен преобразовывать свои значения моментов времени в структуру `time`, которая может быть отображена в виде понятной человеку строки описания.

Чтобы вывести на экран такие моменты времени, мы реализовали оператор `<<` для потока вывода:

```
ostream& operator<<(ostream &os,
                    const chrono::time_point<chrono::system_clock> &t)
{
    const auto tt   (chrono::system_clock::to_time_t(t));
    const auto loct (std::localtime(&tt));
    return os << put_time(loct, "%c");
}
```

Здесь мы сначала преобразуем экземпляр типа `chrono::time_point<chrono::system_clock>` к типу `std::time_t`. Значения этого типа можно преобразовать в локальное время, что мы делаем с помощью функции `std::localtime`. Она возвращает указатель на преобразованное значение (не волнуйтесь об управлении памятью, лежащей за данным указателем; это статический объект, и для него память в куче не выделяется), которое мы наконец можем вывести на экран.

Функция `std::put_time` принимает такой объект и строку формата. Строка `"%c"` отображает стандартную строку даты-времени, например `"Sun Mar 12 11:33:40 2017"`.

Мы также могли бы указать строку `"%m/%d/%y"`, и тогда программа вывела бы на экран дату в формате 03/12/17. Весь список существующих форматов времени слишком длинный, он хорошо задокументирован в онлайн-справочнике по C++.

Помимо вывода на экран мы добавили смещения к нашему моменту времени. Это было просто, поскольку можно выразить промежутки времени, такие как *12 часов 15 минут*, в виде `12h + 15min`. Пространство имен `chrono_literals` предоставляет удобные литералы типов для часов (`h`), минут (`min`), секунд (`s`), миллисекунд (`ms`), микросекунд (`us`) и наносекунд (`ns`).

Добавление подобной продолжительности к значению момента времени создает новое значение момента времени, поскольку типы имеют соответствующие перегруженные версии операторов `+` и `-`, именно поэтому так легко добавлять и отображать смещения времени.

Безопасно извещаем о сбое с помощью std::optional

Когда программа общается с внешним миром и полагается на значения, получаемые извне, могут происходить всевозможные сбои.

Это означает вот что: когда мы пишем функцию, которая должна возвращать значение, но также может дать сбой, это нужно отразить с помощью неких изменений в интерфейсе функции. У нас есть несколько вариантов. Посмотрим, как разработать интерфейс функции, которая возвращает строку, но способна дать сбой:

- использовать возвращаемое значение, указывающее на успех, и выходные параметры: `bool get_string(string&);`;
- возвращать указатель (или умный указатель), значение которого можно установить на `nullptr` в случае сбоя: `string* get_string();`;
- генерировать исключение в случае сбоя и оставить сигнатуру функции очень простой: `string get_string();`.

Все эти подходы имеют свои преимущества и недостатки. Начиная с C++17, существует новый тип, который можно применять для решения такой задачи другим способом: `std::optional`. Идея необязательных значений происходит из чисто функциональных языков программирования (там они иногда называются типами `Maybe` (может быть)) и позволяет писать очень изящный код.

Мы можем обернуть в тип `optional` наши собственные типы, чтобы указать на *пустые* или *ошибочные* значения. Ниже мы узнаем, как это делается.

Как это делается

В этом примере мы реализуем программу, которая считывает целые числа, поступающие от пользователя, и складывает их. Поскольку пользователь всегда может ввести случайные символы вместо чисел, мы увидим, как тип `optional` способен улучшить обработку ошибок.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <optional>

using namespace std;
```

2. Определим целочисленный тип, который, *может быть*, содержит значение. Нам идеально подойдет тип `std::optional`. Обернув любой тип в тип `optional`, задаем ему еще одно возможное состояние, которое отражает тот факт, что тип *не имеет* значения:

```
using oint = optional<int>;
```

3. Определив необязательный целочисленный тип, можем выразить тот факт, что функция, которая возвращает целое число, тоже способна дать сбой. Если мы возьмем целое число из пользовательских входных данных, то существует вероятность сбоя, поскольку пользователь может ввести какой-то другой символ, несмотря на наше приглашение. Возврат целого числа, обернутого в тип `optional`, идеально решает эту проблему. Если нам удалось считать целое число, то передадим его конструктору типа `optional<int>`. В противном случае вернем экземпляр типа `optional`, созданный с помощью конструктора по умолчанию, что говорит о сбое или пустоте.

```
oint read_int()
{
    int i;
    if (cin >> i) { return {i}; }
    return {};
}
```

4. Наши возможности не ограничиваются возвратом целых чисел из функций, способных дать сбой. Что если мы подсчитаем сумму двух целых чисел, которые могут не содержать значений? Мы получим реальную численную сумму только в том случае, если оба операнда содержат значения. В любом другом нужно вернуть пустую переменную типа `optional`. Эту функцию следует немного пояснить: неявно преобразуя переменные типа `optional<int>`, `a` и `b`, к булевым выражениям (с помощью конструкций `!a` и `!b`), мы узнаем, содержат ли они реальные значения. Если да, то можно получить к ним доступ как к указателям или итераторам, просто разыменовав их с использованием конструкций `*a` и `*b`:

```
oint operator+(oint a, oint b)
{
    if (!a || !b) { return {}; }
    return {*a + *b};
}
```

5. Сложение обычного целого числа и целого числа, которое может не содержать значений, следует той же логике:

```
oint operator+(oint a, int b)
{
```

```
    if (!a) { return {}; }
    return {*a + b};
}
```

6. Теперь напишем программу, совершающую некие действия с целыми числами, которые могут не содержать значений. Пригласим пользователя ввести два числа:

```
int main()
{
    cout << "Please enter 2 integers.\n> ";
    auto a {read_int()};
    auto b {read_int()};
```

7. Сложим эти числа, а затем добавим к полученной сумме значение `10`. Поскольку `a` и `b` могут не иметь значений, `sum` также будет необязательной целочисленной переменной:

```
    auto sum (a + b + 10);
```

8. Если переменные `a` и/или `b` не содержат значений, то `sum` не может иметь значения. Положительный момент заключается в том, что не нужно явно проверять значения переменных `a` и `b`. Сложение пустых экземпляров типа `optional` — полностью корректное поведение, поскольку мы определили безопасный оператор `+` для этих типов. Таким образом можно произвольно сложить несколько пустых необязательных экземпляров, а проверять нужно только полученный экземпляр. Если он содержит значение, то мы можем безопасно получить к нему доступ и вывести его на экран:

```
    if (sum) {
       cout << *a << " + " << *b << " + 10 = "
            << *sum << '\n';
```

9. Если пользователь вводит не числа, то мы сообщаем об ошибке:

```
    } else {
       cout << "sorry, the input was "
               "something else than 2 numbers.\n";
    }
}
```

10. На этом все. Компиляция и запуск программы дадут следующий результат:

```
$ ./optional
Please enter 2 integers.
> 1 2
1 + 2 + 10 = 13
```

11. Если мы запустим программу снова и введем не числа, то увидим сообщение об ошибке, подготовленное нами для таких случаев:

```
$ ./optional
Please enter 2 integers.
> 2 z
sorry, the input was something else than 2 numbers.
```

Как это работает

Работать с типом `optional` очень просто и удобно. Если мы хотим, чтобы любой тип `T` имел дополнительное состояние, указывающее на возможный сбой, то можем обернуть его в тип `std::optional<T>`.

Когда мы получаем экземпляр подобного типа откуда бы ни было, нужно проверить, он пуст или же содержит значение. Здесь поможет функция `optional::has_value()`. Если она возвращает значение `true`, то можно получить доступ к этому значению. Это позволяет сделать вызов `T& optional::value()`.

Вместо того чтобы всегда использовать конструкции `if (x.has_value()) {...}` и `x.value()`, можно применить конструкции `if (x) {...}` и `*x`. В типе `std::optional` определено неявное преобразование к типу `bool` и `operator*` так, что работа с типом `optional` похожа на работу с указателем.

Существует еще один удобный вспомогательный оператор — это `->`. Если у нас есть тип `struct Foo { int a; string b; }` и нужно получить доступ к одному из его членов с помощью переменной `x` типа `optional<Foo>`, то можно написать конструкцию `x->a` или `x->b`. Конечно, сначала следует проверить, содержит ли `x` значение.

Если мы попробуем получить доступ к объекту типа `optional`, который не содержит значения, то будет сгенерирована ошибка `std::logic_error`. Таким образом, нельзя работать с большим количеством необязательных экземпляров, не проверяя их. С помощью блока `try-catch` можно писать код в следующей форме:

```
cout << "Please enter 3 numbers:\n";

try {
    cout << "Sum: "
         << (*read_int() + *read_int() + *read_int())
         << '\n';
} catch (const std::bad_optional_access &) {
    cout << "Unfortunately you did not enter 3 numbers\n";
}
```

Еще одним трюком для типа `std::optional` является `optional::value_or`. Это поможет, когда мы хотим взять необязательное значение и, если оно окажется пустым, откатить его к значению, заданному по умолчанию. Можно решить эту задачу с помощью одной емкой строки `x = optional_var.value_or(123)`, где `123` — значение по умолчанию.

Применяем функции для кортежей

Начиная с C++11, STL предоставляет тип `std::tuple`. Он позволяет время от времени *объединять* несколько значений в одну переменную и получать к ним доступ. Кортежи есть во многих языках программирования, и в некоторых примерах данной книги мы уже работали с этим типом, поскольку он крайне гибок.

Однако иногда мы помещаем в кортеж значения, а затем хотим вызвать функции, передав в них его отдельные члены. Распаковывать члены по отдельности для

каждого аргумента функции очень утомительно (а кроме того, могут возникнуть ошибки, если где-то вкрадется опечатка). Это выглядит так: `func(get<0>(tup), get<1>(tup), get<2>(tup), ...);`.

Ниже мы рассмотрим, как упаковывать значения в кортежи и ловко распаковывать из них, чтобы вызвать функции, которые не знают о кортежах.

Как это делается

В этом примере мы реализуем программу, которая упаковывает значения в кортежи и распаковывает из них. Затем увидим, как вызывать функции, ничего не знающие о кортежах, и передавать в них значения из кортежей.

1. Сначала включим множество заголовочных файлов и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <tuple>
#include <functional>
#include <string>
#include <list>

using namespace std;
```

2. Определим функцию, которая принимает несколько параметров, описывающих студента, и выводит их на экран. Многие устаревшие интерфейсы и интерфейсы функций языка C выглядят похоже:

```
static void print_student(size_t id, const string &name, double gpa)
{
    cout << "Student " << quoted(name)
         << ", ID: "   << id
         << ", GPA: "  << gpa << '\n';
}
```

3. В самой программе определим тип кортежа динамически и заполним его осмысленными данными о студентах:

```
int main()
{
    using student = tuple<size_t, string, double>;
    student john {123, "John Doe"s, 3.7};
```

4. Чтобы вывести такой объект на экран, можем разбить его на отдельные члены и вызвать функцию `print_student` для этих отдельных переменных:

```
    {
        const auto &[id, name, gpa] = john;
        print_student(id, name, gpa);
    }
    cout << "-----\n";
```

5. Создадим несколько студентов в виде списка инициализаторов для кортежей:

```
auto arguments_for_later = {
    make_tuple(234, "John Doe"s,  3.7),
    make_tuple(345, "Billy Foo"s, 4.0),
    make_tuple(456, "Cathy Bar"s, 3.5),
};
```

6. Мы все еще можем относительно комфортно вывести их на экран, но, чтобы разбить кортеж на части, следует знать, сколько элементов в нем содержится. Если нужно писать подобный код, то понадобится также реструктурировать его в случае изменения интерфейса вызова функции:

```
for (const auto &[id, name, gpa] : arguments_for_later) {
    print_student(id, name, gpa);
}
cout << "-----\n";
```

7. Можно сделать лучше. Даже не зная типов аргументов функции `print_student` или количества членов кортежа, описывающего студентов, можно направить содержимое кортежа непосредственно в функцию с помощью `std::apply`. Она принимает указатель на функцию или объект функции и кортеж, а затем *распаковывает* кортеж, чтобы вызвать функцию, передав в нее в качестве параметров члены кортежа:

```
apply(print_student, john);
cout << "-----\n";
```

8. Конечно, все это прекрасно работает и в цикле:

```
    for (const auto &args : arguments_for_later) {
        apply(print_student, args);
    }
    cout << "-----\n";
}
```

9. Компиляция и запуск программы покажут, что работают оба подхода, как мы и предполагали:

```
$ ./apply_functions_on_tuples
Student "John Doe", ID: 123, GPA: 3.7
-----
Student "John Doe", ID: 234, GPA: 3.7
Student "Billy Foo", ID: 345, GPA: 4
Student "Cathy Bar", ID: 456, GPA: 3.5
-----
Student "John Doe", ID: 123, GPA: 3.7
-----
Student "John Doe", ID: 234, GPA: 3.7
Student "Billy Foo", ID: 345, GPA: 4
Student "Cathy Bar", ID: 456, GPA: 3.5
-----
```

Как это работает

Функция `std::apply` — это вспомогательная функция времени компиляции, которая позволяет нам работать, не имея сведений обо всех типах данных, которые появляются в нашем коде.

Допустим, у нас есть кортеж `t` со значениями `(123, "abc"s, 456.0)`. Он имеет тип `tuple<int, string, double>`. Вдобавок предположим, что у нас есть функция `f` с сигнатурой `int f(int, string, double)` (типы также могут быть ссылками).

Затем можно написать конструкцию `x = apply(f, t)`, которая приведет к вызову функции `x = f(123, "abc"s, 456.0)`. Метод `apply` даже не возвращает результат работы функции `f`.

Быстрое создание структур данных с помощью std::tuple

Взглянем на простой пример использования кортежей, с которым мы уже сталкивались. Мы можем определить следующую структуру, чтобы просто объединить некоторые переменные в одну сущность:

```
struct Foo {
    int a;
    string b;
    float c;
};
```

Вместо того чтобы определять структуру, как было сделано в предыдущем примере, можно также определить кортеж:

```
using Foo = tuple<int, string, float>;
```

Получить доступ к его элементам можно по порядковому номеру типа из списка типов. Для получения доступа к первому члену кортежа `t` напишем конструкцию `std::get<0>(t)`; для получения доступа ко второму члену — `std::get<1>` и т. д. Если порядковый номер слишком велик, то компилятор безопасно сгенерирует ошибку.

На протяжении этой книги мы уже использовали возможности декомпозиции C++17 для кортежей. Она позволяет быстро разбить кортеж на элементы, просто написав `auto [a, b, c] = some_tuple`, чтобы получить доступ к отдельным элементам.

Композиция и декомпозиция отдельных структур данных — не единственная возможность, которую предоставляют кортежи. Мы также можем конкатенировать или разбивать кортежи. В этом разделе мы поэкспериментируем с данными функциями, чтобы узнать, как они работают.

Как это делается

В этом примере мы напишем программу, которая может динамически вывести на экран любой кортеж. Вдобавок напишем функцию, способную *объединять* кортежи.

1. Сначала включим несколько заголовочных файлов, а затем объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <tuple>
#include <list>
#include <utility>
#include <string>
#include <iterator>
#include <numeric>
#include <algorithm>

using namespace std;
```

2. Поскольку мы будем работать с кортежами, было бы интересно отобразить их содержимое. Поэтому сейчас реализуем очень обобщенную функцию, которая может выводить на экран любые кортежи. Функция принимает ссылку на поток вывода `os`, которая будет использована для вывода данных на экран, и список аргументов переменной длины, содержащий все члены кортежа. Мы разбили на части все аргументы в первом элементе и поместили их в аргумент `v`, а остальные поместили в наборе параметров `vs...`:

```
template <typename T, typename ... Ts>
void print_args(ostream &os, const T &v, const Ts &...vs)
{
    os << v;
```

3. Если в наборе параметров еще есть аргументы, то они выводятся на экран, притом их разделяет символ `", "` — используется прием с развертыванием `initializer_list`. Мы говорили о нем в главе 4.

```
    (void)initializer_list<int>{((os << ", " << vs), 0)...};
}
```

4. Теперь можно выводить на экран произвольное количество аргументов с помощью конструкции `print_args(cout, 1, 2, "foo", 3, "bar")`, например. Но мы пока не трогали кортежи. Для вывода кортежей на экран переопределим оператор потока вывода `<<`, чтобы он работал с кортежами, реализовав шаблонную функцию, которая соответствует любым специализациям для кортежа:

```
template <typename ... Ts>
ostream& operator<<(ostream &os, const tuple<Ts...> &t)
{
```

5. Сейчас все станет чуть сложнее. Сначала мы воспользуемся лямбда-выражением, которое принимает произвольно большое количество параметров. При вызове выражение добавляет аргумент `os` в начало списка данных аргументов, а затем вызывает функцию `print_args`, передавая полученный новый список аргументов. Это значит, что вызов `capt_tup(...`*`некоторые параметры`*`...)` преобразуется в вызов `print_args(os, ...`*`некоторые параметры`*`...)`.

```
auto print_to_os ([&os](const auto &...xs) {
    print_args(os, xs...);
});
```

6. Теперь можем выполнить распаковку кортежа. Для этого воспользуемся методом `std::apply`. Все значения будут извлечены из кортежа и представлены как аргументы функции для той функции, которую мы предоставили в качестве первого аргумента. Это значит, что если у нас есть кортеж `t = (1, 2, 3)` и мы вызываем метод `apply(capt_tup, t)`, то эти действия приведут к вызову функции `capt_tup(1, 2, 3)`, что, в свою очередь, повлечет вызов `print_args(os, 1, 2, 3)`. Это именно то, что нужно. Дополнительно окружим операцию вывода скобками:

```
    os << "(";
    apply(print_to_os, t);
    return os << ")";
}
```

7. О'кей, мы написали сложный фрагмент кода, который сделает нашу жизнь гораздо проще, если мы захотим вывести кортеж на экран. Но с помощью кортежей мы можем сделать больше. Допустим, напишем функцию, принимающую в качестве аргумента итерабельный диапазон данных, например вектор или список чисел. Эта функция проитерирует по данному диапазону и вернет *сумму* всех его чисел, а также *минимальное*, *максимальное* и *среднее* значения. Упаковывая эти четыре значения в кортеж, можем вернуть их как один объект, не определяя дополнительный тип структуры.

```
template <typename T>
tuple<double, double, double, double>
sum_min_max_avg(const T &range)
{
```

8. Функция `std::minmax_element` возвращает пару итераторов, указывающих на минимальный и максимальный элементы входного диапазона. Метод `std::accumulate` складывает все значения в данном диапазоне. Это все нужно, чтобы вернуть четыре значения, которые позже будут помещены в кортеж!

```
    auto min_max (minmax_element(begin(range), end(range)));
    auto sum     (accumulate(begin(range), end(range), 0.0));
    return {sum, *min_max.first, *min_max.second,
            sum / range.size()};
}
```

9. Перед реализацией основной программы создадим последнюю волшебную вспомогательную функцию. Я называю ее волшебной, поскольку она на первый

взгляд выглядит очень сложной, но если понять принцип ее работы, она покажется очень удобным помощником. Она сгруппирует два кортежа. Это значит, что если мы передадим ей кортежи `(1, 2, 3)` и `('a', 'b', 'c')`, то она вернет кортеж `(1, 'a', 2, 'b', 3, 'c')`.

```
template <typename T1, typename T2>
static auto zip(const T1 &a, const T2 &b)
{
```

10. Мы подобрались к самым сложным строкам данного примера. Создадим объект функции `z`, принимающий произвольное количество аргументов. Затем он возвращает другой объект функции, который захватывает все эти аргументы в набор параметров `xs`, а также принимает произвольное количество аргументов. Забудем об этом на минуту. С помощью упомянутого внутреннего объекта функции можно получить доступ к обоим спискам аргументов в виде наборов параметров `xs` и `ys`. А теперь взглянем, что именно происходит с этими наборами параметров. Вызов `make_tuple(xs, ys)...` группирует наборы параметров поэлементно. То есть при наличии наборов `xs = 1, 2, 3` и `ys = 'a', 'b', 'c'` он вернет новый набор параметров `(1, 'a'), (2, 'b'), (3, 'c')`. Данный набор представляет собой разделенный запятыми список, состоящий из трех кортежей. Чтобы объединить их в *один* кортеж, используем функцию `std::tuple_cat`, которая принимает произвольное количество кортежей и упаковывает их в один. Таким образом получим кортеж `(1, 'a', 2, 'b', 3, 'c')`.

```
    auto z ([](auto ...xs) {
        return [xs...](auto ...ys) {
            return tuple_cat(make_tuple(xs, ys) ...);
        };
    });
```

11. Последний шаг состоит в том, чтобы распаковать все значения из входных кортежей `a` и `b` и поместить их в `z`. Вызов `apply(z, a)` помещает все значения из `a` в набор параметров `xs`, а вызов `apply(..., b)` помещает все значения `b` в набор параметров `ys`. Полученный кортеж представляет собой остальные сгруппированные кортежи, которые мы и возвращаем вызывающей стороне:

```
    return apply(apply(z, a), b);
}
```

12. Мы написали довольно много строк кода для вспомогательных функций. Теперь воспользуемся ими. Сначала создадим произвольные кортежи. `student` содержит идентификатор, имя и средний балл студента. `student_desc` включает строки, которые описывают значение этих полей, в форме, доступной человеку. `std::make_tuple` — приятный помощник, поскольку определяет тип всех аргументов и создает подходящий тип кортежа:

```
int main()
{
    auto student_desc (make_tuple("ID", "Name", "GPA"));
    auto student      (make_tuple(123456, "John Doe", 3.7));
```

13. Выведем на экран все, чем располагаем. Сделать это очень легко, поскольку мы только что реализовали соответствующую перегруженную версию оператора <<.

```
cout << student_desc << '\n'
     << student      << '\n';
```

14. Кроме того, можем сгруппировать оба кортежа динамически с помощью std::tuple_cat и вывести их на экран следующим образом:

```
cout << tuple_cat(student_desc, student) << '\n';
```

15. Мы можем создать и новый *сгруппированный* кортеж, задействуя нашу функцию zip, и вывести его на экран:

```
auto zipped (zip(student_desc, student));
cout << zipped << '\n';
```

16. Не будем забывать и о функции sum_min_max_avg. Создадим список инициализации, который содержит некие числа, и передадим его в эту функцию. Чтобы слегка усложнить задачу, создадим еще один кортеж такого же размера, содержащий строки с описанием. Группируя эти кортежи, получим «аккуратные», чередующиеся выходные данные, которые увидим при запуске программы:

```
    auto numbers = {0.0, 1.0, 2.0, 3.0, 4.0};
    cout << zip(
            make_tuple("Sum", "Minimum", "Maximum", "Average"),
            sum_min_max_avg(numbers))
         << '\n';
}
```

17. Компиляция и запуск программы дадут следующий результат. Первые две строки представляют отдельные кортежи student и student_desc. Третья строка — это комбинация кортежей, которую мы получили с помощью вызова tuple_cat. Четвертая содержит сгруппированный кортеж для студента. В последней строке видим сумму, а также минимальное, максимальное и среднее значения для созданного нами списка чисел. Благодаря группировке очень легко понять смысл каждого значения.

```
$ ./tuple
(ID, Name, GPA)
(123456, John Doe, 3.7)
(ID, Name, GPA, 123456, John Doe, 3.7)
(ID, 123456, Name, John Doe, GPA, 3.7)
(Sum, 10, Minimum, 0, Maximum, 4, Average, 2)
```

Как это работает

Отдельные фрагменты кода, представленные в этом разделе, довольно сложны. Мы написали реализацию operator<< для кортежей, которая выглядит очень сложной, но зато поддерживает все виды кортежей, содержащих выводимые типы. Далее мы реализовали функцию sum_min_max_avg, возвращающую кортеж. Еще одним сложным моментом является функция zip.

Самой простой частью была функция `sum_min_max_avg`. Ее идея заключается в следующем: при реализации функции, которая возвращает экземпляр типа `tuple<Foo, Bar, Baz> f()`, можно просто написать `return {foo_instance, bar_instance, baz_instance};` внутри этой функции, чтобы создать такой кортеж. Если вам трудно понять использованные нами алгоритмы STL, то, вероятно, следует обратиться к главе 5, где мы уже рассмотрели их подробнее.

Остальной код настолько сложен, что мы посвятим конкретным вспомогательным функциям отдельные подразделы.

operator<< для кортежей

До работы с `operator<<` для потоков вывода мы реализовали функцию `print_args`. Из-за своей природы он принимает произвольное количество аргументов разных типов до тех пор, пока первым из них является экземпляр типа `ostream`:

```
template <typename T, typename ... Ts>
void print_args(ostream &os, const T &v, const Ts &...vs)
{
    os << v;
    (void)initializer_list<int>{((os << ", " << vs), 0)...};
}
```

Эта функция выводит на экран первый элемент `v`, а затем все элементы из набора параметров `vs`. Мы выводим на экран первый элемент отдельно, поскольку хотим, чтобы все элементы были разделены запятыми, но не хотим, чтобы строка начиналась или заканчивалась запятой (например, `"1, 2, 3, "` или `", 1, 2, 3"`). Мы узнали о приеме с разворачиванием `initializer_list` из примера «Вызов нескольких функций с одними и теми же входными данными» главы 4. Подготовив эту функцию, мы получили все необходимое для вывода кортежей на экран. Наша реализация оператора `<<` выглядит следующим образом:

```
template <typename ... Ts>
ostream& operator<<(ostream &os, const tuple<Ts...> &t)
{
    auto capt_tup ([&os](const auto &...xs) {
        print_args(os, xs...);
    });
    os << "(";
    apply(capt_tup, t);
    return os << ")";
}
```

Первое, что мы делаем, — это определяем объект функции `capt_tup`. Вызов `capt_tup(foo, bar, whatever)` приводит к вызову `print_args(os, foo, bar, whatever)`. Данный объект функции делает только одно: добавляет к списку аргументов объект потока вывода `os`.

После этого мы воспользуемся методом `std::apply`, чтобы распаковать все элементы из кортежа `t`. Если данный шаг выглядит слишком сложным, обратитесь к предыдущему примеру — он демонстрирует работу метода `std::apply`.

Функция zip для кортежей

Функция `zip` принимает два кортежа, но выглядит весьма сложной, несмотря на то что имеет очень четкую реализацию:

```
template <typename T1, typename T2>
auto zip(const T1 &a, const T2 &b)
{
    auto z ([](auto ...xs) {
        return [xs...](auto ...ys) {
            return tuple_cat(make_tuple(xs, ys) ...);
        };
    });
    return apply(apply(z, a), b);
}
```

Для лучшего понимания этого кода представьте, что кортеж `a` содержит значения `1, 2, 3`, а кортеж `b` — значения `'a', 'b', 'c'`.

В данном случае вызов `apply(z, a)` приведет к вызову `z(1, 2, 3)`. Он вернет объект функции, который захватит значения `1, 2, 3` в набор параметров `xs`. В момент вызова с помощью `apply(z(1, 2, 3), b)` этот объект получит значения `'a', 'b', 'c'`, помещенные в набор параметров `ys`. По сути, действие аналогично прямому вызову `z(1, 2, 3)('a', 'b', 'c')`.

О'кей, что произойдет теперь, когда у нас есть значения `xs = (1, 2, 3)` и `ys = ('a', 'b', 'c')`? Выражение `tuple_cat(make_tuple(xs, ys) ...)` сделает следующее (рис. 8.1).

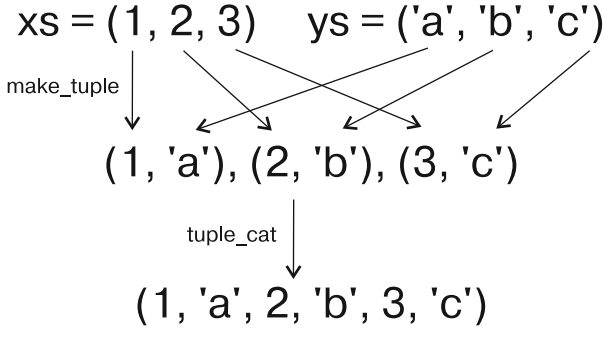

Рис. 8.1

Сначала элементы из наборов `xs` и `ys` будут сгруппированы попарно. Это «попарное чередование» выполняется в вызове `make_tuple(xs, ys) ...`. Сначала мы получим список кортежей переменной длины по два элемента в каждом. Чтобы получить *один большой* кортеж, мы применяем вызов `tuple_cat` — в результате получаем большой сконкатенированный кортеж, содержащий все члены исходных кортежей, которые чередуются.

Замена void* с использованием std::any для повышения безопасности типов

Может случиться так: нам понадобится сохранить элементы *любого* типа в переменной. Для такой переменной следует проверить, содержит ли она *что-либо*, и если да, то нужно определить, *что именно*. Все это надо сделать безопасно для типов.

В прошлом мы имели возможность хранить указатели на различные объекты в указателе типа `void*`. Такой указатель сам по себе не может сказать, на какой объект ссылается, поэтому нужно вручную создать некий дополнительный механизм, который сообщит, чего стоит ожидать. Данное решение приводит к созданию некрасивого и небезопасного кода.

Еще одним дополнением к STL в C++17 является тип `std::any`. Он разработан для того, чтобы хранить переменные любого вида, и предоставляет средства, которые позволяют выполнить проверку, безопасную для типов, и получить доступ к данным.

В текущем разделе мы поработаем с этим вспомогательным типом для того, чтобы несколько лучше его понять.

Как это делается

В этом примере мы реализуем функцию, которая пробует вывести на экран какие-либо данные. В качестве ее типа аргумента служит тип `std::any`.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <list>
#include <any>
#include <iterator>

using namespace std;
```

2. Чтобы сократить объем использования угловых скобок в следующей программе, определим псевдоним для типа `list<int>` и будем применять его впоследствии:

```
using int_list = list<int>;
```

3. Реализуем функцию, которая утверждает, что может вывести на экран любые данные. Обещание заключается вот в чем: она выводит любые данные, предоставленные как аргумент в виде переменной `std::any`:

```
void print_anything(const std::any &a)
{
```

4. Первое, что нужно сделать, — проверить, содержит ли аргумент *какие-то данные*, или же это пустой экземпляр типа `any`. Если он пуст, то нет смысла пытаться определить, как его выводить на экран.

```
    if (!a.has_value()) {
        cout << "Nothing.\n";
```

5. Если он не пуст, то можно попробовать сравнивать его с разными типами до тех пор, пока не получим совпадение. Первым типом послужит тип `string`. Если это строка, то можно выполнить преобразование `a` к ссылке `string` с помощью `std::any_cast` и просто вывести его на экран. Из соображений эстетики мы поместим строку в кавычки:

```
    } else if (a.type() == typeid(string)) {
        cout << "It's a string: "
             << quoted(any_cast<const string&>(a)) << '\n';
```

6. Если это не `string`, то может быть `int`. При совпадении данного типа можно использовать преобразование `any_cast<int>`, чтобы получить реальное значение `int`:

```
    } else if (a.type() == typeid(int)) {
        cout << "It's an integer: "
             << any_cast<int>(a) << '\n';
```

7. `std::any` работает не только для простых типов наподобие `string` и `int`. В переменную `any` можно поместить и ассоциативный массив, список или экземпляр любого другого сложного типа данных. Посмотрим, являются ли входные данные списком целых чисел, и если да, то можем вывести его точно так же, как и любой другой список:

```
    } else if (a.type() == typeid(int_list)) {
        const auto &l (any_cast<const int_list&>(a));
        cout << "It's a list: ";
        copy(begin(l), end(l),
             ostream_iterator<int>{cout, ", "});
        cout << '\n';
```

8. Если не подошел ни один из перечисленных типов, то у нас закончатся догадки. В таком случае просто сдадимся и скажем пользователю, что не знаем, как выводить эти данные на экран:

```
    } else {
        cout << "Can't handle this item.\n";
    }
}
```

9. В функции `main` можем вызвать эту функцию с произвольными типами, с пустой переменной типа `any` с помощью `{}` или передать ей строку `"abc"` или целое число. Поскольку экземпляр типа `std::any` может быть создан на основе этих типов неявно, не возникает задержек, связанных с синтаксисом. Мы даже можем создать целый список и передать его в эту функцию:

```
int main()
{
    print_anything({});
    print_anything("abc"s);
    print_anything(123);
    print_anything(int_list{1, 2, 3});
```

10. Если мы будем помещать объекты, копировать которые действительно дорого, в переменную типа `any`, то можем также выполнить конструкцию *«на месте» (in-place)*. Попробуем сделать это для нашего списочного типа. Выражение `in_place_type_t<int_list>{}` представляет собой пустой объект, дающий конструктору типа `any` достаточно информации о том, что мы собираемся создать. Второй параметр, `{1, 2, 3}`, — просто список инициализации, который будет передан в `int_list`, будучи встроенным в переменную типа `any` с целью создания объекта. Это способ избежать ненужного копирования или перемещения.

```
    print_anything(any(in_place_type_t<int_list>{}, {1, 2, 3}));
}
```

11. Компиляция и запуск программы дадут следующие результаты, они полностью соответствуют нашим ожиданиям:

```
$ ./any
Nothing.
It's a string: "abc"
It's an integer: 123
It's a list: 1, 2, 3,
It's a list: 1, 2, 3,
```

Как это работает

Тип `std::any` в чем-то похож на тип `std::optional` — он поддерживает метод `has_value()`, который говорит, содержит ли экземпляр значение. Но, помимо этого, он может содержать *все что угодно*, вследствие чего с ним работать немного сложнее, нежели с типом `optional`.

Прежде чем получать доступ к содержимому переменной типа `any`, нужно определить, *какого* типа хранящееся в ней значение, а затем *преобразовать* данные к этому типу.

Определить тип значения можно с помощью следующего сравнения: `x.type() == typeid(T)`. Если оно возвращает результат `true`, то можно использовать преобразование `any_cast`, чтобы получить содержимое.

Обратите внимание: `any_cast<T>(x)` возвращает *копию* внутреннего значения. Если нужно получить *ссылку*, чтобы избежать копирования сложных объектов, то следует использовать конструкцию `any_cast<T&>(x)`. Именно это мы и сделали, когда получали доступ к объектам типа `string` или `list<int>` в коде данного раздела.

Если мы преобразуем экземпляр к неправильному типу, будет сгенерировано исключение std::bad_any_cast.

Хранение разных типов с применением std::variant

В языке C++ для создания типов можно использовать не только примитивы `struct` и `class`. Если нужно выразить, что какие-то переменные могут содержать значения типа `A` либо значения типа `B` (или `C`, или любого другого), то на помощь придут объединения. Проблема с объединениями заключается в том, что они не могут сказать, для хранения каких типов были инициализированы.

Рассмотрим следующий код:

```
union U {
    int    a;
    char  *b;
    float  c;
};

void func(U u) { std::cout << u.b << '\n'; }
```

Допустим, мы вызовем функцию `func` для объединения, которое было инициализировано так, чтобы хранить в нем целое число в члене `a`. Тогда ничто не помешает нам получить доступ к нему так, как если бы оно было инициализировано способом, позволяющим хранить в нем указатель на строку в члене `b`. Из подобного кода могут появиться самые разнообразные ошибки. Прежде чем мы поместим в наше объединение вспомогательную переменную, которая скажет нам, для чего оно было инициализировано, можем воспользоваться типом `std::variant`, появившимся в C++17.

Тип `variant`, по сути, представляет собой *обновленную* версию типа `union`. Он не использует кучу, поэтому настолько же эффективно задействует память и время, как и решение, основанное на объединениях, так что нам нет нужды реализовывать его самостоятельно. Тип может хранить все что угодно, кроме ссылок массивов или объектов типа `void`.

В этом разделе мы создадим программу, которая задействует тип `variant`.

Как это делается

В этом примере мы реализуем программу, которая уже знакома с типами `cat` и `dog` и сохраняет смешанный список экземпляров обоих типов, не используя полиморфизм.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

   ```
   #include <iostream>
   #include <variant>
   #include <list>
   ```

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

using namespace std;
```

2. Далее реализуем два класса, имеющих схожий инструментарий, но не связанных друг с другом, что отличает их от классов, которые, скажем, наследуют от одного интерфейса или похожих интерфейсов. Первый класс — это класс `cat`. Объект класса `cat` имеет имя и может сказать *«мяу»* (meow):

```
class cat {
    string name;

public:
    cat(string n) : name{n} {}
    void meow() const {
        cout << name << " says Meow!\n";
    }
};
```

3. Второй класс — это класс `dog`. Объект класса `dog`, конечно, может сказать не *«мяу»*, а *«гав»* (woof):

```
class dog {
    string name;
public:
    dog(string n) : name{n} {}
    void woof() const {
        cout << name << " says Woof!\n";
    }
};
```

4. Теперь можно определить тип `animal`, он будет представлять собой псевдоним типа `std::variant<dog, cat>`. По сути, он работает как старое доброе объединение, но имеет все дополнительные средства, предоставленные типом `variant`:

```
using animal = variant<dog, cat>;
```

5. Прежде чем писать основную программу, нужно реализовать два вспомогательных элемента. Одним из них является предикат `animal`. Вызвав `is_type<cat>(...)` или `is_type<dog>(...)`, можно определить, какого типа данные содержатся в экземпляре типа `animal`. Реализация просто вызывает функцию `holds_alternative`, которая, по сути, является обобщенной функцией-предикатом для типа `variant`:

```
template <typename T>
bool is_type(const animal &a) {
    return holds_alternative<T>(a);
}
```

6. Вторым вспомогательным элементом является структура, которая ведет себя как объект функции. Это двойной объект функции, поскольку он дважды реализует оператор `()`. Одна из реализаций — перегруженная версия, принимающая экземпляры типа `dog`, вторая же принимает экземпляры типа `cat`. Для этих типов она просто вызывает функции `woof` или `meow`:

```
struct animal_voice
{
    void operator()(const dog &d) const { d.woof(); }
    void operator()(const cat &c) const { c.meow(); }
};
```

7. Воспользуемся результатами нашего труда. Сначала определим список переменных типа `animal` и заполним его экземплярами типов `cat` и `dog`:

```
int main()
{
    list<animal> l {cat{"Tuba"}, dog{"Balou"}, cat{"Bobby"}};
```

8. Теперь трижды выведем на экран содержимое списка, каждый раз новым способом. Один из них заключается в использовании `variant::index()`. Поскольку `animal` является псевдонимом для `variant<dog, cat>`, возвращаемое значение `0` означает, что переменная хранит экземпляр типа `dog`. Значение индекса `1` говорит о том, что это экземпляр типа `cat`. Здесь важен порядок типов в специализации `variant`. В блоке `switch case` мы получаем доступ к `variant` с помощью вызова `get<T>` для получения экземпляра типа `cat` или `dog`, хранящегося внутри:

```
    for (const animal &a : l) {
        switch (a.index()) {
        case 0:
            get<dog>(a).woof();
            break;
        case 1:
            get<cat>(a).meow();
            break;
        }
    }
    cout << "-----\n";
```

9. Вместо того чтобы использовать численный индекс типа, можно также явно запросить каждый тип. Вызов `get_if<dog>` возвращает указатель на объект типа `do` на внутренний экземпляр типа `dog`. Если такого экземпляра внутри нет, то указатель равен `null`. Таким образом, мы можем попробовать получать разные типы до тех пор, пока не преуспеем.

```
    for (const animal &a : l) {
        if (const auto d (get_if<dog>(&a)); d) {
            d->woof();
        } else if (const auto c (get_if<cat>(&a)); c) {
            c->meow();
        }
    }
    cout << "-----\n";
```

10. Последний — и самый элегантный вариант — это `variant::visit`. Данная функция принимает объект функции и экземпляр типа `variant`. Объект функции должен реализовывать разные перегруженные версии для всех вероятных типов, которые может хранить `variant`. Ранее мы реализовали структуру, имеющую необходимые перегруженные версии оператора `()`, поэтому можем использовать ее здесь:

```
    for (const animal &a : l) {
        visit(animal_voice{}, a);
    }
    cout << "-----\n";
```

11. Наконец подсчитаем количество экземпляров типов `cat` и `dog` в списке. Предикат `is_type<T>` может быть специализирован для типов `cat` и `dog`, а затем использован в комбинации с `std::count_if`, чтобы получить количество экземпляров этого типа:

```
    cout << "There are "
         << count_if(begin(l), end(l), is_type<cat>)
         << " cats and "
         << count_if(begin(l), end(l), is_type<dog>)
         << " dogs in the list.\n";
}
```

12. После компиляции и запуска программы на экране будет список, выведенный три раза. Затем мы увидим, что предикаты `is_type`, объединенные с `count_if`, тоже работают хорошо:

```
$ ./variant
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
There are 2 cats and 1 dogs in the list.
```

Как это работает

Тип `std::variant` похож на тип `std::any`, поскольку они оба могут содержать объекты разных типов, и нужно определять во время работы программы, что именно в них хранится, прежде чем получить доступ к их содержимому.

С другой стороны, тип `std::variant` отличается от `std::any` тем, что мы должны объявлять, экземпляры каких типов он может хранить в виде списка шаблонных типов. Экземпляр типа `std::variant<A, B, C>` *должен* хранить один

экземпляр типа A, B или C. Нельзя сделать так, чтобы в экземпляре типа variant не хранился *ни один* экземпляр. Это значит, что тип std::variant не поддерживает возможность *опциональности*.

Экземпляр типа variant<A, B, C> имитирует объединение, которое может выглядеть так:

```
union U {
    A a;
    B b;
    C c;
};
```

Проблема с объединениями заключается в том, что нужно создавать собственные механизмы для определения того, экземпляром какого типа оно было инициализировано: A, B или C. Тип std::variant может сделать это за нас, не прилагая особых усилий.

В коде, показанном в этом разделе, мы использовали три разных способа работы с содержимым переменной variant.

Первый способ — применение функции index() типа variant. Для типа variant<A, B, C> она может вернуть индекс 0, если экземпляр был инициализирован переменной типа A, 1 для типа B или 2 для типа C, и т. д. для более сложных вариантов.

Следующий способ — использование функции get_if<T>. Она принимает адрес объекта типа variant и возвращает указатель типа T на его содержимое. Если тип T указан неправильно, то указатель станет нулевым. Кроме того, можно вызвать метод get<T>(x) для переменной типа variant, чтобы получить ссылку на ее содержимое, но если это не сработает, то данная функция сгенерирует исключение (перед выполнением таких преобразований можно проверить правильность типа с помощью булева предиката holds_alternative<T>(x)).

Последний способ получить доступ к значению, хранящемуся в типе variant, — применить функцию std::visit. Она принимает объект функции и экземпляр типа variant. Функция проверяет, какой тип имеет содержимое экземпляра типа variant, а затем вызывает соответствующий перегруженный оператор () объекта функции.

Именно для этих целей мы и реализовали тип animal_voice, поскольку он может быть использован в комбинации с visit и variant<dog, cat>:

```
struct animal_voice
{
    void operator()(const dog &d) const { d.woof(); }
    void operator()(const cat &c) const { c.meow(); }
};
```

Последний описанный способ получения доступа к экземплярам, хранящимся в экземплярах типа variant, считается самым элегантным, поскольку в разделах кода, в которых мы получаем доступ к экземпляру, не нужно жестко кодировать возможные типы. Это позволяет проще расширять код.

Утверждение о том, что тип variant не может не иметь значения, было не совсем верным. Добавив тип std::monostate в список вероятных типов, можно указать, что экземпляр не будет хранить значения.

Автоматическое управление ресурсами с помощью std::unique_ptr

Начиная с C++11 в STL появились умные указатели, помогающие отслеживать динамическую память и ее использование. Даже до C++11 существовал класс `auto_ptr`, который мог управлять динамической памятью, но его было легко применить неправильно.

Однако с появлением умных указателей теперь редко приходится самостоятельно использовать ключевые слова `new` и `delete`, и это очень хорошо. Умные указатели — отличный пример автоматического управления памятью. Поддерживая объекты, память для которых выделяется динамически с помощью `unique_ptr`, мы защищены от утечек памяти, поскольку при разрушении объекта данный класс автоматически вызывает поддерживаемый им объект.

Уникальный указатель выражает принадлежность объекта, на который ссылается, и выполняет свою задачу по освобождению его памяти, если та более не используется. Этот класс может навсегда освободить нас от утечек памяти (во всяком случае вместе со своими компаньонами `shared_ptr` и `weak_ptr`, но в этом примере мы концентрируемся только на `unique_ptr`). Самая приятная особенность заключается в том, что он *не влияет* на производительность и свободное место в сравнении с кодом, содержащим необработанные указатели и предусматривающим ручное управление памятью. (О'кей, он все еще устанавливает значение внутреннего необработанного указателя на `nullptr` после разрушения объекта, на который он указывает, и это нужно учитывать при оптимизации. Большая часть кода, управляющего динамической памятью и написанного вручную, делает то же самое.)

В этом разделе мы рассмотрим `unique_ptr` и способы его использования.

Как это делается

В этом примере мы напишем программу, которая покажет, как `unique_ptr` работает с памятью путем создания пользовательского типа, добавляющего некие отладочные сообщения при создании и разрушении объекта. Затем поработаем с уникальными указателями, управляя экземплярами этого типа, для которых память выделяется динамически.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <memory>

using namespace std;
```

2. Мы реализуем небольшой класс для объекта, которым будем управлять с помощью `unique_ptr`. Его конструктор и деструктор станут выводить сообщения на консоль; это поможет узнать о том, что объект был на самом деле удален.

```
class Foo
{
```

```
public:
    string name;

    Foo(string n)
        : name{move(n)}
    { cout << "CTOR " << name << '\n'; }

    ~Foo() { cout << "DTOR " << name << '\n'; }
};
```

3. Попробуем реализовать функцию, принимающую в качестве аргументов уникальные указатели, чтобы увидеть, какие ограничения она имеет. Она *обрабатывает* элемент типа `Foo`, выводя его название. Обратите внимание: хотя уникальные указатели являются умными, работают без лишних издержек и удобны в использовании, они все еще могут иметь значение `null`. Это значит, что их все еще нужно проверять перед разыменованием.

```
void process_item(unique_ptr<Foo> p)
{
    if (!p) { return; }
    cout << "Processing " << p->name << '\n';
}
```

4. В функции `main` откроем еще одну область видимости, создадим два объекта типа `Foo` в куче и будем управлять ими обоими с помощью уникальных указателей. Создадим первый объект в куче явно, используя оператор `new`, а затем поместим его в конструктор переменной типа `unique_ptr<Foo>`, `p1`. Создадим уникальный указатель `p2` путем вызова `make_unique<Foo>` с аргументами, которые в противном случае передали бы конструктору класса `Foo`. Этот способ более элегантен, поскольку можно воспользоваться автоматическим определением типов и при первом обращении к объекту он уже будет управляться `unique_ptr`:

```
int main()
{
    {
        unique_ptr<Foo> p1 {new Foo{"foo"}};
        auto            p2 (make_unique<Foo>("bar"));
    }
```

5. После того как мы покинем область видимости, оба объекта будут разрушены и их память вернется в кучу. Взглянем на функцию `process_item` и узнаем, как теперь ею пользоваться вкупе с `unique_ptr`. Если мы создадим новый экземпляр типа `Foo`, управляемый `unique_ptr` в вызове функции, то его время жизни будет ограничено областью видимости функции. Когда функция `process_item` отработает, объект будет уничтожен:

```
    process_item(make_unique<Foo>("foo1"));
```

6. Если мы хотим вызвать `process_item` для объекта, который существовал еще до вызова, нужно передать *право владения*, поскольку данная функция принимает

unique_ptr по значению; это значит, что его вызов создаст копию. Но unique_ptr нельзя скопировать, его можно только *переместить*. Создадим еще два объекта типа Foo и переместим один из них в process_item. Взглянув на консоль, мы увидим, что foo2 был уничтожен после того, как отработала функция process_item, поскольку мы передали ей право владения. Объект foo3 продолжит существовать до тех пор, пока не отработает функция main.

```
    auto p1 (make_unique<Foo>("foo2"));
    auto p2 (make_unique<Foo>("foo3"));
    process_item(move(p1));
    cout << "End of main()\n";
}
```

7. Скомпилируем и запустим программу. Сначала мы увидим вызовы конструктора и деструктора для foo и bar. Они разрушаются после того, как программа покидает дополнительную область видимости. Обратите внимание: объекты разрушаются в порядке, обратном тому, в котором были созданы. Следующая строка конструктора принадлежит foo1 — мы создали этот объект во время вызова process_item. Он уничтожается сразу после вызова функции. Затем создали объекты foo2 и foo3. Первый из них уничтожается сразу после вызова process_item, где мы передали право владения. Другой элемент, foo3, разрушается после выполнения последней строки кода функции main.

```
$ ./unique_ptr
CTOR foo
CTOR bar
DTOR bar
DTOR foo
CTOR foo1
Processing foo1
DTOR foo1
CTOR foo2
CTOR foo3
Processing foo2
DTOR foo2
End of main()
DTOR foo3
```

Как это работает

Управлять объектами кучи с помощью std::unique_ptr очень легко. После того как мы инициализировали уникальный указатель так, чтобы он хранил указатель на некий объект, он не может быть случайно удален в какой-то ветке кода.

Если мы присвоим какой-то новый указатель уникальному указателю, то он сначала удалит старый объект, а только затем сохранит новый указатель. Для переменной уникального указателя x можно также вызвать x.reset() только затем, чтобы удалить объект, на который он указывает, не присваивая новый указатель. Еще одна эквивалентная альтернатива повторному присваиванию с помощью x = new_pointer — это x.reset(new_pointer).

Существует единственный способ освободить объект от управления unique_ptr без удаления самого объекта. Это делает функция release, но использовать ее в большинстве ситуаций не рекомендуется.

Поскольку указатели нужно проверять перед разыменованием, они переопределяют некоторые операторы так, чтобы те походили на необработанные указатели. Условия наподобие `if (p) {...}` и `if (p != nullptr) {...}` работают так же, как если бы мы проверяли необработанный указатель.

Разыменовать уникальный указатель можно с помощью функции `get()`, возвращающей необработанный указатель на объект, или непосредственно с применением `operator*`, что опять же делает их похожими на необработанные указатели.

Одна из важных характеристик `unique_ptr` заключается в том, что его экземпляры нельзя *скопировать*, но можно *переместить* из одной переменной типа `unique_ptr` в другую. Именно поэтому нам пришлось перемещать существующий уникальный указатель в функцию `process_item`. Если бы мы могли скопировать уникальный указатель, то это значило бы, что объектом обладали сразу *два* указателя. Такое положение дел противоречит идее *уникальных* указателей, которая гласит: он может быть *единственным владельцем* объекта (а позже «удалителем»).

Поскольку существуют структуры данных, такие как unique_ptr и shared_ptr, необходимость создавать объекты в куче вручную с помощью ключевых слов new и delete возникает редко. Используйте эти классы везде, где возможно! unique_ptr не создает никаких лишних издержек во время выполнения.

Автоматическое управление разделяемой памятью кучи с использованием std::shared_ptr

В предыдущем примере мы узнали, как использовать `unique_ptr`. Это очень полезный и важный класс, поскольку помогает нам управлять объектами, память для которых выделяется динамически. Однако он может владеть объектом только *единолично*. Нельзя сделать так, чтобы *несколько* объектов данного класса обладали одним динамически выделенным объектом, поскольку будет непонятно, кто именно должен удалить его.

Тип указателя `shared_ptr` был разработан специально для этого случая. Общие указатели могут быть *скопированы* любое количество раз. Внутренний механизм подсчета ссылок отслеживает, сколько объектов все еще содержат указатель на объект. Только последний общий указатель, выходящий за пределы области видимости, может удалить объект. Таким образом, можно быть уверенными в том, что утечек памяти не возникнет, поскольку объекты удаляются автоматически после

использования, как и в том, что они не будут удаляться слишком рано или слишком часто (каждый созданный объект должен быть удален всего *один раз*).

В этом примере вы узнаете, как использовать `shared_ptr` для автоматического управления динамическими объектами, которые имеют несколько владельцев, и увидите, чем общие указатели отличаются от `unique_ptr:`.

Как это делается

В этом примере мы напишем программу, похожую на ту, которую мы писали в предыдущем примере, чтобы освоить основные принципы использования общих указателей.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <memory>

using namespace std;
```

2. Затем определим небольшой вспомогательный класс, который поможет увидеть, когда его экземпляры будут создаваться и разрушаться. Мы станем управлять его экземплярами с помощью `shared_ptr`:

```
class Foo
{
public:
    string name;
    Foo(string n)
        : name{move(n)}
    { cout << "CTOR " << name << '\n'; }
    ~Foo() { cout << "DTOR " << name << '\n'; }
};
```

3. Далее реализуем функцию, которая принимает общий указатель на экземпляр типа `Foo` *по значению*. Передача общих указателей по значению в качестве аргументов более интересна, чем их передача по ссылке, поскольку в этом случае их нужно скопировать, что изменит их внутренний счетчик ссылок, как мы увидим далее.

```
void f(shared_ptr<Foo> sp)
{
    cout << "f: use counter at "
         << sp.use_count() << '\n';
}
```

4. В функции `main` мы объявим пустой общий указатель. Вызвав его конструктор по умолчанию, мы сделаем его указателем `null`:

```
int main()
{
    shared_ptr<Foo> fa;
```

5. Далее откроем еще одну область видимости и создадим два объекта типа `Foo`. Первый из них создадим с помощью оператора `new` и передадим его в конструктор типа `shared_ptr`. Далее создадим второй экземпляр, используя `make_shared<Foo>`, что позволяет создать экземпляр типа `Foo` на основе переданных нами параметров. Этот метод более элегантен, поскольку можно воспользоваться автоматическим выведением типов, и объект уже будет управляемым, когда у нас появится первая возможность получить к нему доступ. Это очень похоже на тот код, который мы писали в примере для `unique_ptr`.

```
{
    cout << "Inner scope begin\n";
    shared_ptr<Foo> f1 {new Foo{"foo"}};
    auto            f2 (make_shared<Foo>("bar"));
```

6. Поскольку общие указатели могут быть разделяемыми, они должны отслеживать количество сторон, владеющих ими. Это делается с помощью внутреннего счетчика ссылок или счетчика *использования*. Можно вывести на экран его значение, задействуя `use_count`. Сейчас его значение равно `1`, поскольку мы еще не копировали его. Копирование `f1` в `fa` увеличит значение счетчика использования до `2`.

```
    cout << "f1's use counter at " << f1.use_count() << '\n';
    fa = f1;
    cout << "f1's use counter at " << f1.use_count() << '\n';
```

7. Когда мы покинем область видимости, общие указатели `f1` и `f2` будут уничтожены. Счетчик ссылок переменной `f1` снова уменьшится до `1`, что сделает `fa` единственным владельцем экземпляра типа `Foo`. При разрушении `f2` его счетчик ссылок будет уменьшен до `0`. В данном случае деструктор класса `shared_ptr` выполнит операцию `delete` для этого объекта, который удалит его.

```
}
cout << "Back to outer scope\n";
cout << fa.use_count() << '\n';
```

8. Теперь вызовем функцию `f` для нашего общего указателя двумя разными способами. Сначала вызовем ее путем копирования `fa`. Функция `f` затем выведет на экран значение счетчика ссылок, которое равно `2`. Во втором вызове `f` переместим указатель в функцию. Это сделает `f` единственным владельцем объекта.

```
cout << "first f() call\n";
f(fa);
cout << "second f() call\n";
f(move(fa));
```

9. После того как функция `f` отработает, экземпляр `Foo` будет мгновенно уничтожен, поскольку мы им больше не владеем. Поэтому все объекты подвергнутся уничтожению, когда отработает функция `main`.

```
    cout << "end of main()\n";
}
```

10. Компиляция и запуск программы дадут следующий результат. Сначала мы увидим, что созданы "foo" и "bar". После копирования f1 (указывает на "foo") его счетчик ссылок увеличился до значения 2. При выходе из области видимости "bar" уничтожается, поскольку общий указатель был его единственным владельцем. Одна единица на экране — счетчик ссылок fa, который является единственным владельцем "foo". После этого мы дважды вызываем функцию f. При первом вызове скопировали ее в fa, что снова увеличило его счетчик ссылок до 2. При втором переместили его в f, это не изменило его счетчик ссылок. Более того, поскольку функция f к этому моменту является единственным владельцем "foo", объект мгновенно разрушается после того, как f покидает область видимости. Таким образом, другие объекты кучи не разрушаются после последнего выражения print в функции main.

```
$ ./shared_ptr
Inner scope begin
CTOR foo
CTOR bar
f1's use counter at 1
f1's use counter at 2
DTOR bar
Back to outer scope
1
first f() call
f: use counter at 2
second f() call
f: use counter at 1
DTOR foo
end of main()
```

Как это работает

При создании и удалении объектов shared_ptr работает аналогично unique_ptr. Создание общих указателей выглядит так же, как и создание уникальных указателей (однако существует функция make_shared, которая создает общие объекты в дополнение к функции make_unique для уникальных указателей unique_ptr).

Основное отличие от unique_ptr заключается в том, что можно копировать экземпляры shared_ptr, поскольку общие указатели поддерживают так называемый *блок управления* вместе с объектом, которым они управляют. Блок управления содержит указатель на объект и счетчик ссылок или счетчик *использования*. Если на объект указывают N экземпляров shared_ptr, то счетчик использования имеет значение N. Когда экземпляр типа shared_ptr разрушается, его деструктор уменьшает значение этого внутреннего счетчика использования. Последний общий указатель на такой объект при разрушении снизит значение счетчика использования до 0. В данном случае будет вызван оператор delete для объекта! Таким образом, мы не можем допустить утечку памяти, поскольку счетчик ссылок объекта отслеживается автоматически.

Чтобы проиллюстрировать эту идею, взглянем на рис. 8.2.

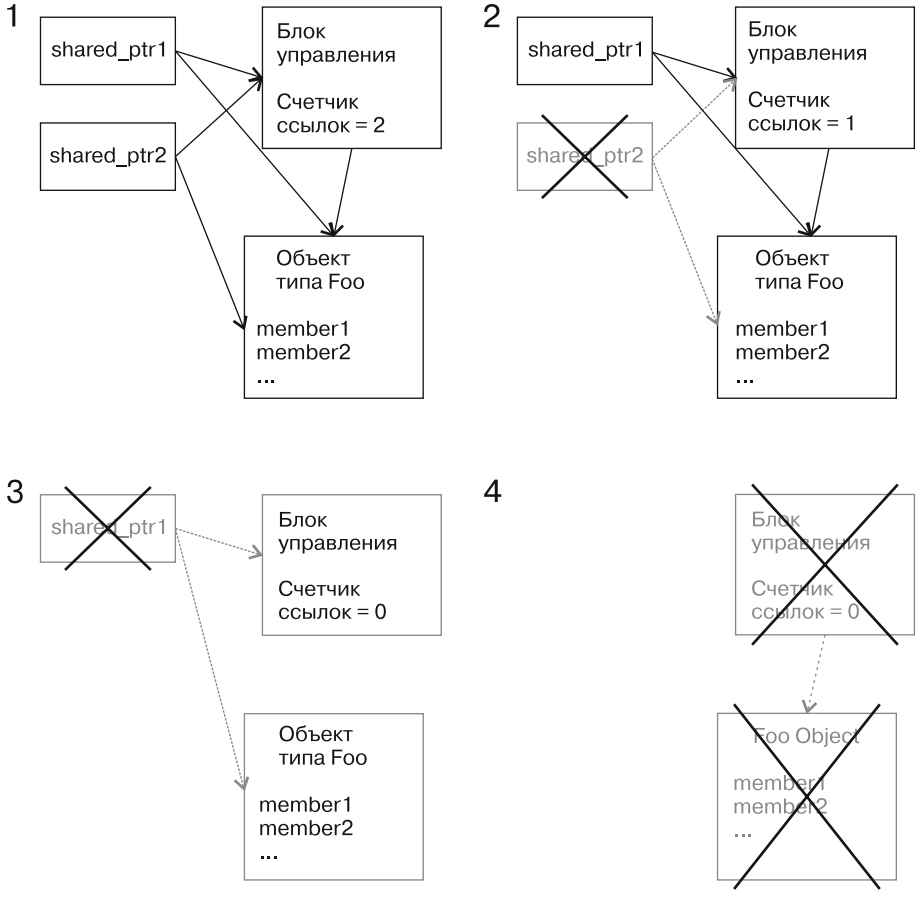

Рис. 8.2

В шаге 1 у нас имеются два экземпляра типа `shared_ptr`, управляющих объектом типа `Foo`. Значение счетчика использования установлено на `2`. Далее `shared_ptr2` уничтожается, это снижает значение счетчика использования до `1`. Экземпляр `Foo` пока не уничтожается, поскольку все еще существует второй общий указатель. В шаге 3 последний общий указатель также уничтожается. Это приводит к тому, что значение счетчика использования становится равным `0`. Шаг 4 выполняется сразу после шага 3. Блок управления и экземпляр типа `Foo` уничтожаются, и занятая ими память возвращается в кучу.

С помощью `shared_ptr` и `unique_ptr` можно автоматически справиться с большинством объектов, память для которых выделяется динамически, не беспокоясь об утечках памяти. Следует, однако, рассмотреть один важный подводный камень. Представьте, что у нас имеются два объекта в куче, которые содержат

общие указатели друг на друга, и какой-то другой общий указатель, указывающий на один из них откуда-то еще. Если этот внешний указатель выйдет за пределы области видимости, то счетчики использования обоих объектов все еще будут иметь *ненулевые* значения, поскольку ссылаются *друг на друга*. Это приводит к *утечке памяти*. В подобной ситуации общие указатели применять нельзя, поскольку цепочки таких циклических ссылок не дают снизить значение счетчика использования до `0`.

Дополнительная информация

Рассмотрим следующий код. Допустим, вам сказали, что он провоцирует потенциальную *утечку памяти*.

```
void function(shared_ptr<A>, shared_ptr<B>, int);
// "function" определена где-то еще

// ...далее по коду:
function(new A{}, new B{}, other_function());
```

Кто-то спросит «Где же утечка?» — ведь объекты `A` и `B`, для которых только что выделена память, мгновенно передаются в экземпляры типа `shared_ptr`, и *это* позволяет обезопасить нас от утечек.

Да, это так, утечки памяти нам не грозят до тех пор, пока указатели хранятся в экземплярах типа `shared_ptr`. Эту проблему решить довольно сложно.

Когда мы вызываем функцию `f(x(), y(), z())`, компилятор должен собрать код, который сначала вызывает функции `x()`, `y()` и `z()`, чтобы он мог перенаправить результат их работы в функцию `f`. Нас не устраивает то, что компилятор может выполнить вызовы функций `x`, `y` и `z` в *любом* порядке.

Взглянем на пример еще раз и подумаем: какие события произойдут, если компилятор решит структурировать код так, что сначала будет вызвана функция `new A{}`, затем — `other_function()`, а затем — `B{}`, прежде чем результаты работы этих функций будут переданы далее? Если функция `other_function()` сгенерирует исключение, то мы получим утечку памяти, поскольку у нас в куче все еще будет неуправляемый объект `A`, поскольку мы не имели возможности передать его под управление `shared_ptr`. Независимо от того, как мы обработаем исключение, дескриптор объекта *пропадет*, и мы *не сможем удалить* его!

Существует два простых способа обойти эту проблему:

```
// 1.)
function(make_shared<A>(), make_shared<B>(), other_function());

// 2.)
shared_ptr<A> ap {new A{}};
shared_ptr<B> bp {new B{}};
function(ap, bp, other_function());
```

Таким образом, объекты уже попадут под управление `shared_ptr` независимо от того, где позднее будет сгенерировано исключение.

Работаем со слабыми указателями на разделяемые объекты

Из примера, посвященного `shared_ptr`, мы узнали, какими полезными и простыми в использовании являются общие указатели. Вместе с `unique_ptr` они предоставляют бесценную возможность по улучшению нашего кода, нуждающегося в управлении объектами, память для которых выделяется динамически.

Копируя `shared_ptr`, мы увеличиваем его внутренний счетчик ссылок. До тех пор пока мы храним нашу копию общего указателя, объект, на который он указывает, не будет удален. Но если нужно что-то вроде *слабого* указателя, который позволит получать объект до тех пор, пока тот существует, но не мешает его удалению? Как мы определим, существует ли еще объект?

В таких ситуациях нам поможет `weak_ptr`. Использовать его чуть сложнее, чем `unique_ptr` и `shared_ptr`, но после прочтения этого раздела вы научитесь применять его.

Как это делается

В этом примере мы реализуем программу, которая поддерживает объекты, используя экземпляры типа `shared_ptr`, а затем добавим `weak_ptr`, чтобы увидеть, как это меняет поведение при управлении памятью с помощью умного указателя.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <iomanip>
#include <memory>

using namespace std;
```

2. Затем реализуем класс, чей деструктор выводит на экран сообщение. Таким образом, нам будет проще проверить факт уничтожения объекта.

```
struct Foo {
    int value;
    Foo(int i) : value{i} {}
    ~Foo() { cout << "DTOR Foo " << value << '\n'; }
};
```

3. Также реализуем функцию, которая выводит на экран информацию о слабом указателе, что позволит узнавать о его состоянии в разные моменты выполнения программы. Функция `expired` класса `weak_ptr` скажет о том, существует ли еще объект, на который он указывает, поскольку хранение слабого указателя на объект не продлевает его время жизни! Счетчик `use_count`

сообщает, сколько экземпляров типа `shared_ptr` в данный момент указывают на наш объект:

```
void weak_ptr_info(const weak_ptr<Foo> &p)
{
    cout << "---------" << boolalpha
         << "\nexpired:   " << p.expired()
         << "\nuse_count: " << p.use_count()
         << "\ncontent:   ";
```

4. При желании получить доступ к самому объекту нужно вызвать функцию `lock`. Она возвращает общий указатель на объект. Если объект больше *не существует*, то полученный общий указатель, по сути, является `null`. Следует это проверять, прежде чем получать доступ к объекту.

```
    if (const auto sp (p.lock()); sp) {
        cout << sp->value << '\n';
    } else {
        cout << "<null>\n";
    }
}
```

5. Создадим пустой слабый указатель в функции `main` и выведем на экран его содержимое, оно, конечно, поначалу будет пустым:

```
int main()
{
    weak_ptr<Foo> weak_foo;
    weak_ptr_info(weak_foo);
```

6. В новой области видимости создадим новый общий указатель, содержащий только что созданный экземпляр класса `Foo`. Затем скопируем его в слабый указатель. Обратите внимание: это не увеличит счетчик ссылок общего указателя. Счетчик ссылок будет иметь значение `1`, поскольку им владеет только один *общий* указатель.

```
    {
        auto shared_foo (make_shared<Foo>(1337));
        weak_foo = shared_foo;
```

7. Вызовем функцию слабого указателя прежде, чем *покинем* область видимости, и снова *после* этого. Экземпляр типа `Foo` должен быть мгновенно уничтожен, *несмотря* на то что на него указывает слабый указатель.

```
        weak_ptr_info(weak_foo);
    }
    weak_ptr_info(weak_foo);
}
```

8. Компиляция и запуск программы дадут три результата работы функции `weak_ptr_info`. В первом вызове слабый указатель пуст. Во втором он уже указывает

на созданный нами экземпляр типа `Foo` и может разыменовать его после *блокировки*. Перед третьим вызовом мы покидаем внутреннюю область видимости, что заставляет сработать деструктор экземпляра типа `Foo` в соответствии с нашими ожиданиями. После этого мы не можем получить содержимое экземпляра типа `Foo` с помощью слабого указателя, а сам слабый указатель корректно распознает, что срок его действия истек.

```
$ ./weak_ptr
---------
expired:   true
use_count: 0
content:   <null>
---------
expired:   false
use_count: 1
content:   1337
DTOR Foo 1337
---------
expired:   true
use_count: 0
content:   <null>
```

Как это работает

Слабые указатели предоставляют способ указать на объект, поддерживаемый общими указателями, не увеличивая его счетчик использования. Да, необработанный указатель способен сделать то же самое, но не может сказать, является ли он висящим. Слабый указатель лишен этого недостатка!

Чтобы понять, как слабые указатели работают с общими, сразу рассмотрим рис. 8.3.

Принцип работы аналогичен тому, что приведен на рис. 8.2. В шаге 1 у нас имеются два общих указателя и слабый, указывающие на объект типа `Foo`. Несмотря на то что на него указывают три объекта, его счетчик использования изменяют только общие указатели, именно поэтому его значение равно `2`. Слабый указатель изменяет только *слабый счетчик* блока управления. В шагах 2 и 3 экземпляры общих указателей уничтожаются, это снижает значение счетчика использования до `0`. В шаге 4 это приводит к тому, что объект `Foo` удаляется, но блок управления *остается*. Слабому указателю все еще нужен блок управления, чтобы определить, является ли он висящим. Блок управления удаляется только в тот момент, когда *последний слабый* указатель, указывающий на него, *тоже* выходит из области видимости.

Мы также можем сказать, что срок действия висящего слабого указателя *истек*. Для проверки этого состояния можно опросить метод `expired` класса `weak_ptr`, он вернет булево значение. Если оно равно `true`, то мы не можем разыменовать слабый указатель, поскольку он не указывает на объект.

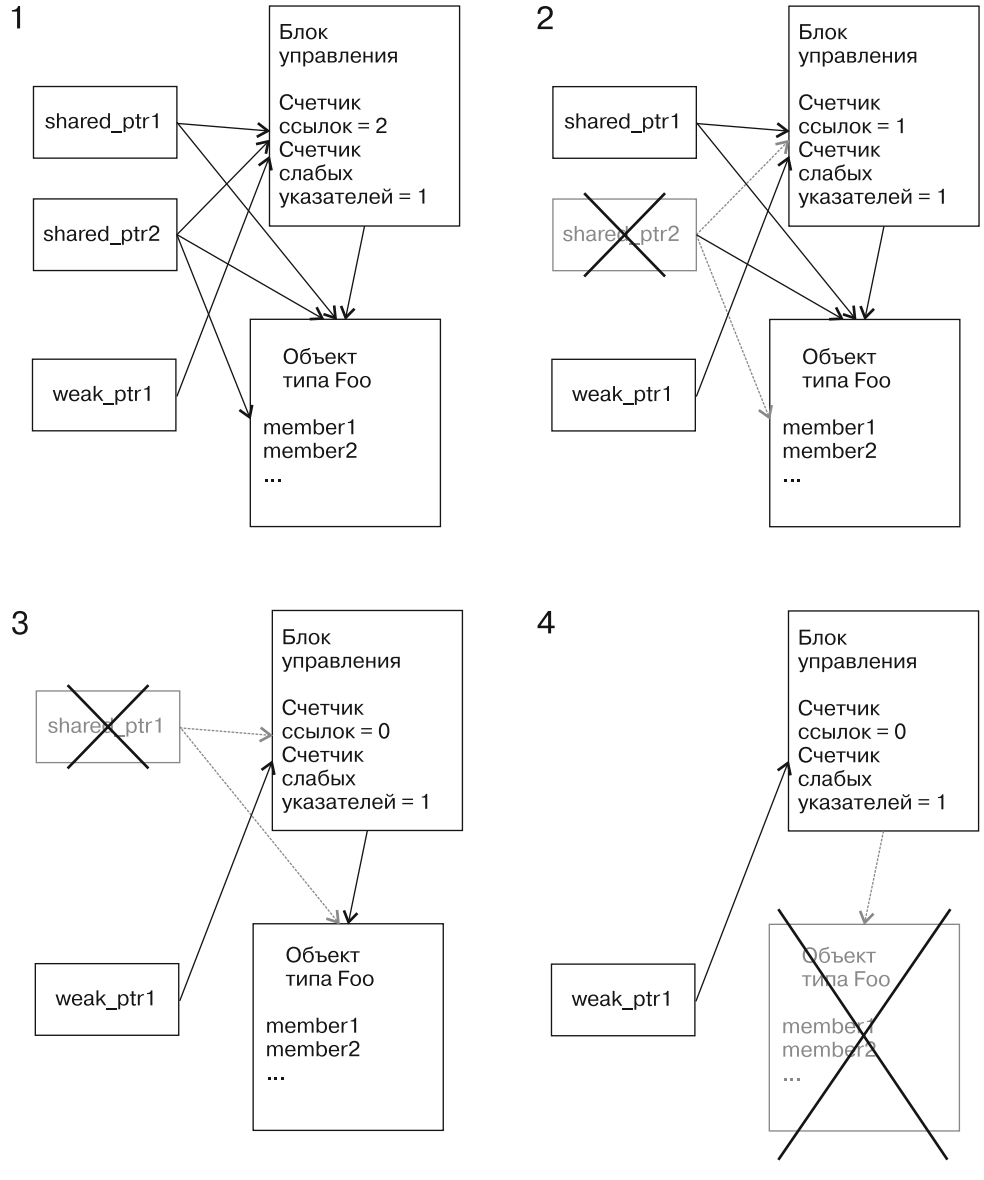

Рис. 8.3

Чтобы разыменовать слабый указатель, нужно вызвать метод `lock()`. Это удобно и безопасно, поскольку данная функция возвращает общий указатель. Пока мы его храним, объект, на который он указывает, не может пропасть, поскольку мы увеличили его счетчик использования путем блокировки. Если объект удаляется вскоре после вызова `lock()`, то общий указатель, по сути, является `null`.

Упрощаем управление ресурсами устаревших API с применением умных указателей

Умные указатели (`unique_ptr`, `shared_ptr` и `weak_ptr`) очень полезны, и можно сказать, что программист должен *всегда* использовать их вместо выделения и освобождения памяти вручную.

Но если для объекта нельзя выделить память с помощью оператора `new` и/или освободить память, задействуя оператор `delete`? Во многих устаревших библиотеках есть собственные функции выделения/удаления памяти. Кажется, это может стать проблемой, поскольку мы узнали, что умные указатели полагаются на операторы `new` и `delete`. Если создание и/или разрушение конкретных типов объектов полагается на конкретные интерфейсы удаления фабричных функций, не помешает ли это получить огромное преимущество от использования умных указателей?

Вовсе нет. В этом разделе мы увидим, что нужно лишь немного изменить умные указатели, чтобы позволить им следовать определенным процедурам выделения и удаления памяти для конкретных объектов.

Как это делается

В данном примере мы определим тип, для которого нельзя непосредственно выделить память с помощью оператора `new` и нельзя освободить ее, прибегнув к оператору `delete`. Поскольку это помешает использовать его вместе с умными указателями, мы внесем небольшие изменения в экземпляры классов `unique_ptr` и `smart_ptr`.

1. Как и всегда, сначала включим необходимые заголовочные файлы и объявим об использовании пространства имен `std` по умолчанию:

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

using namespace std;
```

2. Далее объявим класс, конструктор и деструктор которого имеют модификатор `private`. Таким образом, симулируем проблему, когда нужно получать доступ к конкретным функциям, чтобы создавать и удалять экземпляры этого класса.

```
class Foo
{
    string name;
```

```
    Foo(string n)
        : name{n}
    { cout << "CTOR " << name << '\n'; }
    ~Foo() { cout << "DTOR " << name << '\n';}
```

3. Статические методы `create_foo` и `destroy_foo` будут создавать и удалять экземпляры типа `Foo`. Они работают с необработанными указателями. Это симулирует ситуацию, возникающую при использовании устаревшего API языка C, который не дает задействовать их непосредственно для обычных указателей `shared_ptr`:

```
public:
    static Foo* create_foo(string s) {
       return new Foo{move(s)};
    }

    static void destroy_foo(Foo *p) { delete p; }
};
```

4. Теперь сделаем так, чтобы подобными объектами можно было управлять с помощью `shared_ptr`. Конечно, можно помещать указатели, которые получаем из функции `create_foo`, в конструктор общего указателя. Разрушение объекта выглядит сложнее, поскольку функция удаления класса `shared_ptr`, использующаяся по умолчанию, решит проблему неправильно. Идея заключается в том, что можно задать для класса `shared_ptr` *пользовательскую функцию удаления*. Сигнатура функции, которую следует иметь функции удаления или вызываемому объекту, должна совпадать с сигнатурой функции `destroy_foo`. Если функция, которую нужно вызвать для разрушения объекта, более сложна, то можно обернуть ее в лямбда-выражение.

```
static shared_ptr<Foo> make_shared_foo(string s)
{
    return {Foo::create_foo(move(s)), Foo::destroy_foo};
}
```

5. Обратите внимание: `make_shared_foo` возвращает обычный экземпляр `shared_ptr<Foo>`, поскольку передача пользовательской функции удаления не изменяет ее тип. Это произошло потому, что `shared_ptr` применяет вызовы виртуальных функций для сокрытия таких деталей. Уникальные указатели не создают налишних издержек; это не дает задействовать для них подобный прием. Нужно изменить тип `unique_ptr`. В качестве второго шаблонного параметра мы передадим экземпляр типа `void (*)(Foo*)`, данный тип имеет и указатель на функцию `destroy_foo`:

```
static unique_ptr<Foo, void (*)(Foo*)> make_unique_foo(string s)
{
    return {Foo::create_foo(move(s)), Foo::destroy_foo};
}
```

6. В функции `main` просто создаем экземпляры общего и уникального указателей. В выходных данных программы увидим, будут ли они уничтожаться корректно и автоматически.

```
int main()
{
    auto ps (make_shared_foo("shared Foo instance"));
    auto pu (make_unique_foo("unique Foo instance"));
}
```

7. Компиляция и запуск программы дадут ожидаемый результат:

```
$ ./legacy_shared_ptr
CTOR shared Foo instance
CTOR unique Foo instance
DTOR unique Foo instance
DTOR shared Foo instance
```

Как это работает

Обычно `unique_ptr` и `shared_ptr` просто вызывают оператор `delete` для внутренних указателей, когда должны уничтожить объект, который сопровождают. В этом разделе мы создали класс, для которого нельзя выделить память, используя `x = new Foo{123}`, и разрушить объект непосредственно с помощью `delete x`.

Функция `Foo::create_foo` просто возвращает необработанный указатель на только что созданный экземпляр типа `Foo`, и это не вызывает других проблем, поскольку умные указатели работают с необработанными.

Сложность заключается в том, что нужно научить классы `unique_ptr` и `shared_ptr` *разрушать* объект, если способ по умолчанию *не подходит*.

С этой точки зрения оба типа умных указателей несколько отличаются друг от друга. Чтобы определить пользовательскую функцию удаления для `unique_ptr`, нужно изменить его тип. Поскольку тип сигнатуры `delete` класса `Foo` — `void Foo::destroy_foo(Foo*);`, типом уникального указателя, сопровождающего экземпляр типа `Foo`, должен быть `unique_ptr<Foo, void (*)(Foo*)>`. Теперь он может хранить указатель на функцию `destroy_foo`, которую мы предоставляем в качестве второго параметра конструктора в нашей функции `make_unique_foo`.

Если передача пользовательской функции удаления для класса `unique_ptr` заставила нас сменить его тип, то почему же мы смогли сделать то же самое для `shared_ptr`, *не изменяя* его тип? Единственное, что нам пришлось сделать, — передать второй параметр для конструктора `shared_ptr`. Почему это не может быть так же просто и для типа `unique_ptr`?

Почему так просто передать экземпляру класса `shared_ptr` некоторый вызываемый объект `delete`, не изменяя типа общего указателя? Причина кроется в природе общих указателей, поддерживающих блок управления. Блок управления общих указателей — объект, имеющий виртуальные функции. Это значит, что блок управления обычного общего указателя и блок управления общего ука-

зателя с пользовательским `delete` различаются! Чтобы с помощью уникального указателя применить пользовательскую функцию удаления, нужно изменить тип этого указателя. Если мы хотим, чтобы общий указатель задействовал пользовательскую функцию удаления, то это также изменит тип внутреннего *блока управления*, невидимого для нас, поскольку данная разница скрыта за интерфейсом виртуальной функции.

Описанный прием можно применить и для уникальных указателей, но в таком случае он повлечет некоторые издержки во время выполнения программы. Это не то, что мы хотим, поскольку уникальные указатели не должны создавать лишних издержек.

Открываем доступ к разным переменным — членам одного объекта

Представим, что у нас есть общий указатель на некий сложный объект, память для которого выделяется динамически. Нужно создать новый поток, выполняющий какую-то продолжительную работу для одного из членов этого сложного объекта. Если мы хотим освободить этот общий указатель сейчас, то объект будет удален, хотя другие потоки все еще могут пытаться получить к нему доступ. Если же мы не хотим давать объекту потока указатель на весь объект, поскольку данное действие пересечется с нашим «аккуратным» интерфейсом или по каким-то другим причинам, то значит ли это, что придется управлять памятью вручную?

Нет. Вы можете использовать общие указатели, которые, с одной стороны, ссылаются на член крупного общего объекта, а с другой — выполняют автоматическое управление памятью для всего исходного объекта.

В данном разделе мы создадим подобный сценарий (без потоков, чтобы не усложнять задачу) с целью ознакомиться с этой удобной функцией типа `shared_ptr`.

Как это делается

В этом примере мы определим структуру, которая состоит из нескольких членов. Далее выделим память для экземпляра структуры в куче, ее будет сопровождать общий указатель. Из него мы получим больше общих указателей, указывающих не на сам объект, а на его члены.

1. Сначала включим необходимые заголовочные файлы, а затем объявим об использовании пространства имен `std` по умолчанию:

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

using namespace std;
```

2. Далее определим класс, который имеет разные члены. Позволим общим указателям указывать на отдельные члены. Чтобы увидеть, когда класс создается и уничтожается, будет выводить сообщения на экран в конструкторе и деструкторе.

```
struct person {
    string name;
    size_t age;
    person(string n, size_t a)
        : name{move(n)}, age{a}
    { cout << "CTOR " << name << '\n'; }
    ~person() { cout << "DTOR " << name << '\n'; }
};
```

3. Определим общие указатели, которые имеют корректные типы, чтобы указывать на переменные-члены `name` и `age` экземпляра класса `person`:

```
int main()
{
    shared_ptr<string> shared_name;
    shared_ptr<size_t> shared_age;
```

4. Далее войдем в новую область видимости, создадим объект типа `person` и позволим общему указателю управлять им:

```
    {
        auto sperson (make_shared<person>("John Doe", 30));
```

5. Затем позволим первым двум общим указателям указывать на его члены `name` и `age`. Прием, который мы задействуем, заключается в использовании конкретного конструктора типа `shared_ptr`, который принимает общий указатель и указатель на член общего объекта. Таким образом можно управлять объектом, не указывая на него самого!

```
        shared_name = shared_ptr<string>(sperson, &sperson->name);
        shared_age  = shared_ptr<size_t>(sperson, &sperson->age);
    }
```

6. После выхода из области видимости выведем на экран значения переменных `name` и `age`. Это возможно только в том случае, если память для объекта все еще выделена.

```
    cout << "name: "  << *shared_name
         << "\nage: " << *shared_age << '\n';
}
```

7. Компиляция и запуск программы дадут следующий результат. Из сообщения деструктора мы видим, что объект все еще жив и память для него выделена, когда мы получаем доступ к значениям переменных `name` и `age` с помощью указателей на члены!

```
$ ./shared_members
CTOR John Doe
name: John Doe
age:  30
DTOR John Doe
```

Как это работает

В этом разделе мы сначала создали общий указатель, управляющий объектом `person`, память для которого выделяется динамически. Затем создали два других умных указателя, указывающих на объект типа `person`, но *не на сам* объект, а на его члены `name` и `age`.

Чтобы подытожить созданный сценарий, взглянем на рис. 8.4.

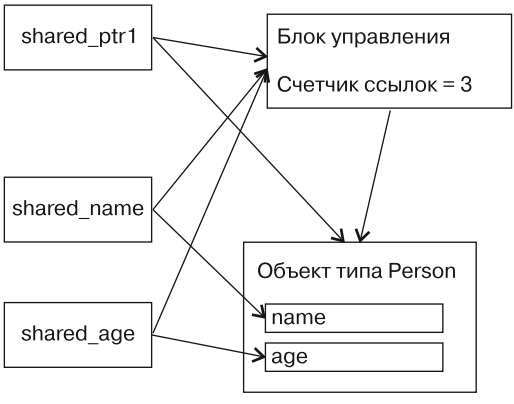

Рис. 8.4

Обратите внимание: `shared_ptr1` указывает на сам объект `person`, а `shared_name` и `shared_age` — на члены `name` и `age` того же объекта. По всей видимости, они будут управлять всем жизненным циклом объекта. Это возможно потому, что указатели внутреннего блока управления все еще ссылаются на тот же блок управления, независимо от того, на какой подобъект указывают отдельные общие указатели.

В этом сценарии счетчик использования блока управления равен `3`. Таким образом, объект типа `person` не будет удален при уничтожении `shared_ptr1`, поскольку другие общие указатели все еще владеют объектом.

При создании подобных экземпляров общих указателей, указывающих на члены общего объекта, синтаксис выглядит несколько странно. Чтобы получить экземпляр типа `shared_ptr<string>`, который указывает на член `name` общего экземпляра типа `person`, нужно написать следующий код:

```
auto sperson (make_shared<person>("John Doe", 30));
auto sname   (shared_ptr<string>(sperson, &sperson->name));
```

Чтобы получить указатель на конкретный член общего объекта, мы создаем экземпляр общего указателя, чей тип специализирован для того члена, к которому нужно получить доступ. Именно поэтому используем конструкцию `shared_ptr<string>`. Затем в конструкторе сначала предоставляем оригинальный общий указатель, сопровождающий объект типа `person`, а в качестве второго аргумента — адрес объекта, которым будет пользоваться новый общий указатель при разыменовании.

Генерируем случайные числа и выбираем правильный генератор случайных чисел

Чтобы получить случайные числа, программисты C++ до появления C++11 обычно просто использовали функцию `rand()` из библиотеки C. Начиная с C++11, в нашем распоряжении целый *арсенал* генераторов случайных чисел, они служат для разных целей и имеют различные характеристики.

Эти генераторы не говорят сами за себя, так что в данном разделе рассмотрим их все. Мы увидим, чем они отличаются, научимся выбирать правильный и узнаем, что, скорее всего, никогда не будем ими пользоваться.

Как это делается

В этом примере мы реализуем процедуру, которая выводит на экран гистограмму, содержащую числа, создаваемые генератором случайных чисел. Затем запустим все генераторы случайных чисел, доступные в STL, и воспользуемся нашей процедурой, чтобы исследовать результаты. Данная программа содержит множество повторяющихся фрагментов, поэтому может быть полезным просто скопировать исходный код из репозитория, дополняющего эту книгу, а не писать повторяющийся код вручную.

1. Сначала включим все заголовочные файлы, а затем объявим об использовании пространства имен `std` по умолчанию:

```
#include <iostream>
#include <string>
#include <vector>
#include <random>
#include <iomanip>
#include <limits>
#include <cstdlib>
#include <algorithm>

using namespace std;
```

2. Затем реализуем вспомогательную функцию, которая позволит сопровождать и выводить на экран статистику для каждого типа генераторов случайных

чисел. Она принимает два параметра — количество *сегментов* и количество *образцов*. Мы сразу увидим, для чего они нужны. Тип генератора случайных чисел определяется с помощью шаблонного параметра `RD`. Первое, что мы сделаем в этой функции, — определим псевдоним типа для полученного численного типа чисел, возвращаемых генератором. Кроме того, убедимся, что у нас есть как минимум десять сегментов:

```
template <typename RD>
void histogram(size_t partitions, size_t samples)
{
    using rand_t = typename RD::result_type;
    partitions = max<size_t>(partitions, 10);
```

3. Далее создадим экземпляр генератора типа `RD`. Затем определим переменную-делитель с именем `div`. Все генераторы случайных чисел создают случайные числа в диапазоне от `0` до `RD::max()`. Аргумент функции `partitions` позволяет вызывающей стороне выбирать, на сколько сегментов мы разделим каждый диапазон случайных чисел. Разделив наибольшее возможное значение на количество сегментов, мы узнаем, насколько большим является каждый из них:

```
    RD rd;
    rand_t div ((double(RD::max()) + 1) / partitions);
```

4. Создадим вектор переменных-счетчиков. Он будет иметь размер, равный количеству сегментов. Затем получим случайные значения от генератора в количестве, равном значению переменной `samples`. Выражение `rd()` получает случайное число от генератора и изменяет его внутреннее состояние так, чтобы подготовить его к выдаче следующего случайного числа. Разделив каждое случайное число на `div`, мы получим номер сегмента, в который оно попадает, и можем увеличить соответствующий счетчик в векторе:

```
    vector<size_t> v (partitions);
    for (size_t i {0}; i < samples; ++i) {
        ++v[rd() / div];
    }
```

5. Теперь у нас есть «аккуратная» гистограмма, содержащая значения-примеры. Чтобы вывести ее на экран, нужно получить более подробную информацию о ее реальных значениях счетчика. Извлечем самое большое значение с помощью алгоритма `max_element`. Затем разделим это значение на `100`. Таким образом можно разделить все значения счетчика на `max_div` и вывести множество звездочек на консоль, не выходя за значение ширины, равное `100`. Если самое крупное значение меньше `100` (это может произойти в случае применения небольшого количества образцов), то воспользуемся `max` для получения минимального делителя `1`:

```
    rand_t max_elm (*max_element(begin(v), end(v)));
    rand_t max_div (max(max_elm / 100, rand_t(1)));
```

6. Теперь выведем гистограмму на консоль. Каждый сегмент получает собственную строку на консоли. Разделив его значение счетчика на `max_div` и выведя соответствующее количество символов `'*'`, получаем строки гистограммы, которые помещаются в окно консоли:

```
    for (size_t i {0}; i < partitions; ++i) {
        cout << setw(2) << i << ": "
             << string(v[i] / max_div, '*') << '\n';
    }
}
```

7. О'кей, на этом все. Теперь перейдем к основной программе. Позволим пользователю определить, сколько сегментов и образцов следует применить:

```
int main(int argc, char **argv)
{
    if (argc != 3) {
        cout << "Usage: " << argv[0]
             << " <partitions> <samples>\n";
        return 1;
    }
```

8. Затем считаем эти переменные из командной строки. Конечно, она содержит строки, которые можно преобразовать в числа с помощью функции `std::stoull` (`stoull` — это аббревиатура для **s**tring **to** **u**nsigned **l**ong **l**ong, строки к беззнаковым значениям типа `long long`):

```
    size_t partitions {stoull(argv[1])};
    size_t samples    {stoull(argv[2])};
```

9. Теперь вызовем нашу вспомогательную функцию, создающую гистограммы, для *каждого* генератора случайных чисел, предоставляемого STL. Это сделает наш пример длинным и повторяющим код. Лучше скопируйте пример из Интернета. Интересно взглянуть на результат работы данной программы. Начнем с `random_device`. Это устройство пытается распределить случайность поровну между всеми возможными значениями:

```
    cout << "random_device" << '\n';
    histogram<random_device>(partitions, samples);
```

10. Следующий генератор случайных чисел — это `default_random_engine`. Тип генератора, на который ссылается данный тип, зависит от конкретной реализации. Он может оказаться *одним* из следующих генераторов:

```
    cout << "ndefault_random_engine" << '\n';
    histogram<default_random_engine>(partitions, samples);
```

11. Затем опробуем ее для всех других генераторов:

```
    cout << "nminstd_rand0" << '\n';
    histogram<minstd_rand0>(partitions, samples);
```

```
    cout << "nminstd_rand" << '\n';
    histogram<minstd_rand>(partitions, samples);
    cout << "nmt19937" << '\n';
    histogram<mt19937>(partitions, samples);
    cout << "nmt19937_64" << '\n';
    histogram<mt19937_64>(partitions, samples);
    cout << "nranlux24_base" << '\n';
    histogram<ranlux24_base>(partitions, samples);
    cout << "nranlux48_base" << '\n';
    histogram<ranlux48_base>(partitions, samples);
    cout << "nranlux24" << '\n';
    histogram<ranlux24>(partitions, samples);
    cout << "nranlux48" << '\n';
    histogram<ranlux48>(partitions, samples);
    cout << "nknuth_b" << '\n';
    histogram<knuth_b>(partitions, samples);
}
```

12. Компиляция и запуск программы дадут интересные результаты. Мы увидим длинный список, содержащий выходные данные, и узнаем, что все генераторы случайных чисел имеют разные характеристики. Сначала запустим программу, в которой количество сегментов равно `10`, а образцов всего `1000` (рис. 8.5).

Рис. 8.5

13. Затем запустим эту же программу снова. В этот раз количество сегментов все еще будет равно `10`, но образцов уже `1,000,000`. Станет очевидно, что гистограммы выглядят гораздо лучше, когда мы берем для них больше образцов (рис. 8.6). Это важное наблюдение.

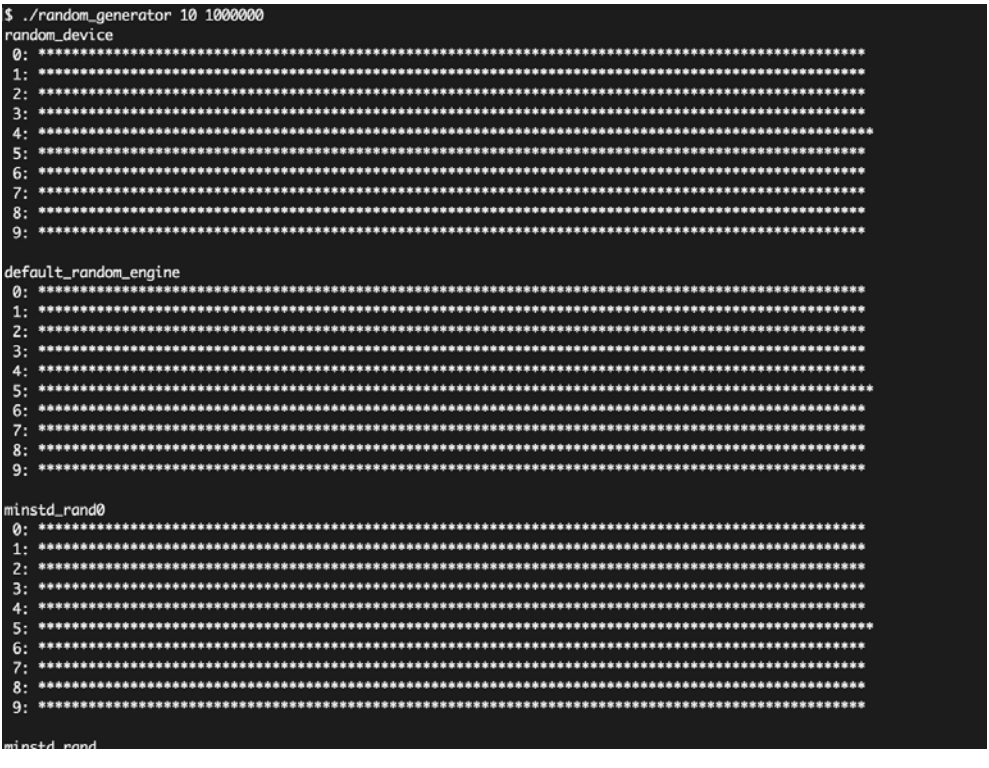

Рис. 8.6

Как это работает

Как правило, перед использованием генератора случайных чисел нужно создать объект этого класса. Полученный объект может быть вызван как функция без параметров, поскольку он перегружает `operator()`. Тогда каждый вызов будет приводить к получению нового случайного числа. Все очень просто.

В этом разделе мы написали довольно сложную программу, чтобы чуть больше узнать о генераторах случайных чисел. Пожалуйста, поработайте с получившейся программой, запуская ее с разными аргументами командной строки, и проверьте следующее:

- чем больше образцов мы возьмем, тем больше будут равны наши счетчики разделов;

- неравенство счетчиков разделов значительно отличается между отдельными генераторами;
- для большого числа образцов становится понятно, что *производительность* отдельных генераторов различается;
- запустите программу с небольшим количеством образцов несколько раз. Шаблоны распределения будут выглядеть *одинаково*: генераторы постоянно создают *одни и те же* последовательности; это говорит о том, что они *совсем не случайны*. Такие генераторы называются *детерминированными*, поскольку можно предсказать получаемые значения. Единственным исключением является `std::random_device`.

Как видите, необходимо рассмотреть несколько характеристик. Для большинства стандартных приложений достаточно применить `std::default_random_engine`. Эксперты в области криптографии или в других областях, чувствительных к безопасности, будут тщательно выбирать один из доступных генераторов, но для нас, типичных программистов, это не так важно.

Из данного примера следует сделать три вывода.

1. Как правило, `std::default_random_engine` — это хороший вариант по умолчанию для типичного приложения.
2. Если действительно нужно получить недетерминированные случайные числа, то поможет `std::random_device`.
3. Можно передать конструктору любого генератора случайных чисел *реальное* случайное число, полученное от `std::random_device` (или, например, текущее время на системных часах), чтобы заставить его создавать разные случайные числа при каждом обращении. Это называется *посевом*.

Обратите внимание: std::random_device может откатиться к одному из детерминированных генераторов, если библиотека не поддерживает недетерминированные генераторы.

Генерируем случайные числа и создаем конкретные распределения с помощью STL

Из предыдущего примера мы узнали о генераторах случайных чисел, предоставляемых STL. Генерация случайных чисел тем или иным способом — зачастую лишь половина работы.

Возникает еще один вопрос: для чего нужны эти числа? Мы просто программно «подбрасываем монетку»? Обычно это делается с помощью конструкции

`rand() % 2`, что дает результаты `0` и `1`, которые можно сопоставить с *орлом* и *решкой*. Справедливо; для этого не нужна библиотека (однако эксперты в области случайных чисел знают, что использование лишь нескольких младших битов случайного числа не позволяет получить хорошую подборку случайных чисел).

Что, если мы хотим смоделировать бросок кубика? Конечно, можно написать код `(rand() % 6) + 1` с целью представить результат броска. Для выполнения таких простых задач не нужно использовать библиотеку.

А если мы хотим смоделировать событие, которое случается с вероятностью 66 %? Окей, можно создать формулу наподобие `bool yesno = (rand() % 100 > 66)`. (Погодите, нам следует использовать оператор `>=` или правильнее будет оставить оператор `>`?)

Кроме того, как смоделировать бросок *нечестного* кубика, грани которого могут выпасть с разной вероятностью? Как смоделировать более сложные распределения? Такие задачи могут быстро перерасти в научные. Чтобы сконцентрироваться на наших основных задачах, взглянем на инструменты, предоставляемые STL.

Библиотека содержит более дюжины алгоритмов распределения, которые могут формировать случайные числа для определенных потребностей. В этом примере мы очень кратко рассмотрим их все, а также более детально взглянем на самые полезные.

Как это делается

В этом примере мы будем генерировать случайные числа, придавать им форму и выводить на экран шаблоны распределения. Таким образом, рассмотрим их все и разберем их самые важные свойства, которые могут оказаться полезными, если потребуется смоделировать что-то конкретное.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <random>
#include <map>
#include <string>
#include <algorithm>

using namespace std;
```

2. Для каждого распределения, предоставляемого STL, выведем гистограмму, чтобы увидеть его характеристики, поскольку каждое из них выглядит особенным образом. Гистограмма принимает в качестве аргумента распределение и количество образцов, которые будут взяты из него. Затем создадим генератор

случайных чисел по умолчанию и ассоциативный массив. В последнем будут соотнесены значения, полученные из распределения, со счетчиками, показывающими, как часто встречается то или иное значение. Мы всегда создаем экземпляр генератора случайных чисел, потому что все распределения используются только в качестве *функции для формирования* случайных чисел, которые все еще должны быть сгенерированы.

```
template <typename T>
void print_distro(T distro, size_t samples)
{
    default_random_engine e;
    map<int, size_t> m;
```

3. Возьмем столько образцов, сколько указано в переменной `samples`, и заполним ими ассоциативный массив счетчиков. Таким образом получим очередную гистограмму. Простой вызов `e()` даст необработанное простое число, `distro(e)` придает случайным числам форму с помощью объекта распределения:

```
    for (size_t i {0}; i < samples; ++i) {
        m[distro(e)] += 1;
    }
```

4. Чтобы получить выходные данные, которые помещаются в окно консоли, нужно узнать *самое большое* значение счетчика. Функция `max_element` поможет определить такое значение путем сравнения всех связанных счетчиков в массиве и возвращения итератора, указывающего на узел, содержащий данное значение. Зная это значение, можем определить, на какое число следует разделить все значения счетчиков, чтобы уместить полученный результат в окно консоли.

```
    size_t max_elm (max_element(begin(m), end(m),
        [](const auto &a, const auto &b) {
             return a.second < b.second;
        })->second);
    size_t max_div (max(max_elm / 100, size_t(1)));
```

5. Теперь пройдем по массиву в цикле и выведем полоски из символов `'*'` для всех счетчиков большого размера. Остальные значения отбросим, поскольку некоторые генераторы случайных чисел распределяют числа так широко, что это переполнит наши окна консоли.

```
    for (const auto [randval, count] : m) {
        if (count < max_elm / 200) { continue; }
        cout << setw(3) << randval << " : "
             << string(count / max_div, '*') << '\n';
    }
}
```

6. В функции `main` проверим, предоставил ли пользователь ровно один параметр, который указывает, сколько именно образцов нужно взять из каждого распределения. Если пользователь передал ноль или несколько параметров, то сгенерируем ошибку:

```
int main(int argc, char **argv)
{
    if (argc != 2) {
        cout << "Usage: " << argv[0]
             << " <samples>\n";
        return 1;
    }
```

7. Теперь преобразуем аргумент командной строки в число с помощью вызова `std::stoull`:

```
    size_t samples {stoull(argv[1])};
```

8. Сначала попробуем распределения `uniform_int_distribution` и `normal_distribution`. Они используются в большинстве случаев, когда нужно применить генератор случайных чисел. Все, кто когда-то изучал стохастику в университете, скорее всего, слышали о них. Равномерное распределение принимает два значения, указывая нижнюю и верхнюю границы диапазона, в котором будут распределены случайные значения. Выбрав `0` и `9`, мы получим одинаково часто встречающиеся значения между `0` и `9` (включительно). Нормальное распределение принимает в качестве аргументов *математическое ожидание* и *среднеквадратическое отклонение*.

```
    cout << "uniform_int_distribution\n";
    print_distro(uniform_int_distribution<int>{0, 9}, samples);
    cout << "normal_distribution\n";
    print_distro(normal_distribution<double>{0.0, 2.0}, samples);
```

9. Еще одним очень интересным распределением является `piecewise_constant_distribution`. Оно принимает в качестве аргументов два входных диапазона. Первый диапазон содержит числа, которые указывают границы интервалов. Определив их как `0`, `5`, `10`, `30`, получим три интервала, простирающиеся от `0` до `4`, от `5` до `9` и от `10` до `29`. Еще один входной диапазон определяет веса входных диапазонов. Установив значения этих весов равными `0.2`, `0.3`, `0.5`, мы укажем, что из соответствующих интервалов случайные числа будут получены с вероятностями 20, 30 и 50 %. Внутри каждого из интервалов значения будут иметь одинаковую вероятность выпадения.

```
    initializer_list<double> intervals {0, 5, 10, 30};
    initializer_list<double> weights {0.2, 0.3, 0.5};
    cout << "piecewise_constant_distribution\n";
    print_distro(
        piecewise_constant_distribution<double>{
```

```
        begin(intervals), end(intervals),
        begin(weights)},
    samples);
```

10. Распределение `piecewise_linear_distribution` создается аналогично, но веса работают совершенно по-другому. Для каждой граничной точки интервала существует одно значение веса. При переходе от одной границы к другой вероятность интерполируется линейно. Воспользуемся теми же интервалами, но передадим другой список весов:

```
cout << "piecewise_linear_distribution\n";
initializer_list<double> weights2 {0, 1, 1, 0};
print_distro(
    piecewise_linear_distribution<double>{
        begin(intervals), end(intervals), begin(weights2)},
    samples);
```

11. Распределение Бернулли — это еще одно важное распределение, поскольку распределяет лишь значения *«да/нет»*, *«попадание/промах»* или *«орел/решка»* с конкретной вероятностью. Его выходными значениями будут только `0` и `1`. Еще одним интересным распределением, полезным во многих случаях, является `discrete_distribution`. В нашем случае инициализируем его дискретными значениями `1`, `2`, `4`, `8`. Они интерпретируются как веса для возможных выходных значений от `0` до `3`.

```
cout << "bernoulli_distribution\n";
print_distro(std::bernoulli_distribution{0.75}, samples);
cout << "discrete_distribution\n";
print_distro(discrete_distribution<int>{{1, 2, 4, 8}}, samples);
```

12. Существует множество других генераторов распределений. Они полезны только в очень специфических ситуациях. Если вы никогда о них не слышали, то они, *возможно*, вам и не нужны. Однако, поскольку наша программа создает «аккуратные» гистограммы, показывающие распределение, из интереса выведем их все:

```
cout << "binomial_distribution\n";
print_distro(binomial_distribution<int>{10, 0.3}, samples);
cout << "negative_binomial_distribution\n";
print_distro(
    negative_binomial_distribution<int>{10, 0.8},
    samples);
cout << "geometric_distribution\n";
print_distro(geometric_distribution<int>{0.4}, samples);
cout << "exponential_distribution\n";
print_distro(exponential_distribution<double>{0.4}, samples);
cout << "gamma_distribution\n";
print_distro(gamma_distribution<double>{1.5, 1.0}, samples);
```

```
    cout << "weibull_distribution\n";
    print_distro(weibull_distribution<double>{1.5, 1.0}, samples);
    cout << "extreme_value_distribution\n";
    print_distro(
        extreme_value_distribution<double>{0.0, 1.0},
        samples);
    cout << "lognormal_distribution\n";
    print_distro(lognormal_distribution<double>{0.5, 0.5}, samples);
    cout << "chi_squared_distribution\n";
    print_distro(chi_squared_distribution<double>{1.0}, samples);
    cout << "cauchy_distribution\n";
    print_distro(cauchy_distribution<double>{0.0, 0.1}, samples);
    cout << "fisher_f_distribution\n";
    print_distro(fisher_f_distribution<double>{1.0, 1.0}, samples);
    cout << "student_t_distribution\n";
    print_distro(student_t_distribution<double>{1.0}, samples);
}
```

13. Компиляция и запуск программы дадут следующий результат. Сначала запустим программу с 1000 образцами для каждого распределения (рис. 8.7).

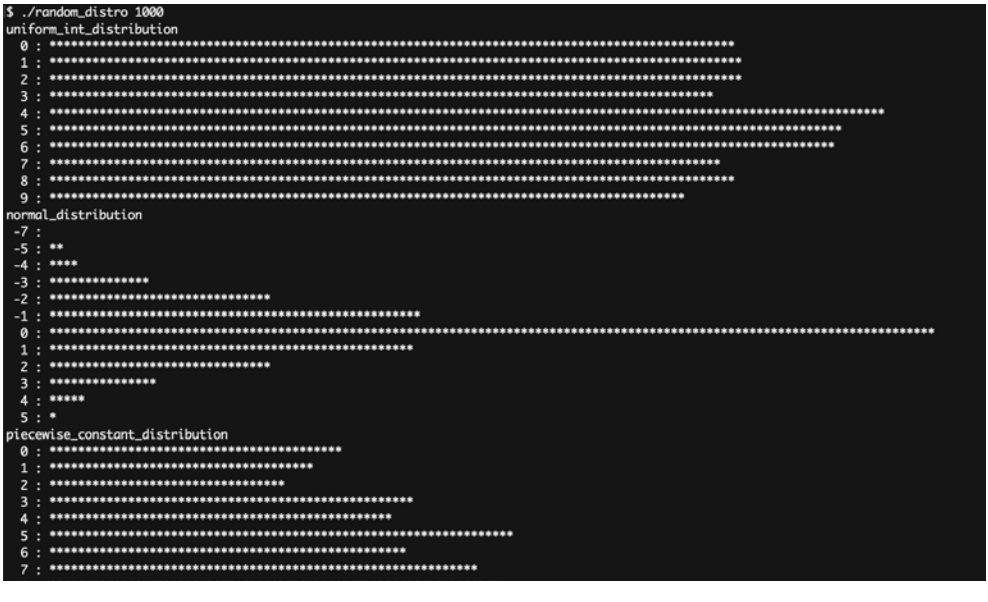

Рис. 8.7

14. Еще один запуск, на этот раз с 1 000 000 образцов для каждого распределения, покажет, что гистограммы выглядят гораздо чище и более характерно для каждого из них. Кроме того, мы увидим, какие распределения генерируются медленно, а какие — быстро (рис. 8.8).

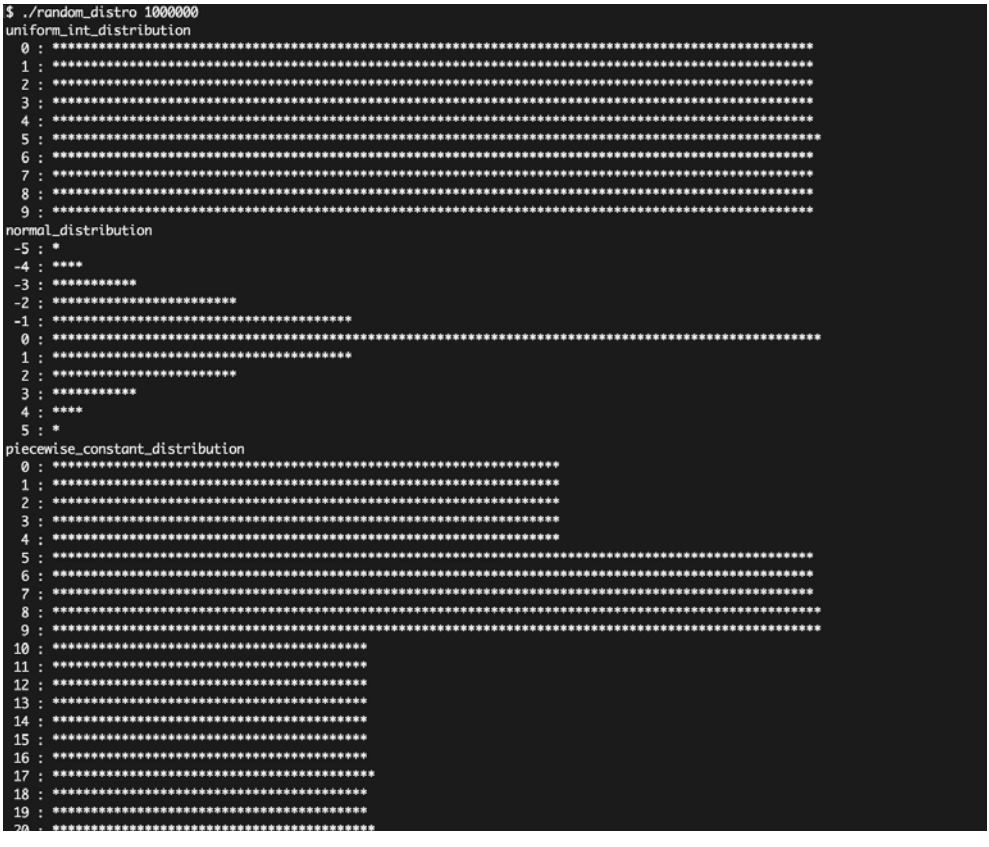

Рис. 8.8

Как это работает

Хотя генераторы случайных чисел нас не интересуют до тех пор, пока работают быстро и создают числа максимально случайным образом, нам *следует* тщательно выбирать распределение в зависимости от решаемой задачи.

Чтобы использовать любое распределение, сначала нужно создать для него соответствующий объект. Мы видели, что разные распределения принимают разные аргументы конструктора. В описании примера мы кратко остановились на некоторых видах распределения, поскольку большинство из них слишком специфичны и/или сложны, чтобы рассматривать их здесь. Не волнуйтесь, все они подробно описаны в документации к C++ STL.

Однако как только появляется экземпляр распределения, можно вызвать его как функцию, которая принимает в качестве единственного параметра объект генератора случайных чисел. Далее объект распределения получает случайное число, придает

ему некую форму (которая полностью зависит от выбранного распределения), а затем возвращает его нам. Это приводит к появлению совершенно разных гистограмм, что мы видели после запуска программы.

Программа, которую мы только что написали, позволит нам получить наиболее полную информацию о разных распределениях. В дополнение к этому рассмотрим самые важные виды распределения (табл. 8.2). Для всех остальных видов распределения вы можете обратиться к документации C++ STL.

Таблица 8.2

Распределение	Описание
uniform_int_distribution	Это распределение принимает в качестве аргументов конструктора нижнюю и верхнюю границы. Вероятность генерации каждого значения, лежащего в пределах этих границ (включительно), будет одинаковой, что делает гистограмму плоской. Данное распределение может представлять собой, например, бросок кубика, поскольку каждая грань может выпасть с одинаковой вероятностью
normal_distribution	Нормальное, или гауссово, распределение встречается в природе практически везде. Его версия из STL принимает в качестве аргументов конструктора математическое ожидание и среднеквадратическое отклонение и формирует гистограмму в виде крыши. Если мы сравним размеры тела, либо IQ людей или других животных, или оценки учеников, то увидим, что эти числа тоже имеют нормальное распределение
bernoulli_distribution	Распределение Бернулли идеально подходит в том случае, если мы хотим подбросить монетку или получить ответ «да/нет». Оно генерирует только значения 0 и 1, единственным параметром конструктора является значение 1
discrete_distribution	Дискретное распределение будет интересно только в том случае, если нужно получить ограниченный набор значений, для каждого из которых мы хотим определить свою вероятность. Его конструктор принимает список весов, и оно будет генерировать случайные числа с вероятностью, зависящей от их веса. Если мы хотим смоделировать случайно распределенные группы крови, когда каждое из четырех значений имеет конкретную вероятность появления, то данное распределение идеально подойдет

9 Параллелизм и конкурентность

В этой главе:

- автоматическое распараллеливание кода, использующего стандартные алгоритмы;
- приостановка программы на конкретный промежуток времени;
- запуск и приостановка потоков;
- выполнение устойчивой к исключениям общей блокировки с помощью `std::unique_lock` и `std::shared_lock`;
- избегание взаимных блокировок с применением `std::scoped_lock`;
- синхронизация конкурентного использования `std::cout`;
- безопасное откладывание инициализации с помощью `std::call_once`;
- отправка выполнения задач в фоновый режим с применением `std::async`;
- реализация идиомы «производитель/потребитель» с использованием `std::condition_variable`;
- реализация идиомы «несколько потребителей/производителей» с помощью `std::condition_variable`;
- распараллеливание отрисовщика множества Мандельброта в ASCII с применением `std::async`;
- реализация небольшой автоматической библиотеки для распараллеливания с использованием `std::future`.

Введение

До C++11 язык C++ не поддерживал параллельные вычисления. Это не значило, что запуск, управление, остановка и синхронизация потоков были невыполнимы, но для каждой операционной системы требовались специальные библиотеки, поскольку потоки по своей природе связаны с ОС.

С появлением C++11 мы получили библиотеку `std::thread`, которая позволяет управлять потоками всех операционных систем. Для синхронизации потоков в C++11 были созданы классы-мьютексы, а также удобные оболочки блокировок в стиле RAII. Вдобавок `std::condition_variable` позволяет отправлять гибкие уведомления о событиях между потоками.

Кроме того, интересными дополнениями являются `std::async` и `std::future`: теперь можно оборачивать произвольные нормальные функции в вызовы `std::async`, чтобы выполнять их асинхронно в фоновом режиме. Такие обернутые функции возвращают объекты типа `std::future`, которые обещают содержать результаты работы функции, и можно сделать что-то еще, прежде чем дождаться их появления.

Еще одно значительное улучшение STL — *политики выполнения*, которые могут быть добавлены к 69 уже существующим алгоритмам. Это дополнение означает, что можно просто добавить один аргумент, описывающий политику выполнения, в существующие вызовы стандартных алгоритмов и получить доступ к параллелизации, не нуждаясь в переписывании сложного кода.

В данной главе мы пройдемся по всем указанным дополнениям, чтобы узнать их самые важные особенности. После этого у нас будет достаточно информации о поддержке параллелизации в STL версии C++17. Мы не станем рассматривать все свойства, только самые важные. Информация, полученная из этой книги, позволит быстро понять остальную часть механизмов распараллеливания, которую можно найти в Интернете в документации к STL версии C++17.

Наконец, в этой главе содержатся два дополнительных примера. В одном из них мы распараллелим отрисовщик множества Мандельброта в ASCII из главы 6, внеся минимальные изменения. В последнем примере реализуем небольшую библиотеку, которая помогает распараллелить выполнение сложных задач неявно и автоматически.

Автоматическое распараллеливание кода, использующего стандартные алгоритмы

В C++17 появилось одно действительно *крупное* расширение для параллелизма: *политики выполнения* для стандартных алгоритмов. Шестьдесят девять алгоритмов были расширены и теперь принимают политики выполнения, чтобы работать параллельно на нескольких ядрах и даже при включенной векторизации.

Для пользователя это значит следующее: если мы уже повсеместно задействуем алгоритмы STL, то можем параллелизовать их работу без особых усилий. Мы *легко* можем дополнить наши приложения параллелизацией, просто добавив один аргумент, описывающий политику выполнения, в существующие вызовы алгоритмов STL.

В данном разделе мы реализуем простую программу (с не самым серьезным сценарием применения), которая генерирует несколько вызовов алгоритмов STL. При этом увидим, как легко использовать политики выполнения C++17, чтобы запустить их в нескольких потоках. В последних подразделах мы более подробно рассмотрим разные политики выполнения.

Как это делается

В данном примере мы напишем программу, использующую некоторые стандартные алгоритмы. Сама программа является скорее примером того, как могут выглядеть реальные сценарии, а не средством решения настоящей рабочей проблемы. Применяя эти стандартные алгоритмы, мы встраиваем политики выполнения, чтобы ускорить выполнение кода.

1. Сначала включим некоторые заголовочные файлы и объявим об использовании пространства имен `std`. Заголовочный файл `execution` мы еще не видели, он появился в C++17.

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

using namespace std;
```

2. В качестве примера создадим функцию-предикат, которая говорит, является ли число четным. Мы воспользуемся ею далее.

```
static bool odd(int n) { return n % 2; }
```

3. Сначала определим в нашей функции `main` большой вектор. Заполним его большим количеством данных, чтобы для выполнения вычислений потребовалось какое-то время. Скорость выполнения этого кода будет значительно различаться в зависимости от того, на каком компьютере работает этот код. Для разных компьютеров скорее подойдут меньшие/большие размеры вектора.

```
int main()
{
    vector<int> d (50000000);
```

4. Чтобы получить большое количество случайных данных для вектора, создадим генератор случайных чисел и распределение и упакуем их в вызываемый объект. Если это кажется странным, пожалуйста, взгляните сначала на примеры, в которых рассматриваются генераторы случайных чисел и распределения в главе 8.

```
    mt19937 gen;
    uniform_int_distribution<int> dis(0, 100000);
    auto rand_num ([=] () mutable { return dis(gen); });
```

5. Теперь воспользуемся стандартным алгоритмом `std::generate`, чтобы заполнить вектор случайными данными. В C++17 существует новая версия этого алгоритма, которая принимает аргумент нового типа: политику выполнения. Здесь поместим `std::par`, что позволит автоматически распараллелить данный код. Сделав это, позволим нескольким потокам заполнять вектор одновременно;

данное действие снижает время выполнения, если компьютер имеет более одного процессора, что обычно верно для современных машин.

```
generate(execution::par, begin(d), end(d), rand_num);
```

6. Метод `std::sort` тоже должен быть вам знаком. Версия C++17 также поддерживает дополнительный аргумент, определяющий политику выполнения:

```
sort(execution::par, begin(d), end(d));
```

7. Это верно и для `std::reverse`:

```
reverse(execution::par, begin(d), end(d));
```

8. Затем воспользуемся функцией `std::count_if`, чтобы подсчитать количество четных чисел в векторе. Мы даже можем распараллелить выполнение алгоритма, просто добавив политику выполнения снова!

```
auto odds (count_if(execution::par, begin(d), end(d), odd));
```

9. Вся эта программа не выполняет никакой реальной научной работы, поскольку мы просто смотрим, как можно распараллелить стандартные алгоритмы, но выведем что-нибудь на экран в конце работы.

```
    cout << (100.0 * odds / d.size())
         << "% of the numbers are odd.\n";
}
```

10. Компиляция и запуск программы дадут следующий результат. К этому моменту интересно посмотреть, как отличается скорость выполнения при использовании алгоритмов без политики выполнения в сравнении со всеми другими политиками выполнения. Данное упражнение оставим на откуп читателю. Попробуйте его сделать; доступными политиками выполнения являются `seq`, `par` и `par_vec`. Для каждой из них мы должны получить разное время выполнения:

```
$ ./auto_parallel
50.4% of the numbers are odd.
```

Как это работает

Поскольку в этом примере мы не отвлекались на решение сложных реальных задач, можем полностью сконцентрироваться на вызовах функций стандартных библиотек. Довольно очевидно, что их параллелизованные версии едва отличаются от классических последовательных вариаций. Они отличаются всего на *один дополнительный* аргумент — политику выполнения.

Взглянем на вызовы и ответим на три основных вопроса:

```
generate(execution::par, begin(d), end(d), rand_num);
sort(    execution::par, begin(d), end(d));
reverse( execution::par, begin(d), end(d));

auto odds (count_if(execution::par, begin(d), end(d), odd));
```

Какие алгоритмы STL можно распараллелить таким образом?

Шестьдесят девять существующих алгоритмов STL получили поддержку параллелизма по стандарту C++17. Появились также семь новых алгоритмов, которые тоже поддерживают параллелизм. Несмотря на то что такое улучшение может оказаться довольно инвазивным для реализации, с точки зрения их интерфейса изменилось не очень много: все они получили дополнительный аргумент `ExecutionPolicy&& policy` и все. Это не значит, что мы *всегда* должны предоставлять аргумент, описывающий политику выполнения. Они просто *дополнительно* поддерживают передачу политики выполнения в качестве первого аргумента.

Перед вами 69 улучшенных стандартных алгоритмов. Кроме того, в этот список включены семь новых алгоритмов, изначально поддерживающих политики выполнения (выделены **полужирным**):

```
std::adjacent_difference
std::adjacent_find
std::all_of
std::any_of
std::copy
std::copy_if
std::copy_n
std::count
std::count_if
std::equal
std::exclusive_scan
std::fill
std::fill_n
std::find
std::find_end
std::find_first_of
std::find_if
std::find_if_not
std::for_each
std::for_each_n
std::generate
std::generate_n
std::includes
std::inclusive_scan
std::inner_product
std::inplace_merge
std::is_heap
std::is_heap_until
std::is_partitioned
std::is_sorted
std::is_sorted_until
std::lexicographical_compare
std::max_element
std::merge
std::min_element
std::minmax_element
```

```
std::mismatch
std::move
std::none_of
std::nth_element
std::partial_sort
std::partial_sort_copy
std::partition
std::partition_copy
std::remove
std::remove_copy
std::remove_copy_if
std::remove_if
std::replace
std::replace_copy
std::replace_copy_if
std::replace_if
std::reverse
std::reverse_copy
std::rotate
std::rotate_copy
std::search
std::search_n
std::set_difference
std::set_intersection
std::set_symmetric_difference
std::set_union
std::sort
std::stable_partition
std::stable_sort
std::swap_ranges
std::transform
std::transform_exclusive_scan
std::transform_inclusive_scan
std::transform_reduce
std::uninitialized_copy
std::uninitialized_copy_n
std::uninitialized_fill
std::uninitialized_fill_n
std::unique
std::unique_copy
```

Улучшение этих алгоритмов — отличная новость! Чем больше алгоритмов STL используется в наших старых программах, тем проще добавить поддержку параллелизма задним числом. Обратите внимание: это не значит, что такие изменения автоматически сделают программу в *N* раз быстрее, поскольку концепция многопроцессорной обработки гораздо сложнее.

Однако вместо того, чтобы разрабатывать собственные сложные параллельные алгоритмы с помощью `std::thread`, `std::async` или внешних библиотек, можно распараллелить выполнение стандартных задач способом, не зависящим от операционной системы.

Как работают эти политики выполнения

Политика выполнения указывает, какую стратегию автоматического распараллеливания необходимо использовать при вызове стандартных алгоритмов.

Следующие три типа политик существуют в пространстве имен `std::execution` (табл. 9.1).

Таблица 9.1

Политика	Значение
sequenced_policy	Алгоритм нужно выполнить последовательно, аналогично оригинальному алгоритму, не имеющему политики выполнения. Глобально доступный экземпляр имеет имя std::execution::seq
parallel_policy	Алгоритм может быть выполнен в нескольких потоках, которые будут работать параллельно. Глобально доступный экземпляр называется std::execution::par
parallel_unsequenced_policy	Алгоритм может быть выполнен в нескольких потоках, которые поделят работу. Вдобавок можно векторизовать код. В этом случае доступ к контейнеру способен чередоваться между потоками, а также внутри одного потока благодаря векторизации. Глобально доступный экземпляр носит имя std::execution::par_unseq

Политики выполнения подразумевают конкретные ограничения. Чем они строже, тем больше мер по распараллеливанию можно позволить:

- все элементы функций доступа, используемые параллелизованными алгоритмами, *не должны* вызывать *взаимных блокировок* и *гонок*;
- в случае параллелизации и векторизации все функции получения доступа *не должны* использовать блокирующую синхронизацию.

До тех пор пока подчиняемся этим правилам, мы не столкнемся с ошибками, которые могут появиться в параллельных версиях алгоритмов STL.

Обратите внимание: правильное использование параллельных алгоритмов STL не всегда гарантирует ускорение работы. В зависимости от того, какую задачу мы пытаемся решить, ее размера, эффективности наших структур и других методов доступа, измеряемое ускорение будет значительно различаться или даже и вовсе не произойдет. Многопроцессорная обработка — это все еще довольно сложно.

Что означает понятие «векторизация»

Векторизация — это свойство, которое должны поддерживать как процессор, так и компилятор. Кратко рассмотрим простой пример, чтобы понять суть векторизации

и как она работает. Допустим, нужно сложить числа, находящиеся в очень большом векторе. Простая реализация данной задачи может выглядеть так:

```
std::vector<int> v {1, 2, 3, 4, 5, 6, 7 /*...*/};

int sum {std::accumulate(v.begin(), v.end(), 0)};
```

Компилятор в конечном счете сгенерирует цикл из вызова `accumulate`, который может выглядеть следующим образом:

```
int sum {0};
for (size_t i {0}; i < v.size(); ++i) {
    sum += v[i];
}
```

С этого момента при разрешенной и включенной векторизации компилятор может создать следующий код. Цикл выполняет четыре шага сложения в одной итерации цикла, что сокращает количество итераций в четыре раза. Для простоты пример не работает с остатком, если вектор не содержит `N * 4` элементов:

```
int sum {0};
for (size_t i {0}; i < v.size() / 4; i += 4) {
    sum += v[i] + v[i+1] + v[i + 2] + v[i + 3];
}
// если операция v.size() / 4 имеет остаток,
// в реальном коде также нужно это обработать.
```

Зачем это делать? Многие процессоры предоставляют инструкции, которые могут выполнять математические операции наподобие `sum += v[i] + v[i+1] + + v[i + 2] + v[i + 3];` всего за *один шаг*. Сжатие *большого количества* математических операций в *минимальное количество* инструкций — наша цель, поскольку это ускоряет программу.

Автоматическую векторизацию выполнять сложно, поскольку компилятору нужно в некоторой степени понимать нашу программу, чтобы ускорить ее, не нарушая *правильности*. По крайней мере помочь компилятору можно, используя стандартные алгоритмы максимально часто, поскольку компилятору проще понять их, чем запутанные циклы со сложными зависимостями.

Приостанавливаем программу на конкретный промежуток времени

Простая и удобная возможность управления потоками добавлена в C++11. В данной версии появилось пространство имен `this_thread`, содержащее функции, которые влияют только на вызывающий поток. Оно включает две разные функции, позволяющие приостановить поток на определенный промежуток времени, это позволяет перестать использовать внешние библиотеки или библиотеки, зависящие от операционной системы.

В этом примере мы сконцентрируемся на том, как приостанавливать потоки на определенный промежуток времени.

Как это делается

В этом примере мы напишем короткую программу, которая приостанавливает основной поток на определенные промежутки времени.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `chrono_literals`. Второе пространство содержит удобные аббревиатуры для описания промежутков времени:

```
#include <iostream>
#include <chrono>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. Сразу же приостановим основной поток на 5 секунд 300 миллисекунд. Благодаря пространству имен `chrono_literals` можем выразить эти промежутки времени в читабельном формате:

```
int main()
{
    cout << "Going to sleep for 5 seconds"
            " and 300 milli seconds.\n";
    this_thread::sleep_for(5s + 300ms);
```

3. Последним выражением приостановки являлось `relative`. Кроме того, можно выразить запросы `absolute` на приостановку. Приостановим поток на 3 секунды, начиная *с текущего момента*:

```
    cout << "Going to sleep for another 3 seconds.\n";
    this_thread::sleep_until(
        chrono::high_resolution_clock::now() + 3s);
```

4. Перед завершением программы выведем на экран какое-нибудь сообщение, что укажет на окончание второго периода приостановки.

```
    cout << "That's it.\n";
}
```

5. Компиляция и запуск программы дадут следующие результаты. В Linux, Mac и других UNIX-подобных операционных системах имеется команда `time`, принимающая другую команду, чтобы выполнить ее и определить время, которое требуется на ее выполнение. Запуск нашей программы с помощью команды `time` показывает: она работала 8,32 секунды, это значение примерно равно 5,3 и 3 секундам, на которые мы приостанавливали программу. При запуске

программы можно определить промежуток времени между появлением строк, выводимых на консоль:

```
$ time ./sleep
Going to sleep for 5 seconds and 300 milli seconds.
Going to sleep for another 3 seconds.
That's it.
real 0m8.320s
user 0m0.005s
sys  0m0.003s
```

Как это работает

Функции `sleep_for` и `sleep_until` появились в версии C++11 и находятся в пространстве имен `std::this_thread`. Они блокируют выполнение текущего потока (но не процесса или программы) на конкретный промежуток времени. Поток не потребляет время процессора на протяжении блокировки. Он просто помещается операционной системой в неактивное состояние. ОС, конечно же, напоминает себе о необходимости возобновить поток. Самое лучшее заключается в том, что нам не придется волноваться, в какой операционной системе запущена программа, поскольку эту информацию от нас абстрагирует STL.

Функция `this_thread::sleep_for` принимает значение типа `chrono::duration`. В простейшем случае это просто `1s` или `5s + 300ms`, как это было показано в нашем примере кода. Чтобы получить возможность применять такие удобные литералы, нужно объявить об использовании пространства имен `std::chrono_literals;`.

Функция `this_thread::sleep_until` принимает значение типа `chrono::time_point` вместо промежутка времени. Это удобно в том случае, если нужно приостановить поток до наступления конкретного момента времени.

Точность момента пробуждения зависит от операционной системы. Она будет довольно *высокой* для большинства ОС, но могут возникнуть проблемы, если требуется точность вплоть до наносекунд.

Приостановить выполнение потока на короткий промежуток времени также можно с помощью функции `this_thread::yield`. Она *не принимает* аргументы; это значит, что неизвестно, как надолго будет отложено выполнение потока. Причина заключается в следующем: данная функция не знает о том, как приостанавливать потоки на какое-то время. Она просто говорит ОС, что может перепланировать выполнение других потоков любых процессов. Если таких потоков нет, то поток возобновится мгновенно. По этой причине функция `yield` зачастую менее полезна, чем приостановка на короткий промежуток времени, установленный заранее.

Запускаем и приостанавливаем потоки

Еще одним дополнением, появившимся в C++11, является класс `std::thread`. Он предоставляет простой способ запускать и приостанавливать потоки, не прибегая к использованию внешних библиотек и знаний о том, в какой операционной системе запущен процесс. Все это включено в STL.

В данном примере мы реализуем программу, которая запускает и останавливает потоки. Далее рассмотрим информацию о том, что с ними делать после запуска.

Как это делается

В этом примере мы запустим несколько потоков и увидим, как ведет себя программа, когда мы задействуем несколько ядер процессора, чтобы выполнить разные части ее кода одновременно.

1. Сначала включим всего два заголовочных файла, а затем объявим об использовании пространств имен `std` и `chrono_literals`:

```
#include <iostream>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. Чтобы запустить поток, следует указать, какой код он должен выполнить. Поэтому определим функцию, которую ему нужно выполнить. Функции — естественные потенциальные входные точки для потоков. Наша функция-пример принимает аргумент `i`, выступающий в роли идентификатора потока. Таким образом, можно сказать, какой поток отобразил то или иное сообщение. Кроме того, воспользуемся идентификатором потока с целью указать, что различные потоки нужно приостановить на разные промежутки времени. Это позволит убедиться в том, что они не пытаются задействовать команду `cout` одновременно. Если такая ситуация произойдет, то выходные данные будут искажены. Другой пример в настоящей главе посвящен именно этой проблеме.

```
static void thread_with_param(int i)
{
    this_thread::sleep_for(1ms * i);
    cout << "Hello from thread " << i << '\n';
    this_thread::sleep_for(1s * i);
    cout << "Bye from thread " << i << '\n';
}
```

3. В функции `main` (просто из любопытства) выведем на экран информацию о том, сколько потоков можно запустить в одно время с помощью `std::thread::hardware_concurrency`. Данное значение зависит от того, сколько ядер имеет процессор и сколько ядер поддерживается реализацией STL. Это говорит о том, что значения будут различаться для разных компьютеров.

```
i nt main()
{
    cout << thread::hardware_concurrency()
         << " concurrent threads are supported.\n";
```

4. Наконец, начнем работать с потоками. Запустим три потока с разными идентификаторами. При создании экземпляра потока с помощью выражения

наподобие `t {f, x}` получаем вызов функции `f(x)`. Таким образом можно передавать функциям `thread_with_param` разные аргументы для каждого потока:

```
thread t1 {thread_with_param, 1};
thread t2 {thread_with_param, 2};
thread t3 {thread_with_param, 3};
```

5. Поскольку данные потоки запущены свободно, нужно остановить их, когда они закончат выполнять свою работу. Сделаем это с помощью функции `join`. Она *заблокирует* вызов потока до тех пор, пока вызываемый поток не отработает:

```
t1.join();
t2.join();
```

6. Альтернативой присоединению является *открепление*. Если мы не вызовем функцию `join` или `detach`, то все приложение завершится довольно шумно, как только будет выполнен деструктор объекта потока. Путем вызова функции `detach` указываем `thread`, что хотим продолжения работы потока номер 3 даже после того, как его экземпляр будет разрушен:

```
t3.detach();
```

7. Перед завершением функции `main` и всей программы выведем еще одно сообщение:

```
    cout << "Threads joined.\n";
}
```

8. Компиляция и запуск программы дадут следующий результат. Мы можем увидеть, что моя машина имеет восемь ядер процессора. Далее сообщения *hello* видим из всех потоков, а сообщения *bye* — лишь из двух, которые мы объединили. Поток 3 все еще ожидает завершения трехсекундного ожидания, но вся программа уже завершилась после того, как поток 2 завершил свое двухсекундное ожидание. Таким образом, мы не можем увидеть прощальное сообщение потока 3, поскольку он был уничтожен:

```
$ ./threads
8 concurrent threads are supported.
Hello from thread 1
Hello from thread 2
Hello from thread 3
Bye from thread 1
Bye from thread 2
Threads joined.
```

Как это работает

Запуск и остановку потоков выполнить очень просто. Многопроцессорная обработка начинает усложняться в момент, когда потокам нужно работать вместе (делить ресурсы, ожидать завершения других потоков и т. д.).

Чтобы запустить поток, нужно иметь функцию, которую он будет выполнять. Функция не обязательно должна быть особенной, поскольку в потоке можно вы-

полнить практически любую функцию. Напишем небольшую программу-пример, которая запускает поток и ожидает его завершения:

```
void f(int i) { cout << i << '\n'; }

int main()
{
    thread t {f, 123};
    t.join();
}
```

Вызов конструктора `std::thread` принимает указатель на функцию или вызываемый объект; за ним следуют аргументы, которые нужно использовать в вызове функции. Конечно, можете также запустить поток или функцию, не принимающие никаких параметров.

При наличии в системе нескольких ядер процессора потоки можно выполнять параллельно и конкурентно. В чем заключается разница? Если компьютер имеет всего одно ядро ЦП, то можно создать множество потоков, работающих параллельно, но не конкурентно, поскольку ядро способно запускать лишь один поток в любой момент времени. Потоки запускаются и чередуются, где каждый поток выполняется какую-то часть секунды, затем приостанавливается, после чего следующий поток получает время (для пользователей-людей кажется, что потоки выполняются одновременно). Если потокам не нужно делить одно ядро, то они могут быть запущены конкурентно и *действительно работать одновременно*.

К этому моменту мы *не контролируем* следующие детали:

- *порядок*, в котором потоки чередуются на одном ядре;
- *приоритет* потока, указывающий, что один поток главнее другого;
- *распределение* потоков между ядрами. Вполне *возможна* ситуация, когда все потоки будут выполняться на одном ядре, несмотря на то что машина имеет более 100 ядер.

Большая часть операционных систем предоставляет возможности управления этими аспектами многопроцессорной обработки, но на текущий момент данные функции *не включены* в STL.

Однако можно запускать и останавливать потоки и указывать им, когда и над чем работать и когда останавливаться. Этого должно быть достаточно для большинства приложений. В данном разделе мы создали три дополнительных потока. После этого *объединили* большую их часть и *открепили* последний. Подытожим на одном рисунке все, что произошло (рис. 9.1).

Читая рисунок сверху вниз, мы заметим, что в какой-то момент разбиваем рабочий поток программы на четыре потока. Мы запускаем три дополнительных потока, которые совершают некие действия (а именно, ожидают и выводят сообщения), но после их запуска основной поток, выполняющий функцию `main`, остается без работы.

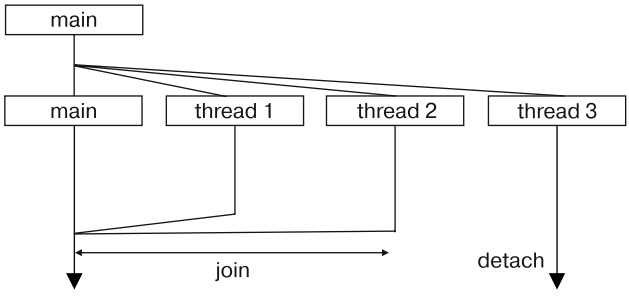

Рис. 9.1

Когда поток завершает выполнение своей функции, он возвращает значение, возвращенное ею. Затем стандартная библиотека «делает уборку», что приводит к удалению потока из планировщика системы и, возможно, его уничтожению, но волноваться об этом не нужно.

Единственное, о чем *следует* волноваться, — это *объединение*. Когда поток вызывает функцию `x.join()` для объекта другого потока, его выполнение приостанавливается до того, как будет выполнен поток `x`. Обратите внимание: нас ничто не спасет при попадании потока в бесконечный цикл! Если нужно, чтобы поток продолжал существовать до тех пор, пока не решит завершиться, то можно вызвать функцию `x.detach()`. После этого у нас не будет возможности управлять потоком. Независимо от принятого решения, мы *должны* всегда *объединять* или *откреплять* потоки. Если мы не сделаем этого, то деструктор объекта `thread` вызовет функцию `std::terminate()`, что приведет к внезапному завершению работы приложения.

В момент, когда функция `main` возвращает значение, приложение заканчивает работу. Однако в это же время наш открепленный поток `t3` все еще находится в приостановленном состоянии и не успевает отправить сообщение *bye* на консоль. Операционной системе это неважно: она просто завершает всю программу, не дожидаясь завершения данного потока. Указанный факт важно иметь в виду. Если дополнительный поток должен был соревноваться за что-то важное, то нужно было бы *подождать* его завершения в функции `main`.

Выполняем устойчивую к исключениям общую блокировку с помощью std::unique_lock и std::shared_lock

Поскольку работа потоков значительно зависит от поддержки операционной системы, а STL предоставляет хорошие интерфейсы, позволяющие абстрагироваться от операционных систем, разумно также предоставить поддержку STL для *синхронизации* между потоками. Таким образом, можно не только запускать и останавливать потоки без внешних библиотек, но и синхронизировать их с помощью абстракций из одной объединенной библиотеки — STL.

В этом разделе мы взглянем на классы-мьютексы STL и абстракции блокировки RAII. Поэкспериментируем с ними в нашей конкретной реализации примера, а также изучим другие вспомогательные средства синхронизации, предоставляемые STL.

Как это делается

В этом примере мы напишем программу, которая использует экземпляр класса `std::shared_mutex` в *эксклюзивном* и *коллективном* режимах, и увидим, что это значит. Кроме того, не будем вызывать функции `lock` и `unlock` самостоятельно, а сделаем это с помощью вспомогательных функций RAII.

1. Сначала включим все необходимые заголовочные файлы. Поскольку мы задействуем функции и структуры данных STL, а также временные литералы, объявим об использовании пространств имен `std` и `chrono_literal`:

```
#include <iostream>
#include <shared_mutex>
#include <thread>
#include <vector>

using namespace std;
using namespace chrono_literals;
```

2. Вся программа строится вокруг одного общего мьютекса, поэтому для простоты объявим его глобальный экземпляр:

```
shared_mutex shared_mut;
```

3. Мы будем использовать вспомогательные функции RAII `std::shared_lock` и `std::unique_lock`. Чтобы их имена выглядели более понятными, определим для них короткие псевдонимы:

```
using shrd_lck = shared_lock<shared_mutex>;
using uniq_lck = unique_lock<shared_mutex>;
```

4. Прежде чем начнем писать функцию `main`, определим две вспомогательные функции, которые пытаются заблокировать мьютекс в *эксклюзивном* режиме. Эта функция создаст экземпляр класса `unique_lock` для общего мьютекса. Второй аргумент конструктора `defer_lock` указывает объекту поддерживать блокировку снятой. В противном случае его конструктор попробует заблокировать мьютекс, а затем будет удерживать его до завершения. Далее вызываем метод `try_lock` для объекта `exclusive_lock`. Этот вызов немедленно вернет булево значение, которое говорит, получили мы блокировку или же мьютекс уже был заблокирован кем-то еще.

```
static void print_exclusive()
{
    uniq_lck l {shared_mut, defer_lock};
```

```
    if (l.try_lock()) {
        cout << "Got exclusive lock.\n";
    } else {
        cout << "Unable to lock exclusively.\n";
    }
}
```

5. Другая вспомогательная функция также пытается заблокировать мьютекс в эксклюзивном режиме. Она делает это до тех пор, пока не получит блокировку. Затем мы симулируем какую-нибудь ошибку, генерируя исключение (содержащее лишь простое целое число). Несмотря на то что это приводит к мгновенному выходу контекста, в котором мы хранили заблокированный мьютекс, последний будет освобожден. Это происходит потому, что деструктор объекта `unique_lock` освободит блокировку в любом случае по умолчанию.

```
static void exclusive_throw()
{
    uniq_lck l {shared_mut};
    throw 123;
}
```

6. Теперь перейдем к функции `main`. Сначала откроем еще одну область видимости и создадим экземпляр класса `shared_lock`. Его конструктор мгновенно заблокирует мьютекс в коллективном режиме. Мы увидим, что это значит, в следующих шагах.

```
int main()
{
    {
        shrd_lck sl1 {shared_mut};
        cout << "shared lock once.\n";
```

7. Откроем еще одну область видимости и создадим второй экземпляр типа `shared_lock` для того же мьютекса. Теперь у нас есть два экземпляра типа `shared_lock`, и оба содержат общую блокировку мьютекса. Фактически можно создать произвольно большое количество экземпляров типа `shared_lock` для одного мьютекса. Затем вызываем функцию `print_exclusive`, которая пытается заблокировать мьютекс в *эксклюзивном* режиме. Эта операция не увенчается успехом, поскольку он уже находится в *коллективном* режиме.

```
        {
            shrd_lck sl2 {shared_mut};
            cout << "shared lock twice.\n";
            print_exclusive();
        }
```

8. После выхода из самой поздней области видимости деструктор объекта `sl2` типа `shared_lock` освобождает свою общую блокировку мьютекса. Функция `print_exclusive` снова даст сбой, поскольку мьютекс все еще находится в коллективном режиме блокировки.

```
        cout << "shared lock once again.\n";
        print_exclusive();
    }
    cout << "lock is free.\n";
```

9. После выхода из второй области видимости все объекты типа `shared_lock` подвергнутся уничтожению и мьютекс снова будет находиться в разблокированном состоянии. Теперь наконец можно заблокировать мьютекс в эксклюзивном режиме. Сделаем это путем вызовов `exclusive_throw` и `print_exclusive`. Помните, что мы генерируем исключение в вызове `exclusive_throw`. Но поскольку `unique_lock` — это объект RAII, который помогает защититься от исключений, мьютекс снова будет разблокирован независимо от того, что вернет вызов `exclusive_throw`. Таким образом, функция `print_exclusive` не будет ошибочно блокировать все еще заблокированный мьютекс:

```
    try {
        exclusive_throw();
    } catch (int e) {
        cout << "Got exception " << e << '\n';
    }
    print_exclusive();
}
```

10. Компиляция и запуск программы дадут следующий результат. Первые две строки показывают наличие двух экземпляров общей блокировки. Затем функция `print_exclusive` дает сбой при попытке заблокировать мьютекс в эксклюзивном режиме. После того как мы покинем внутреннюю область видимости и разблокируем вторую общую блокировку, функция `print_exclusive` все еще будет давать сбой. После выхода из второй области видимости, что наконец снова освободит мьютекс, функции `exclusive_throw` и `print_exclusive` смогут заблокировать мьютекс:

```
$ ./shared_lock
shared lock once.
shared lock twice.
Unable to lock exclusively.
shared lock once again.
Unable to lock exclusively.
lock is free.
Got exception 123
Got exclusive lock.
```

Как это работает

При просмотре документации C++ может показаться несколько странным факт существования разных классов мьютексов и абстракций блокировок RAII. Прежде чем рассматривать наш конкретный пример кода, подытожим все, что может предложить STL.

Классы мьютексов

Термин mutex расшифровывается как **mut**ual **ex**clusion (взаимное исключение). Чтобы предотвратить неуправляемое изменение одного объекта несколькими конкурирующими потоками, способное привести к повреждению данных, можно использовать объекты мьютексов. STL предоставляет разные классы мьютексов, которые хороши в разных ситуациях. Все они похожи в том, что имеют методы `lock` и `unlock`.

Когда некий поток первым вызывает метод `lock()` для мьютекса, который не был заблокирован ранее, он получает контроль над мьютексом. На данном этапе другие потоки будут блокироваться при вызове метода `lock` до тех пор, пока первый поток не вызовет снова метод `unlock`. Класс `std::mutex` может делать именно это.

В STL существует множество разных классов мьютексов (табл. 9.2).

Таблица 9.2

Имя типа	Описание
mutex	Стандартный мьютекс, имеющий методы lock и unlock. Предоставляет дополнительный неблокирующий метод try_lock
timed_mutex	Аналогичен классу mutex, но предоставляет дополнительные методы try_lock_for и try_lock_until, которые позволяют заблокировать мьютекс на какое-то время
recursive_mutex	Аналогичен классу mutex, но если поток заблокирует экземпляр этого класса, то может заблокировать один мьютекс несколько раз без блокировки. Данный мьютекс будет освобожден после того, как обладающий им поток вызовет метод unlock столько раз, сколько раз он вызвал метод lock
recursive_timed_mutex	Предоставляет свойства классов timed_mutex и recursive_mutex
shared_mutex	Этот мьютекс выделяется тем, что его можно заблокировать в эксклюзивном и в коллективном режимах. В эксклюзивном он ведет себя так же, как и стандартный класс mutex. Если же поток заблокирует его в коллективном режиме, то другие потоки также могут заблокировать его в этом режиме. Он будет разблокирован, как только последний владелец блокировки в коллективном режиме освободит его. Пока мьютекс заблокирован в коллективном режиме, его нельзя заблокировать в эксклюзивном. Данное поведение очень похоже на поведение shared_ptr, только мы управляем не памятью, а владением блокировкой
shared_timed_mutex	Объединяет свойства классов shared_mutex и timed_mutex в коллективном и эксклюзивном режимах

Классы блокировок

Работать с многопоточностью легко и просто до тех пор, пока потоки просто блокируют мьютекс, получают доступ к защищенному от конкурентности объекту, а затем снова разблокируют мьютекс. Как только программист забывает разблокировать мьютекс после одной из блокировок, ситуация значительно усложняется.

В лучшем случае программа просто мгновенно зависает, и вызов, не выполняющий разблокировку, быстро выявляется. Такие ошибки, однако, очень похожи на утечки памяти, которые случаются, если отсутствуют вызовы `delete`.

Для управления памятью существуют вспомогательные классы `unique_ptr`, `shared_ptr` и `weak_ptr`. Они предоставляют очень удобный способ избежать утечек памяти. Такие классы существуют и для мьютексов. Простейшим из них является `std::lock_guard`. Его можно использовать следующим образом:

```
void critical_function()
{
    lock_guard<mutex> l {some_mutex};

    // критический раздел
}
```

Конструктор элемента `lock_guard` принимает мьютекс, для которого мгновенно вызывает метод `lock`. Весь вызов конструктора заблокируется до тех пор, пока тот не получит блокировку для мьютекса. При разрушении объекта он разблокирует мьютекс. Таким образом, понять цикл блокировки/разблокировки сложно, поскольку она происходит автоматически.

В STL версии C++17 предоставляются следующие разные абстракции блокировок RAII (табл. 9.3). Все они принимают аргумент шаблона, который будет иметь тот же тип, что и мьютекс (однако, начиная с C++17, компилятор может вывести этот тип самостоятельно).

Таблица 9.3

Имя	Описание
lock_guard	Предоставляет только конструктор и деструктор, которые блокируют и разблокируют мьютекс
scoped_lock	Аналогичен классу lock_guard, но поддерживает произвольно большое количество мьютексов в конструкторе. Освободит их в противоположном порядке в деструкторе
unique_lock	Блокирует мьютекс в эксклюзивном режиме. Конструктор также принимает аргументы, которые указывают ему сделать тайм-аут вместо постоянной блокировки. Можно и вовсе не блокировать мьютекс или предположить, что он был уже заблокирован, или только попытаться заблокировать мьютекс. Дополнительные методы позволяют заблокировать и разблокировать мьютекс во время существования блокировки unique_lock
shared_lock	Аналогичен unique_lock, но все операции применяются к мьютексу в коллективном режиме

В то время как `lock_guard` и `scoped_lock` имеют простейшие интерфейсы, которые состоят только из конструктора и деструктора, `unique_lock` и `shared_lock` более сложны, но и более гибки. В последующих примерах мы увидим, как еще их можно использовать, помимо простой блокировки.

Вернемся к коду примера. Хотя код запускался только в контексте одного потока, мы увидели, что он собирался использовать вспомогательные классы для блокировки. Псевдоним типа `shrd_lck` расшифровывается как `shared_lock<shared_mutex>` и позволяет блокировать экземпляр мьютекса несколько раз в коллективном режиме. До тех пор пока существуют `sl1` и `sl2`, никакие вызовы `print_exclusive` не могут заблокировать мьютекс в эксклюзивном режиме. Это все еще просто.

Теперь перейдем к эксклюзивным функциям блокировки, которые появились позднее в функции `main`:

```
int main()
{
    {
        shrd_lck sl1 {shared_mut};
        {
            shrd_lck sl2 {shared_mut};
            print_exclusive();
        }
        print_exclusive();
    }
    try {
        exclusive_throw();
    } catch (int e) {
        cout << "Got exception " << e << '\n';
    }
    print_exclusive();
}
```

Важная деталь — после возвращения из `exclusive_throw` функция `print_exclusive` снова может заблокировать мьютекс, несмотря на то что `exclusive_throw` завершила работу некорректно из-за генерации исключения.

Еще раз взглянем на функцию `print_exclusive`, поскольку в ней был использован странный вызов конструктора:

```
void print_exclusive()
{
    uniq_lck l {shared_mut, defer_lock};

    if (l.try_lock()) {
        // ...
    }
}
```

Мы предоставили `shared_mut` и `defer_lock` в качестве аргументов конструктора для `unique_lock` в данной процедуре. `defer_lock` — пустой глобальный объект, который послужит для выбора другого конструктора класса `unique_lock`, просто не блокирующего мьютекс. Позднее можно вызвать функцию `l.try_lock()`, которая не блокирует мьютекс. Если мьютекс уже был заблокирован, то можно сделать что-то еще. При полученной блокировке деструктор поможет выполнить уборку.

Избегаем взаимных блокировок с применением std::scoped_lock

Если бы взаимные блокировки происходили на дорогах, то выглядели бы так (рис. 9.2).

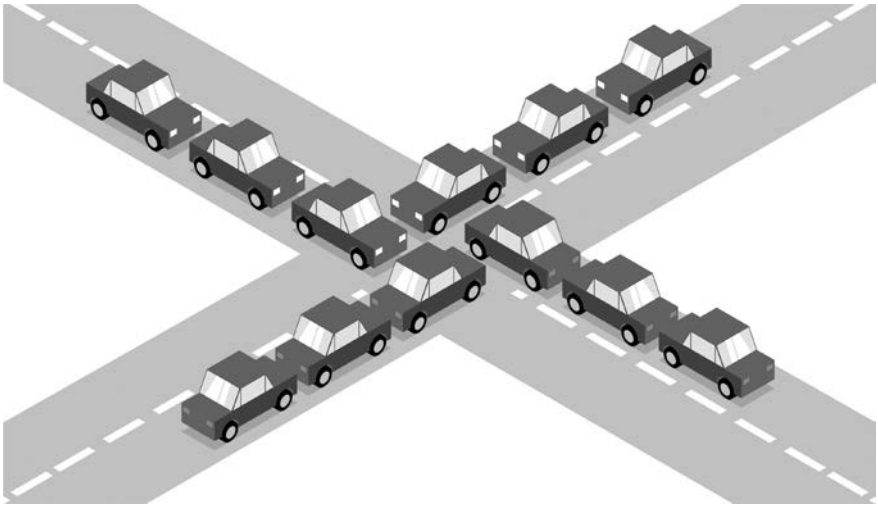

Рис. 9.2

Чтобы снова запустить движение, понадобился бы большой кран, который в случайном порядке выбирает по одной машине из центра перекрестка и удаляет ее. Если это невозможно, то достаточное количество водителей должны взаимодействовать друг с другом. Взаимную блокировку могут разрешить все водители, едущие в одном направлении, сдвинувшись на несколько метров назад, позволяя продвинуться остальным водителям.

В многопоточных программах такие ситуации должен разрешать программист. Однако слишком легко потерпеть неудачу, если программа сама по себе довольно сложная.

В этом примере мы напишем код, намеренно создающий взаимную блокировку. Затем увидим, как писать код, который получает те же ресурсы, что привели другой код к взаимной блокировке, но воспользуемся новым классом блокировки STL `std::scoped_lock`, появившимся в C++17 с целью избежать этой ошибки.

Как это делается

Код этого примера состоит из двух пар функций, которые должны быть выполнены конкурирующими потоками, они получают два ресурса в форме мьютексов. Одна пара создает взаимную блокировку, а вторая избегает такой ситуации. В функции `main` мы опробуем их в деле.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `chrono_literals`:

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;
using namespace chrono_literals;
```

2. Затем создадим два объекта мьютексов, которые понадобятся для создания взаимной блокировки:

```
mutex mut_a;
mutex mut_b;
```

3. Чтобы создать взаимную блокировку с двумя ресурсами, нужны две функции. Одна пробует заблокировать мьютекс `A`, а затем и мьютекс `B`, а другая сделает это в противоположном порядке. Позволив обеим функциям приостановиться между блокировками, можно гарантировать, что код навсегда попадет во взаимную блокировку. (Это делается только для демонстрации. Программа, не содержащая команд по приостановке потоков, может запуститься успешно и не столкнуться с взаимной блокировкой, если запускать ее раз за разом.) Обратите внимание: мы не используем символ `'\n'` для вывода на экран перевода строки, а применяем для данных целей `endl`. Он не только выполняет перевод строки, но еще и опустошает буфер потока `cout`, поэтому можно убедиться, что операции вывода не объединяются и не откладываются:

```
static void deadlock_func_1()
{
    cout << "bad f1 acquiring mutex A..." << endl;
    lock_guard<mutex> la {mut_a};
    this_thread::sleep_for(100ms);
    cout << "bad f1 acquiring mutex B..." << endl;
    lock_guard<mutex> lb {mut_b};
    cout << "bad f1 got both mutexes." << endl;
}
```

4. Как мы и говорили на предыдущем шаге, функция `deadlock_func_2` выглядит точно так же, как и `deadlock_func_1`, но блокирует мьютексы A и B в противоположном порядке:

```
static void deadlock_func_2()
{
    cout << "bad f2 acquiring mutex B..." << endl;
    lock_guard<mutex> lb {mut_b};
    this_thread::sleep_for(100ms);
    cout << "bad f2 acquiring mutex A..." << endl;
    lock_guard<mutex> la {mut_a};
    cout << "bad f2 got both mutexes." << endl;
}
```

5. Теперь напишем свободный от взаимных блокировок вариант этих функций. Они используют класс `scoped_lock`, блокирующий все мьютексы, которые мы предоставляем в качестве аргументов конструктора. Его деструктор снова их разблокирует. При блокировании мьютексов он изнутри применяет стратегию избегания взаимных блокировок. Обратите внимание: обе функции все еще используют мьютексы A и B в разном порядке:

```
\
static void sane_func_1()
{
    scoped_lock l {mut_a, mut_b};
    cout << "sane f1 got both mutexes." << endl;
}
static void sane_func_2()
{
    scoped_lock l {mut_b, mut_a};
    cout << "sane f2 got both mutexes." << endl;
}
```

6. В функции `main` пройдем по двум сценариям. Сначала воспользуемся *внятными* функциями в многопоточном контексте:

```
int main()
{
    {
        thread t1 {sane_func_1};
        thread t2 {sane_func_2};
        t1.join();
        t2.join();
    }
```

7. Затем воспользуемся функциями, создающими взаимные блокировки, которые не следуют стратегиям избегания взаимных блокировок:

```
    {
        thread t1 {deadlock_func_1};
        thread t2 {deadlock_func_2};
        t1.join();
        t2.join();
    }
}
```

8. Компиляция и запуск программы дадут следующий результат. В первых двух строках показывается, что *внятный* сценарий блокировки работает и обе функции возвращают свое значение и не блокируются навсегда. Две другие функции создают взаимную блокировку. Мы можем сказать, что это точно взаимная блокировка, поскольку видим, как они выводят на экран строки, которые указывают отдельным потокам блокировать мьютексы A и B, а затем вечно ожидают. Обе функции не достигают момента, когда успешно блокируют оба мьютекса. Можно оставить программу включенной на часы, дни и годы, и ничего не произойдет.

Это приложение нужно завершить снаружи, например нажав Ctrl+C:

```
$ ./avoid_deadlock
sane f1 got both mutexes
sane f2 got both mutexes
bad f2 acquiring mutex B...
bad f1 acquiring mutex A...
bad f1 acquiring mutex B...
bad f2 acquiring mutex A...
```

Как это работает

Реализуя код, намеренно вызывающий взаимную блокировку, мы увидели, как быстро может возникнуть этот нежелательный сценарий. В крупном проекте, где несколько программистов пишут код, который должен разделять один набор ресурсов, защищенных мьютексами, всем программистам необходимо следовать *одному порядку* при блокировании и разблокировании мьютексов. Несмотря на то что таким стратегиям или правилам следовать очень просто, о них легко и забыть. Еще одним термином для этой проблемы является *инверсия порядка блокировки*.

В подобных ситуациях поможет `scoped_lock`. Этот класс появился в C++17 и работает точно так же, как и классы `lock_guard` и `unique_lock`: его конструктор выполняет блокирование, а его деструктор разблокирует мьютекс. Класс может работать с *несколькими* мьютексами сразу.

Класс `scoped_lock` использует функцию `std::lock`, которая применяет особый алгоритм, выполняющий набор вызовов `try_lock` для всех предоставленных мьютексов, что позволяет предотвратить взаимные блокировки. Поэтому совершенно безопасно задействовать `scoped_lock` или вызывать `std::lock` для одного набора блокировок, но в разном порядке.

Синхронизация конкурентного использования std::cout

Многопоточные программы неудобны тем, что нужно охранять *каждую* структуру данных, которую они изменяют, с помощью мьютексов или других средств защиты от неуправляемых конкурентных изменений.

Одной из структур данных, часто применяемых для вывода данных, является `std::cout`. Если несколько потоков пытаются получить доступ к `cout` на конкурентной основе, то мы получим смешанные выходные данные. Чтобы это предотвратить, следует написать собственную функцию, которая выводит данные на экран и защищена от конкурентности.

Мы узнаем, как предоставить оболочку для `cout`, которая состоит из минимального объема кода и так же удобна в использовании, как и `cout`.

Как это делается

В этом примере мы реализуем программу, выводящую на экран данные на конкурентной основе из нескольких потоков. Чтобы предотвратить искажение сообщений из-за конкурентности, реализуем небольшой вспомогательный класс, который синхронизирует вывод данных между потоками.

1. Как и обычно, сначала укажем все директивы `include` и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <sstream>
#include <vector>

using namespace std;
```

2. Далее реализуем вспомогательный класс, который назовем `pcout`. Буква *p* означает *«параллельный»*, поскольку он синхронизирован для параллельных контекстов. Идея заключается в том, что `pcout` явно наследует от `stringstream`. Таким образом, можно применять `operator<<` для экземпляров этого класса. Как только экземпляр `pcout` уничтожается, его деструктор блокирует мьютекс, а затем выводит на экран содержимое буфера `stringstream`. На следующем шаге мы увидим, как это использовать.

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

3. Теперь напишем две функции, которые можно выполнить с помощью дополнительных потоков. Обе функции принимают в качестве аргументов идентификаторы потоков. Затем они отличаются только тем, что одна из них использует для вывода данных непосредственно `cout`. Вторая же выглядит практически так же, но вместо непосредственного применения `cout` создает экземпляр `pcout`. Этот экземпляр представляет собой временный объект, существующий только для одной строки кода. После выполнения всех вызовов `operator<<` внутренний строковый поток заполняется всеми данными, которые мы бы хотели вывести. Затем вызывается деструктор для экземпляра типа `pcout`. Мы уже видели, что он делает: блокирует конкретный мьютекс, разделяемый всеми экземплярами класса `pcout`, и выводит данные на экран:

```
static void print_cout(int id)
{
```

```
    cout << "cout hello from " << id << '\n';
}
static void print_pcout(int id)
{
     pcout{} << "pcout hello from " << id << '\n';
}
```

4. Опробуем его. Сначала воспользуемся `print_cout`, которая для вывода данных просто обращается к `cout`. Запускаем десять потоков, конкурентно выводящих свои строки и ожидающих завершения:

```
int main()
{
    vector<thread> v;
    for (size_t i {0}; i < 10; ++i) {
        v.emplace_back(print_cout, i);
    }
    for (auto &t : v) { t.join(); }
```

5. Далее делаем то же самое для функции `print_pcout`:

```
    cout << "====================\n";
    v.clear();
    for (size_t i {0}; i < 10; ++i) {
        v.emplace_back(print_pcout, i);
    }
    for (auto &t : v) { t.join(); }
}
```

6. Компиляция и запуск программы дадут следующий результат (рис. 9.3). Как видите, первые десять строк полностью перепутались. Именно так может выглядеть результат, когда `cout` используется конкурентно и без блокировки. Последние десять строк выведены с помощью `print_pcout`, никакой путаницы нет. Можно увидеть, что они выводятся из разных потоков, поскольку их порядок различается при каждом запуске программы.

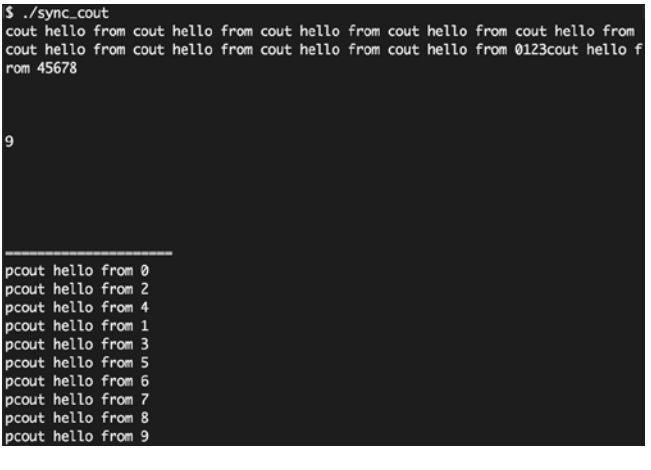

Рис. 9.3

Как это работает

О'кей, мы создали эту *«оболочку для cout»*, которая автоматически сериализует последовательные попытки вывода текста. Как она работает?

Выполним те же действия, что и `pcout`, вручную. Сначала создаем строковый поток и принимаем входные данные, которые будем передавать в него:

```
stringstream ss;
ss << "This is some printed line " << 123 << '\n';
```

Затем блокируем глобально доступный мьютекс:

```
{
    lock_guard<mutex> l {cout_mutex};
```

В этой заблокированной области видимости мы получаем доступ к содержимому строкового потока `ss`, выводим его на экран и освобождаем мьютекс, покидая область видимости. Строка `cout.flush()` указывает объекту потока вывести данные на консоль немедленно. Без данной строки программа способна работать быстрее, поскольку несколько строк можно вывести за один раз. В наших примерах мы хотим видеть результат работы немедленно, так что используем метод `flush`:

```
    cout << ss.rdbuf();
    cout.flush();
}
```

О'кей, это достаточно просто, но утомительно писать, если нужно делать одно и то же раз за разом. Можно сократить создание объекта `stringstream` таким образом:

```
stringstream{} << "This is some printed line " << 123 << '\n';
```

Эта строка создает объект строкового потока, передает ему все, что нужно вывести на экран, а затем снова разрушает его. Жизненный цикл строкового потока сокращается до данной строки. После этого мы не можем выводить на экран данные, поскольку у нас нет доступа к указанному объекту. Какой фрагмент кода последним может получить содержимое потока? Это деструктор `stringstream`.

Мы не можем изменить методы-члены экземпляра `stringstream`, но способны расширить их, обернув наш собственный тип вокруг них с помощью наследования:

```
struct pcout : public stringstream {
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

Этот класс *все еще* является строковым потоком, и его можно использовать так же, как и любой другой строковый поток. Единственное отличие заключается в том, что он будет блокировать мьютекс и выводить собственный буфер с помощью `cout`.

Кроме того, мы поместили объект `cout_mutex` в структуру `pcout` как статический экземпляр, и теперь все элементы находятся в одном месте.

Безопасно откладываем инициализацию с помощью std::call_once

Иногда встречаются специфические разделы кода, которые можно запустить в параллельном контексте в нескольких потоках, при этом перед выполнением самих функций нужно выполнить *программу настройки*. Простым решением будет выполнить существующую функцию настройки до того, как программа войдет в состояние, из которого время от времени может работать параллельный код.

Однако данный подход имеет следующие недостатки.

- Если параллельная функция находится в библиотеке, то пользователь не должен забывать вызывать функцию настройки. Это не упрощает применение библиотеки.
- Предположим, функция настройки в какой-то степени дорогая, и ее, возможно, даже не требуется выполнять, если параллельные функции, которые ее используют, не запускаются. В таком случае необходим код, определяющий, запускать эту функцию или нет.

В данном примере мы рассмотрим вспомогательную функцию `std::call_once`, которая решает эту проблему простым, элегантным и неявным способом.

Как это делается

В этом примере мы напишем программу, которая запускает несколько потоков, выполняющих один и тот же код. Несмотря на полностью одинаковый выполняемый ими код, наша функция настройки будет вызвана всего раз.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

using namespace std;
```

2. Далее будем использовать функцию `std::call_once`. Чтобы ее применить, нужен экземпляр типа `once_flag` для синхронизации между всеми потоками, которые задействуют `call_once` для конкретной функции.

```
once_flag callflag;
```

3. Функция, которая должна быть выполнена всего раз, выглядит так. Она просто выводит один восклицательный знак:

```
static void once_print()
{
```

```
    cout << '!';
}
```

4. Все потоки будут выполнять функцию `print`. Первое, что мы сделаем, — вызовем функцию `once_print` для функции `std::call_once`. Функции `call_once` требуется переменная `callflag`, которую мы определили ранее. Она послужит для управления потоками.

```
static void print(size_t x)
{
    std::call_once(callflag, once_print);
    cout << x;
}
```

5. О'кей, теперь запустим десять потоков, все они будут использовать функцию `print`:

```
int main()
{
    vector<thread> v;
    for (size_t i {0}; i < 10; ++i) {
        v.emplace_back(print, i);
    }
    for (auto &t : v) { t.join(); }
    cout << '\n';
}
```

6. Компиляция и запуск дадут следующий результат. Сначала мы увидим восклицательный знак благодаря функции `once_print`. Затем увидим все идентификаторы потоков. Функция `call_once` не только помогла убедиться в том, что функция `once_print` была вызвана всего раз. Помимо этого, она синхронизировала все потоки и ни один идентификатор не был выведен на экран *до* выполнения `once_print`.

```
$ ./call_once
!1239406758
```

Как это работает

Функция `std:call_once` работает как барьер. Она поддерживает доступ к функции (или вызываемому объекту). Первый поток, достигший ее, выполняет эту функцию. Пока ее выполнение не завершится, любой другой поток, который достигнет `call_once`, заблокируется. После того как первый поток вернется из этой функции, все другие потоки также будут освобождены.

Чтобы организовать этот небольшой «танцевальный номер», требуется переменная, на основе которой другие потоки могут определить, следует ли им ждать, а также время их освобождения. Именно для этого и предназначена переменная `once_flag callflag;`. Каждая строка `call_once` нуждается и в экземпляре типа `once_flag`. Он будет передан как аргумент перед функцией, которая должна быть вызвана всего раз.

Еще одна приятная деталь: если поток, который был выбран для выполнения функции `call_once`, *даст сбой* из-за какого-то *исключения*, то следующий поток сможет попытаться выполнить функцию снова. Это делается вследствие вероятности того, что в следующий раз исключение не будет сгенерировано.

Отправляем выполнение задач в фоновый режим с применением std::async

При необходимости выполнить некий код в фоновом режиме можно просто запустить новый поток, который выполнит данный код. В подобных ситуациях можно сделать что-то еще, а затем подождать результата. Это просто:

```
std::thread t {my_function, arg1, arg2, ...};
// сделать что-то еще
t.join(); // подождать завершения потока
```

Но здесь начинаются неудобства: `t.join()` не дает возвращаемое значение функции `my_function`. Чтобы получить его, следует написать функцию, которая вызывает функцию `my_function` и сохраняет ее возвращаемое значение в какой-то переменной. Последняя также доступна первому потоку, в котором и был запущен новый поток. Если такие ситуации происходят постоянно, то нужно написать очень много стереотипного кода снова и снова.

В C++11 появилась функция `std::async`, способная решить эту задачу для нас, и не только. В этом примере мы напишем простую программу, которая делает несколько дел одновременно с помощью асинхронных вызовов функций. Поскольку `std::async` эффективна не только в данной области, рассмотрим все ее аспекты.

Как это делается

В этом примере мы реализуем программу, которая делает несколько дел конкурентно, но вместо того, чтобы явно запускать потоки, мы используем `std::async` и `std::future`.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>
#include <future>

using namespace std;
```

2. Реализуем три функции, которые не связаны с параллелизмом, а просто выполняют интересные задачи. Первая функция принимает строку и создает гистограмму включения всех символов внутри этой строки:

```
static map<char, size_t> histogram(const string &s)
{
    map<char, size_t> m;
    for (char c : s) { m[c] += 1; }
    return m;
}
```

3. Вторая функция также принимает строку и возвращает ее отсортированную копию:

```
static string sorted(string s)
{
    sort(begin(s), end(s));
    return s;
}
```

4. Третья функция подсчитывает, как много гласных находится внутри принимаемой строки:

```
static bool is_vowel(char c)
{
    char vowels[] {"aeiou"};
    return end(vowels) !=
           find(begin(vowels), end(vowels), c);
}
static size_t vowels(const string &s)
{
    return count_if(begin(s), end(s), is_vowel);
}
```

5. В функции `main` считываем весь стандартный поток ввода в одну строку. Чтобы не разбивать входные данные на слова, деактивируем `ios::skipws`. Подобным образом получаем одну большую строку независимо от того, сколько пробелов содержится во входных данных. Используем `pop_back` для полученной строки, поскольку так мы получаем слишком много символов-терминаторов `'\0'`:

```
int main()
{
    cin.unsetf(ios::skipws);
    string input {istream_iterator<char>{cin}, {}};
    input.pop_back();
```

6. Теперь вернем значения из всех функций, которые реализовали ранее. Чтобы ускорить выполнение программы в случае получения очень больших входных данных, запустим их *асинхронно*. Функция `std::async` принимает политику, функцию и аргументы этой функции. Вызываем функции `histogram`, `sorted` и `vowels`, передавая в качестве политики `launch::async` (далее узнаем, что

это значит). Все функции получают в качестве аргументов одинаковые входные строки:

```
    auto hist        (async(launch::async,
                           histogram, input));
    auto sorted_str  (async(launch::async,
                           sorted,    input));
    auto vowel_count (async(launch::async,
                           vowels,    input));
```

7. Вызовы `async` возвращают значения мгновенно, поскольку не выполняют сами функции. Вместо этого они подготавливают структуры для синхронизации, которые в дальнейшем станут получать результаты вызова функций. Результаты теперь будут определяться конкурентно с помощью дополнительных потоков. В это же время мы можем делать все, что захотим, поскольку получим данные значения позже. Возвращаемые значения `hist`, `sorted_str` и `vowel_count` имеют типы, указанные для функций `histogram`, `sorted` и `vowels`, но они обернуты в тип `future` функцией `std::async`. Объекты этого типа выражают тот факт, что в какой-то момент времени будут содержать значения. Вызов `.get()` позволяет получить их все, а до их появления можем заблокировать функцию `main`. После получения этих значений выводим их на экран:

```
    for (const auto &[c, count] : hist.get()) {
        cout << c << ": " << count << '\n';
    }
    cout << "Sorted string: "
         << quoted(sorted_str.get()) << '\n'
         << "Total vowels: "
         << vowel_count.get()        << '\n';
}
```

8. Компиляция и запуск кода выглядят так. Мы использовали короткую строку, которую не стоит распараллеливать, но для примера выполнили ее конкурентно. Вдобавок общая структура программы не изменилась по сравнению с наивной последовательной реализацией:

```
$ echo "foo bar baz foobazinga" | ./async
 : 3
a: 4
b: 3
f: 2
g: 1
i: 1
n: 1
o: 4
r: 1
z: 2
Sorted string: "   aaaabbbffginoooorzz"
Total vowels: 9
```

Как это работает

Если бы мы не использовали `std::async`, то последовательный нераспараллеленный код выглядел бы довольно просто:

```
auto hist        (histogram(input));
auto sorted_str  (sorted(   input));
auto vowel_count (vowels(   input));

for (const auto &[c, count] : hist) {
    cout << c << ": " << count << '\n';
}
cout << "Sorted string: " << quoted(sorted_str) << '\n';
cout << "Total vowels: "  << vowel_count        << '\n';
```

Чтобы распараллелить код, мы сделали следующее: обернули три вызова функций в вызовы `async(launch::async, ...)`. Следовательно, указанные функции выполняются не в основном потоке. Вместо этого `async` запускает новые потоки и позволяет им выполнить функции конкурентно. Таким образом, мы имеем дело только с теми издержками, которые возникают при запуске другого потока, и можем продолжить выполнение следующих строк кода, а вся работа совершится в фоновом режиме:

```
auto hist        (async(launch::async, histogram, input));
auto sorted_str  (async(launch::async, sorted,    input));
auto vowel_count (async(launch::async, vowels,    input));

for (const auto &[c, count] : hist.get()) {
    cout << c << ": " << count << '\n';
}
cout << "Sorted string: "
     << quoted(sorted_str.get()) << '\n'
     << "Total vowels: "
     << vowel_count.get()        << '\n';
```

В то время как, например, функция `histogram` возвращает экземпляр типа `map`, вызов `async(..., histogram, ...)` возвращает ассоциативный массив, который обернут в объект типа `future`. Последний является чем-то вроде *заполнителя* до тех пор, пока поток, выполняющий функцию `histogram`, не вернет значение. Полученный ассоциативный массив помещается в объект типа `future`, и мы наконец можем получить к нему доступ. Функция `get` дает доступ к инкапсулированному результату.

Рассмотрим еще один короткий пример. Взгляните на этот фрагмент:

```
auto x (f(1, 2, 3));
cout << x;
```

Вместо предыдущего кода мы могли написать следующее:

```
auto x (async(launch::async, f, 1, 2, 3));
cout << x.get();
```

Вот, по сути, и все. Выполнение задач в фоновом режиме никогда не было проще по стандартам C++. Осталось разрешить один момент: что означает `launch::async`? Это флаг, который определяет политику запуска. Существуют два флага политики, соответственно, возможны три их сочетания (табл. 9.4).

Таблица 9.4

Выбор политики	Значение
launch::async	Гарантируется, что функция будет выполнена другим потоком
launch::deferred	Функция будет выполнена этим же потоком, но позже (ленивая оценка). Функция будет выполнена, когда для объекта future вызовется метод get или wait. Если этого не произойдет, то вызова функции не будет
launch::async | launch::deferred	Если установить оба флага, то реализация async, предоставляемая STL, самостоятельно выберет, какую политику использовать. Этот вариант является выбором по умолчанию

Вызов наподобие async(f, 1, 2, 3) без аргумента политики автоматически выберет обе политики. Реализация async сама выберет, какую политику использовать. Это означает отсутствие уверенности в том, что другой поток вообще запустится или что выполнение будет просто отложено в другом потоке.

Дополнительная информация

Следует рассмотреть последний момент. Предположим, мы пишем код следующим образом:

```
async(launch::async, f);
async(launch::async, g);
```

Это может привести к тому, что функции `f` и `g` (в данном примере неважны возвращаемые ими значения) будут выполняться в конкурирующих потоках и в это же время будут запускаться разные задачи. При запуске такого кода мы увидим *блокировку* кода при этих вызовах, что нам, вероятно, не требуется.

Почему же код блокируется? Разве `async` не должен отвечать за неблокируемость асинхронных вызовов? Да, это так, но есть одна особая тонкость: если объект типа `future` был получен из вызова `async`, имеющего политику `launch::async`, то его деструктор выполнит *блокирующее ожидание*.

Это значит, что *оба* вызова `async` из данного короткого примера являются блокирующими, поскольку сроки жизни объектов типа `future`, которые они возвращают, заканчиваются в одной строке! Можно исправить данную ситуацию, получив их возвращаемые значения и поместив в переменные с более длинным сроком жизни.

Реализуем идиому «производитель/потребитель» с использованием std::condition_variable

В этом примере мы реализуем типичную программу, работающую по принципу «производитель/потребитель», в которой запускается несколько потоков. Основная идея заключается в том, что существует один поток, который создает элементы и помещает их в очередь. Еще один поток потребляет (использует) эти элементы. Если создавать нечего, то поток-производитель приостанавливается. При отсутствии в очереди элементов приостанавливается потребитель.

Оба потока имеют доступ к очереди и могут изменить, поэтому ее нужно защитить с помощью мьютекса.

Важно рассмотреть и следующий момент: что будет делать потребитель, если в очереди нет элементов? Будет ли он опрашивать очередь каждую секунду до тех пор, пока не увидит новые элементы? В этом нет необходимости, поскольку можно позволить потребителю подождать *событий*, которые его пробудят, отправляемых производителем в момент, когда появляются новые элементы.

Для таких событий в C++11 предоставляется удобная структура данных: `std::condition_variable`. В этом примере мы реализуем простое приложение, работающее по принципу «производитель/потребитель», которое пользуется этими структурами.

Как это делается

Итак, в этом примере мы реализуем простую программу, работающую по принципу «производитель/потребитель», которая запускает одного производителя значений в отдельном потоке, а также одного потребителя в другом потоке.

1. Сначала выполним все необходимые директивы `include` и объявим об использовании пространства имен `std`:

```
#include <iostream>
#include <queue>
#include <tuple>
#include <condition_variable>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. Создадим экземпляр очереди простых численных значений и назовем ее `q`. Производитель будет помещать туда значения, а потребитель — доставать их оттуда. Для их синхронизации понадобится мьютекс. Вдобавок создадим экземпляр типа `condition_variable` и назовем его `cv`. Переменная `finished` представляет

собой способ, с помощью которого производитель укажет потребителю, что других значений больше не будет:

```
queue<size_t>      q;
mutex              mut;
condition_variable cv;
bool               finished {false};
```

3. Реализуем функцию-производитель. Она принимает аргумент `items`, который ограничивает максимальное количество создаваемых элементов. В простом цикле он будет приостанавливаться на 100 миллисекунд для каждого элемента, что симулирует некоторую вычислительную *сложность*. Затем заблокируем мьютекс, синхронизирующий доступ к очереди. После успешного создания и вставки элемента в очередь вызываем `cv.notify_all()`. Данная функция пробуждает потребителя. Далее мы увидим на стороне потребителя, как это работает.

```
static void producer(size_t items) {
    for (size_t i {0}; i < items; ++i) {
        this_thread::sleep_for(100ms);
        {
            lock_guard<mutex> lk {mut};
            q.push(i);
        }
        cv.notify_all();
    }
```

4. После создания всех элементов снова блокируем мьютекс, поскольку нужно задать значение для бита `finished`. Затем опять вызываем метод `cv.notify_all()`:

```
    {
        lock_guard<mutex> lk {mut};
        finished = true;
    }
    cv.notify_all();
}
```

5. Теперь можем реализовать функцию потребителя. Она не принимает никаких аргументов, поскольку будет слепо перебирать все элементы до тех пор, пока очередь не опустеет. В цикле, который станет выполняться до тех пор, пока не установится значение переменной `finished`, функция будет блокировать мьютекс, который защищает очередь и флаг `finished`. В момент получения блокировки последняя вызовет функцию `cv.wait`, передав ей блокировку и лямбда-выражение в качестве аргументов. Лямбда-выражение — это предикат, указывающий, жив ли еще поток-производитель и есть ли значения в очереди.

```
static void consumer() {
    while (!finished) {
        unique_lock<mutex> l {mut};
        cv.wait(l, [] { return !q.empty() || finished; });
```

6. Вызов `cv.wait` разблокирует блокировку и ждет до тех пор, пока условие, описанное предикатом, не перестанет выполняться. Затем функция снова блокирует мьютекс и перебирает все из очереди до тех пор, пока та не опустеет. Если производитель все еще жив, то она снова проитерирует по циклу. В противном случае функция завершит работу, поскольку установлен флаг `finished` — таким способом производитель указывает, что новых элементов больше не будет.

```
        while (!q.empty()) {
            cout << "Got " << q.front()
                 << " from queue.\n";
            q.pop();
        }
    }
}
```

7. В функции `main` запускаем поток-производитель, который создает десять элементов, а также поток-потребитель. Затем ждем их выполнения и завершаем программу.

```
int main() {
    thread t1 {producer, 10};
    thread t2 {consumer};
    t1.join();
    t2.join();
    cout << "finished!\n";
}
```

8. Компиляция и запуск программы дадут следующий результат. При выполнении программы можно увидеть, что между появлением каждой строки проходит какое-то время (100 миллисекунд), поскольку создание элементов занимает время:

```
$ ./producer_consumer
Got 0 from queue.
Got 1 from queue.
Got 2 from queue.
Got 3 from queue.
Got 4 from queue.
Got 5 from queue.
Got 6 from queue.
Got 7 from queue.
Got 8 from queue.
Got 9 from queue.
finished!
```

Как это работает

В данном примере мы просто запустили два потока. Первый создает элементы и помещает их в очередь. Второй извлекает их из очереди. При взаимодействии с очередью один из этих потоков блокирует общий мьютекс `mut`, доступный им обоим. Таким образом, можно быть уверенными, что оба потока не будут взаимодействовать с состоянием очереди в одно и то же время.

Помимо очереди и мьютекса мы объявили четыре переменные, которые были включены во взаимодействие производителя/потребителя:

```
queue<size_t>      q;
mutex              mut;
condition_variable cv;
bool               finished {false};
```

Переменную `finished` объяснить очень просто. Ее значение было установлено в `true`, когда производитель создал ограниченное количество элементов. Когда потребитель видит, что значение этой переменной равно `true`, он использует последние элементы и завершает работу. Но для чего нужна переменная `condition_variable cv`? Мы использовали `cv` в двух разных контекстах. Один из контекстов *ждал наступления конкретного условия*, а второй — *указывал на выполнение этого условия*.

Сторона-потребитель, ждущая выполнения конкретного условия, выглядит так. Поток-потребитель проходит в цикле по блоку, который изначально блокирует мьютекс `mut` с помощью `unique_lock`. Затем вызывает `cv.wait`:

```
while (!finished) {
    unique_lock<mutex> l {mut};

    cv.wait(l, [] { return !q.empty() || finished; });

    while (!q.empty()) {
        // consume
    }
}
```

Данный код чем-то похож на следующий эквивалентный код. Вскоре мы рассмотрим, почему эти фрагменты не похожи друг на друга:

```
while (!finished) {
    unique_lock<mutex> l {mut};

    while (q.empty() && !finished) {
        l.unlock();
        l.lock();
    }

    while (!q.empty()) {
        // consume
    }
}
```

Это значит, что сначала мы получили блокировку, а затем проверили, с каким сценарием сейчас работаем.

1. Есть ли элементы, которые можно использовать? В таком случае сохраняем блокировку, потребляем эти элементы, возвращаем блокировку и начинаем сначала.

2. В противном случае, если *элементов для потребления нет*, но производитель *все еще жив*, то возвращаем мьютекс с целью дать ему шанс добавить новые элементы в очередь. Далее снова пытаемся получить блокировку в надежде, что ситуация изменится и мы перейдем к ситуации 1.

Реальная причина, по которой строка `cv.wait` не эквивалентна конструкции `while (q.empty() && ...)`, заключается в том, что мы не можем просто пройти по циклу `l.unlock(); l.lock();`. Отсутствие активности потока-производителя в течение какого-то промежутка времени приводит к постоянным блокировкам/разблокировкам мьютекса, что не имеет смысла, поскольку мы впустую тратим циклы процессора.

Выражение наподобие `cv.wait(lock, predicate)` будет ждать до тех пор, пока вызов `predicate()` не вернет значение `true`. Но это не делается путем постоянной блокировки/разблокировки. Чтобы возобновить поток, который блокируется вызовом `wait` объекта `condition_variable`, другой поток должен вызывать методы `notify_one()` или `notify_all()` для одного объекта. Только тогда ожидающий поток будет возобновлен и проверит условие `predicate()`. (Последнее действительно и для нескольких потоков.)

Положительный момент таков: после вызова `wait`, проверившего предикат так, словно поступил *внезапный* сигнал к пробуждению, поток будет снова мгновенно приостановлен. Это значит, что мы не навредим рабочему потоку программы (но, возможно, пострадает производительность), если добавим в код слишком много вызовов `notify`.

На стороне производителя мы просто вызвали `cv.notify_all()` после того, как производитель добавил новый элемент в очередь, а также вслед за тем, как создал последний элемент и установил значение флага `finished`, равное `true`. Этого было достаточно для того, чтобы направить потребителя.

Реализуем идиому «несколько производителей/потребителей» с помощью std::condition_variable

Возьмем задачу из прошлого примера и усложним ее: создадим *несколько* производителей и *несколько* потребителей. Кроме того, укажем, что размер очереди не должен превышать определенное значение.

Таким образом, нужно приостанавливать не только потребителей, при отсутствии в очереди элементов, но и производителей, если элементов в ней *слишком много*.

Мы увидим, как решить эту проблему с помощью нескольких объектов типа `std::condition_variable`, а также воспользуемся ими несколько иным образом, нежели это было показано в предыдущем примере.

Как это делается

В данном примере мы реализуем программу, похожую на программу из предыдущего примера, но в этот раз у нас будет несколько производителей и несколько потребителей.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `chrono_literals`:

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

using namespace std;
using namespace chrono_literals;
```

2. Затем реализуем синхронизированную вспомогательную функцию для вывода сообщений на экран, показанную в другом примере этой главы, поскольку будем выводить множество сообщений на конкурентной основе:

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
    }
};
```

3. Все производители пишут значения в одну очередь, а все потребители также получают значения из нее. В дополнение к этой очереди нужен мьютекс, защищающий очередь и флаг, на основе которого можно сказать, что создание элементов будет приостановлено в какой-то момент:

```
queue<size_t> q;
mutex         q_mutex;
bool          production_stopped {false};
```

4. В этой программе задействуем две разные переменные `condition_variables`. В предыдущем примере у нас была переменная `condition_variable`, которая указывала на появление в очереди новых элементов. В этом случае ситуация чуть более запутанная. Мы хотим, чтобы производители создавали новые элементы до тех пор, пока в очереди не будет содержаться их *определенное количество*. Если мы его достигли, то они должны *приостановиться*. Таким образом, переменная `go_consume` пригодна для возобновления потребителей, которые,

в свою очередь, смогут возобновить производителей с помощью переменной `go_produce`:

```
condition_variable go_produce;
condition_variable go_consume;
```

5. Функция-производитель принимает идентификатор производителя, общее количество элементов, которые нужно создать, а также максимальное количество элементов в очереди. Затем она входит в собственный производственный цикл. Далее блокирует мьютекс очереди, а затем разблокирует его снова в вызове `go_produce.wait`. Функция ожидает выполнения условия, согласно которому размер очереди должен быть меньше порогового значения `stock`:

```
static void producer(size_t id, size_t items, size_t stock)
{
    for (size_t i = 0; i < items; ++i) {
        unique_lock<mutex> lock(q_mutex);
        go_produce.wait(lock,
            [&] { return q.size() < stock; });
```

6. После того как производитель будет возобновлен, он создаст элемент и поместит его в очередь. Значение, помещаемое в очередь, определяется на основе выражения `id * 100 + i`. Таким образом, мы впоследствии можем увидеть, какой производитель создал его, поскольку количество сотен показывает идентификатор производителя. Кроме того, выводим сообщение о создании элемента в терминал. Формат этого сообщения может выглядеть странно, но оно будет выравнено в соответствии с сообщениями в окне консоли:

```
        q.push(id * 100 + i);
        pcout{} << "   Producer " << id << " --> item "
                << setw(3) << q.back() << '\n';
```

7. После создания элемента можно возобновить приостановленных потребителей. Период приостановки, равный 90 миллисекундам, симулирует тот факт, что на создание элементов требуется какое-то время:

```
        go_consume.notify_all();
        this_thread::sleep_for(90ms);
    }
    pcout{} << "EXIT: Producer " << id << '\n';
}
```

8. Теперь перейдем к функции-потребителю, которая принимает в качестве аргумента только идентификатор. Она продолжает ожидать элементов при условии, что их производство не остановлено или очередь не пуста. Если очередь пуста, но производство не остановлено, то, возможно, скоро появятся новые элементы:

```
static void consumer(size_t id)
{
    while (!production_stopped || !q.empty()) {
        unique_lock<mutex> lock(q_mutex);
```

9. После блокирования мьютекса очереди снова разблокируем его, чтобы подождать установки значения переменной события `go_consume`. Аргумент лямбда-выражения описывает, что нужно вернуть из вызова функции `wait`, когда очередь содержит элементы. Второй аргумент `1s` указывает, что мы не должны ждать вечно. Если мы ждем больше секунды, то хотим выйти из функции `wait`. Можно определить, вернула ли функция `wait_for` значение (условие-предикат будет верным) или мы вышли из нее по тайм-ауту (в этом случае возвратится значение `false`). При наличии в очереди новых элементов используем (потребим) их и выведем соответствующее сообщение на консоль:

```
        if (go_consume.wait_for(lock, 1s,
                [] { return !q.empty(); })) {
            pcout{} << "                  item "
                    << setw(3) << q.front()
                    << " --> Consumer "
                    << id << '\n';
            q.pop();
```

10. После потребления элемента оповещаем производителей и приостанавливаем поток на 130 миллисекунд для симуляции того факта, что потребление элементов тоже требует времени:

```
            go_produce.notify_all();
            this_thread::sleep_for(130ms);
        }
    }
    pcout{} << "EXIT: Producer " << id << '\n';
}
```

11. В функции `main` создаем один экземпляр вектора для рабочих потоков и еще один — для потоков-потребителей:

```
int main()
{
    vector<thread> workers;
    vector<thread> consumers;
```

12. Далее порождаем три потока-производителя и пять потоков-потребителей:

```
    for (size_t i = 0; i < 3; ++i) {
        workers.emplace_back(producer, i, 15, 5);
    }
    for (size_t i = 0; i < 5; ++i) {
        consumers.emplace_back(consumer, i);
    }
```

13. Сначала позволим закончить работу потокам-производителям. Как только все из них вернут значения, установим значение флага `production_stopped`; это приведет к тому, что потребители также закончат свою работу. Нужно собрать их, а затем завершить программу:

```
    for (auto &t : workers)   { t.join(); }
    production_stopped = true;
```

```
    for (auto &t : consumers) { t.join(); }
}
```

14. Компиляция и запуск программы дадут следующий результат. Сообщений получилось довольно много, поэтому мы приводим их в сокращенном виде. Как видите, производители приостанавливаются время от времени и позволяют потребителям использовать несколько элементов, чтобы снова получить возможность их производить. Интересно будет изменить время ожидания для производителей/потребителей, а также манипулировать количеством производителей/потребителей и максимальным количеством элементов в очереди, поскольку это значительно меняет шаблоны появления выходных сообщений:

```
$ ./multi_producer_consumer
   Producer 0 --> item   0
   Producer 1 --> item 100
                  item   0 --> Consumer 0
   Producer 2 --> item 200
                  item 100 --> Consumer 1
                  item 200 --> Consumer 2
   Producer 0 --> item   1
   Producer 1 --> item 101
                  item   1 --> Consumer 0
...
   Producer 0 --> item  14
EXIT: Producer 0
   Producer 1 --> item 114
EXIT: Producer 1
                  item  14 --> Consumer 0
   Producer 2 --> item 214
EXIT: Producer 2
                  item 114 --> Consumer 1
                  item 214 --> Consumer 2
EXIT: Consumer 2
EXIT: Consumer 3
EXIT: Consumer 4
EXIT: Consumer 0
EXIT: Consumer 1
```

Как это работает

Этот пример дополняет предыдущий. Вместо того чтобы синхронизировать одного производителя с одним потребителем, мы реализовали программу, которая синхронизирует `M` производителей и `N` потребителей. Кроме того, приостанавливаются не только потребители при отсутствии элементов для них, но и производители, если очередь становится *слишком длинной*. Когда несколько потребителей ждут заполнения одной очереди, они будут действовать по принципу, работающему и для сценария «один производитель — один потребитель». Пока только один поток блокирует мьютекс, защищающий очередь, а затем извлекает оттуда элементы,

код безопасен. Неважно, как много потоков ожидают блокировки одновременно. Это же верно и для производителя, поскольку единственный важный аспект в обоих сценариях таков: к очереди одномоментно может получить доступ только один поток, и не больше.

Более сложной, чем предыдущий пример, в котором запускались всего один производитель и один потребитель, эту программу делает тот факт, что мы указываем потокам-производителям останавливаться, когда длина очереди превышает какое-то значение. Чтобы соответствовать этому требованию, мы реализовали два разных сигнала, имеющих собственную переменную `condition_variable`.

1. `go_produce` сигнализирует о том, что очередь снова заполнена не до конца и производители могут опять начать ее заполнять.
2. `go_consume` уведомляет о достижении очереди максимального размера и о том, что потребители снова могут свободно использовать элементы.

Таким образом производители заполняют очередь элементами и сигнализируют с помощью события `go_consume` потокам-потребителям, которые ожидают на следующей строке:

```
if (go_consume.wait_for(lock, 1s, [] { return !q.empty(); })) {
    // получили событие без тайм-аута
}
```

Производители, с другой стороны, ждут на следующей строке до тех пор, пока не смогут создавать элементы снова:

```
go_produce.wait(lock, [&] { return q.size() < stock; });
```

Интересный момент: мы не позволяем потребителям ждать *вечно*. В вызове `go_consume.wait_for` добавляем дополнительный аргумент `timeout`, имеющий значение, равное 1 секунде. Он представляет собой механизм выхода для потребителей: если очередь пуста более секунды, то, возможно, активных производителей больше не осталось.

Для простоты код пытается поддерживать длину очереди *всегда на максимуме*. Более сложная программа могла бы позволить потокам отправлять уведомления о пробуждении *только* в том случае, если очередь достигнет *половины* максимального размера. Таким образом, производители будут пробуждаться до того, как очередь опустеет, но не раньше, когда в очереди все еще хватает элементов.

Рассмотрим следующую ситуацию, которую позволяет элегантно решить `condition_variable`: если потребитель отправляет уведомление `go_produce`, то, возможно, множество производителей пытаются перегнать друг друга в попытке создать новый элемент. При нехватке только одного элемента работать будет только один производитель. Если все производители всегда станут создавать элемент при появлении события `go_produce`, то мы зачастую будем сталкиваться с ситуацией, когда очередь заполняется сверх своего максимального размера.

Представим ситуацию, когда у нас в очереди имеется `(max - 1)` элементов и нужно создать один новый элемент, чтобы очередь снова стала заполненной.

Независимо от того, какой метод вызовет поток-потребитель — `go_produce.notify_one()` (возобновит только один ожидающий поток) или `go_produce.notify_all()` (возобновит *все* ожидающие потоки), — можно гарантировать, что только один поток-производитель завершит вызов `go_produce.wait`, поскольку для остальных потоков-производителей не будет удовлетворяться условие ожидания `q.size() < stock` в момент получения ими мьютекса при пробуждении.

Распараллеливание отрисовщика множества Мандельброта в ASCII с применением std::async

Помните *отрисовщик множества Мандельброта в ASCII* из главы 6? В этом примере мы воспользуемся потоками, чтобы немного сократить время его вычисления.

Сначала изменим строку оригинальной программы, которая ограничивает количество итераций для каждой выбранной координаты. Это сделает программу *медленнее*, а результаты — *более точными* в сравнении с той точностью, которая доступна при выводе данных на консоли, но у нас будет хороший пример программы для параллелизации.

Далее мы применим минимальные модификации к программе и увидим, что вся программа работает быстрее. После применения этих модификаций программа будет работать с `std::async` и `std::future`. Чтобы полностью уяснить данный пример, очень важно понять оригинальную программу.

Как это делается

В этом примере мы возьмем отрисовщик фрактала Мандельброта, который реализовали в главе 6. Сначала увеличим время вычисления, повысив границу вычислений. Затем ускорим программу, внеся четыре небольших изменения, чтобы распараллелить ее.

1. Чтобы следовать шагам, лучше всего скопировать всю программу из другого примера. Затем следуйте инструкциям, показанным в следующих шагах, для внесения всех необходимых изменений. Все отличия от оригинальной программы выделяются *полужирным* шрифтом.

   Первое изменение — это дополнительный заголовочный файл `<future>`:

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <complex>
#include <numeric>
#include <vector>
#include <future>

using namespace std;
```

2. Функции `scaler` и `scaled_cmplx` менять не нужно:

```
using cmplx = complex<double>;
static auto scaler(int min_from, int max_from,
                   double min_to, double max_to)
{
    const int w_from {max_from - min_from};
    const double w_to {max_to - min_to};
    const int mid_from {(max_from - min_from) / 2 + min_from};
    const double mid_to {(max_to - min_to) / 2.0 + min_to};
    return [=] (int from) {
        return double(from - mid_from) / w_from * w_to + mid_to;
    };
}
template <typename A, typename B>
static auto scaled_cmplx(A scaler_x, B scaler_y)
{
    return [=](int x, int y) {
        return cmplx{scaler_x(x), scaler_y(y)};
    };
}
```

3. В функции `mandelbrot_iterations` просто увеличим количество итераций, чтобы программа выполняла больше вычислений:

```
static auto mandelbrot_iterations(cmplx c)
{
    cmplx z {};
    size_t iterations {0};
    const size_t max_iterations {100000};
    while (abs(z) < 2 && iterations < max_iterations) {
        ++iterations;
        z = pow(z, 2) + c;
    }
    return iterations;
}
```

4. Далее перед нами часть функции `main`, которую также не нужно изменять:

```
int main()
{
    const size_t w {100};
    const size_t h {40};
    auto scale (scaled_cmplx(
        scaler(0, w, -2.0, 1.0),
        scaler(0, h, -1.0, 1.0)
    ));
    auto i_to_xy ([=](int x) {
        return scale(x % w, x / w);
    });
```

5. В функции `to_iteration_count` больше не вызываем `mandelbrot_iterations(x_to_xy(x))` непосредственно, этот вызов делается асинхронно с помощью `std::async`:

```
auto to_iteration_count ([=](int x) {
    return async(launch::async,
                 mandelbrot_iterations, i_to_xy(x));
});
```

6. Перед внесением последнего изменения функция `to_iteration_count` возвращала количество итераций, нужных конкретной координате, чтобы алгоритм Мандельброта сошелся. Теперь она возвращает переменную типа `future`, которая станет содержать то же значение позднее, когда оно будет рассчитано асинхронно. Из-за этого требуется вектор для хранения всех значений типа `future`, добавим его. Выходной итератор, который мы предоставили функции `transform` в качестве третьего аргумента, должен быть начальным итератором для нового вектора выходных данных `r`:

```
vector<int> v (w * h);
vector<future<size_t>> r (w * h);
iota(begin(v), end(v), 0);
transform(begin(v), end(v), begin(r),
          to_iteration_count);
```

7. Вызов `accumulate`, который выводит результат на экран, больше не получает значения типа `size_t` в качестве второго аргумента, вместо этого он получает значения типа `future<size_t>`. Нужно адаптировать его к данному типу (если бы мы изначально использовали тип `auto&`, то этого бы не требовалось), а затем вызвать `x.get()`, где мы получили доступ к `x`, чтобы дождаться появления значения.

```
    auto binfunc ([w, n{0}] (auto output_it, future<size_t> &x)
            mutable {
        *++output_it = (x.get() > 50 ? '*' : ' ');
        if (++n % w == 0) { ++output_it = '\n'; }
        return output_it;
    });
    accumulate(begin(r), end(r),
               ostream_iterator<char>{cout}, binfunc);
}
```

8. Компиляция и запуск программы дадут результат, который мы видели ранее. Увеличение количества итераций и для оригинальной версии приведет к тому, что распараллеленная версия отработает быстрее. На моем компьютере, имеющем четыре ядра ЦП, поддерживающих гиперпараллельность (что дает восемь виртуальных ядер), я получаю разные результаты для GCC и clang. В лучшем случае программа ускоряется в 5,3 раза, а в худшем — в 3,8. Результаты, конечно, будут различаться для разных машин.

Как это работает

Сначала важно понять всю программу, поскольку далее становится ясно, что вся работа, зависящая от ЦП, происходит в одной строке кода в функции `main`:

```
transform(begin(v), end(v), begin(r), to_iteration_count);
```

Вектор `v` содержит все индексы, соотнесенные с комплексными координатами, по которым мы итерируем в алгоритме Мандельброта. Результат каждой итерации сохраняется в векторе `r`.

В оригинальной программе в данной строке тратится все время, требуемое для расчета фрактального изображения. Весь код, находящийся перед ним, выполняет подготовку, а весь код, следующий за ним, нужен лишь для вывода информации на экран. Это значит, что распараллеливание выполнения этой строки критически важно для производительности.

Одним из возможных подходов к распараллеливанию является разбиение всего линейного диапазона от `begin(v)` до `end(v)` на фрагменты одного размера и равномерное их распределение между ядрами. Таким образом, все ядра выполнят одинаковый объем работы. Если бы мы использовали параллельную версию функции `std::transform` с параллельной политикой выполнения, то все бы так и произошло. К сожалению, это неверная стратегия для *нашей* задачи, поскольку каждая точка множества Мандельброта показывает индивидуальное количество итераций.

Наш подход заключается в том, что мы заполним вектор, в котором содержатся отдельные символы, элементами `future`, чьи значения будут высчитаны асинхронно. Поскольку вектор-источник и вектор-приемник содержат `w * h` элементов, что для нашего случая означает `100 * 40`, у нас есть вектор, содержащий 4000 значений типа `future`, которые высчитываются асинхронно. Если бы наша система имела 4000 ядер ЦП, то это означало бы запуск 4000 потоков, которые выполнили бы перебор множества Мандельброта действительно конкурентно. В обычной системе, имеющей меньшее количество ядер, ЦП будут обрабатывать один асинхронный элемент за другим.

В то время как вызов `transform` с асинхронной версией `to_iteration_count` сам по себе *не выполняет расчеты*, а лишь подготавливает потоки и объекты типа `future`, он возвращает значение практически мгновенно. Оригинальная версия программы к этому моменту будет заблокирована, поскольку итерации занимают слишком много времени.

Распараллеленная версия программы *также* может быть заблокирована. Функция, которая выводит на экран все наши значения в консоли, должна получать доступ к результатам, находящимся внутри объектов типа `future`. Чтобы это сделать, она вызывает функцию `x.get()` для всех значений. Идея заключается в следующем: пока она ждет вывода первого значения, множество других значений высчитываются одновременно. Поэтому, если метод `get()` для первого значения типа `future` возвращается, то следующий объект типа `future` тоже может быть готов к выводу на экран!

Если размер вектора будет гораздо больше, то возникнет некоторая измеряемая задержка при создании и синхронизации всех этих значений. В нашем случае задержка не так велика. На моем ноутбуке с процессором Intel i7, имеющем четыре ядра, которые могут работать в режиме *гиперпараллельности* (что дает восемь виртуальных ядер), параллельная версия этой программы работала в 3–5 раз быстрее оригинальной программы. Идеальное распараллеливание сделало бы программу в восемь раз быстрее. Конечно, это ускорение будет различаться для разных компьютеров, поскольку зависит от множества факторов.

Небольшая автоматическая библиотека для распараллеливания с использованием std::future

Большую часть сложных задач можно разбить на подзадачи. Из всех подзадач можно построить *направленный ациклических граф* (directed acyclic graph, DAG), который описывает, какие подзадачи зависят друг от друга, чтобы выполнить задачу более высокого уровня. Для примера предположим, что хотим создать строку `"foo bar foo bar this that"`, и можем сделать это только путем создания отдельных слов и их объединения с другими словами или с самими собой. Предположим, что этот механихм предоставляется тремя примитивными функциями `create`, `concat` и `twice`.

Принимая это во внимание, можно нарисовать следующий DAG, который визуализирует зависимости между ними, что позволяет получить итоговый результат (рис. 9.4).

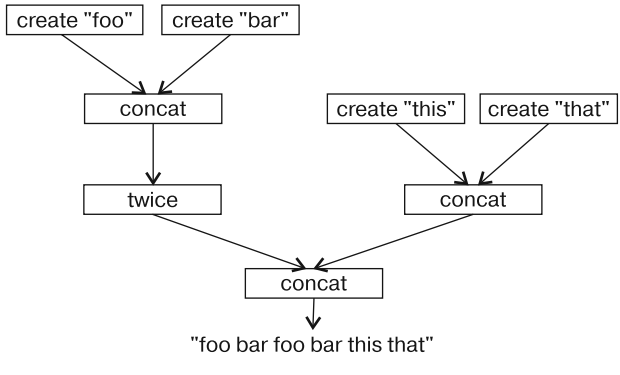

Рис. 9.4

При реализации данной задачи в коде понятно, что все можно реализовать последовательно на одном ядре ЦП. Помимо этого, все подзадачи, которые не зависят от других подзадач или зависят от уже завершенных, могут быть выполнены *конкурентно* на нескольких ядрах ЦП.

Может показаться утомительным писать подобный код, даже с помощью `std::async`, поскольку зависимости между подзадачами нужно смоделировать. В этом примере мы реализуем две небольшие вспомогательные функции, которые позволяют преобразовать нормальные функции `create`, `concat` и `twice` в функции, работающие асинхронно. С их помощью мы найдем действительно элегантный способ создать граф зависимостей. Во время выполнения программы граф сам себя распараллелит, чтобы максимально быстро получить результат.

Как это делается

В этом примере мы реализуем некие функции, которые симулируют сложные для вычисления задачи, зависящие друг от друга, и попробуем максимально их распараллелить.

1. Сначала включим все необходимые заголовочные файлы и объявим пространство имен `std`:

```
#include <iostream>
#include <iomanip>
#include <thread>
#include <string>
#include <sstream>
#include <future>

using namespace std;
using namespace chrono_literals;
```

2. Нужно синхронизировать конкурентный доступ к `cout`, поэтому задействуем вспомогательный класс, который написали в предыдущей главе:

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

3. Теперь реализуем три функции, которые преобразуют строки. Первая функция создаст объект типа `std::string` на основе строки, созданной в стиле C. Мы приостановим его на 3 секунды, чтобы симулировать сложность создания строки:

```
static string create(const char *s)
{
    pcout{} << "3s CREATE " << quoted(s) << '\n';
    this_thread::sleep_for(3s);
    return {s};
}
```

4. Следующая функция принимает два строковых объекта и возвращает их сконкатенированный вариант. Мы будем приостанавливать ее на 5 секунд, чтобы симулировать сложность выполнения этой задачи:

```
static string concat(const string &a, const string &b)
{
    pcout{} << "5s CONCAT "
            << quoted(a) << " "
            << quoted(b) << '\n';
    this_thread::sleep_for(5s);
    return a + b;
}
```

5. Последняя функция, наполненная вычислениями, принимает строку и конкатенирует ее с самой собой. На это потребуется 3 секунды:

```
static string twice(const string &s)
{
    pcout{} << "3s TWICE " << quoted(s) << '\n';
    this_thread::sleep_for(3s);
    return s + s;
}
```

6. Теперь можно использовать эти функции в последовательной программе, но мы же хотим элегантно ее распараллелить! Так что реализуем некоторые вспомогательные функции. *Будьте внимательны,* следующие три функции выглядят действительно сложными. Функция `asynchronize` принимает функцию `f` и возвращает вызываемый объект, который захватывает ее. Можно вызвать данный объект, передав ему любое количество аргументов, и он захватит их вместе с функцией `f` в другой вызываемый объект, который будет возвращен. Этот последний вызываемый объект может быть вызван без аргументов. Затем он вызывает функцию `f` асинхронно со всеми захваченными им аргументами:

```
template <typename F>
static auto asynchronize(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async, f, xs...);
        };
    };
}
```

7. Следующая функция будет использоваться функцией, которую мы объявим на шаге 8. Она принимает функцию `f` и захватывает ее в вызываемый объект; его и возвращает. Данный объект можно вызвать с несколькими объектами типа `future`. Затем он вызовет функцию `.get()` для всех этих объектов, применит к ним функцию `f` и вернет результат:

```
template <typename F>
static auto fut_unwrap(F f)
```

```
{
    return [f](auto ... xs) {
        return f(xs.get()...);
    };
}
```

8. Последняя вспомогательная функция также принимает функцию `f`. Она возвращает вызываемый объект, который захватывает `f`. Такой вызываемый объект может быть вызван с любым количеством аргументов, представляющих собой вызываемые объекты, которые он возвращает вместе с `f` в другом вызываемом объекте. Этот итоговый вызываемый объект можно вызвать без аргументов. Он вызывает все вызываемые объекты, захваченные в наборе `xs...`. Они возвращают объекты типа `future`, которые нужно распаковать с помощью функции `fut_unwrap`. Распаковка объектов типа `future` и применение самой функции `f` для реальных значений, находящихся в объектах типа `future`, происходит асинхронно с помощью `std::async`:

```
template <typename F>
static auto async_adapter(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async,
                         fut_unwrap(f), xs()...);
        };
    };
}
```

9. О'кей, возможно, предыдущие фрагменты кода были несколько запутанными и напоминали фильм *«Начало»* из-за лямбда-выражений, возвращающих лямбда-выражения. Мы подробно рассмотрим этот вуду-код далее. Теперь возьмем функции `create`, `concat` и `twice` и сделаем их асинхронными. Функция `async_adapter` заставляет обычную функцию ожидать получения аргументов типа `future` и возвращает в качестве результата объект типа `future`. Она похожа на оболочку, преобразующую синхронный мир в асинхронный. Необходимо использовать функцию `asynchronize` для функции `create`, поскольку она будет возвращать объект типа `future`, но следует передать ей реальные значения. Цепочка зависимостей для задач должна начинаться с вызовов `create`:

```
int main()
{
    auto pcreate (asynchronize(create));
    auto pconcat (async_adapter(concat));
    auto ptwice  (async_adapter(twice));
```

10. Теперь у нас есть функции, которые распараллеливаются автоматически и имеют такие же имена, что и оригинальные функции, но с префиксом `p`. Далее создадим сложное дерево зависимостей. Сначала добавим строки `"foo "` и `"bar "`, которые мгновенно сконкатенируем в `"foo bar "`. Эта строка будет сконкатенирована сама с собой с помощью функции `twice`. Затем создадим строки `"this "`

и "that ", которые сконкатенируем в "this that ". Наконец, сконкатенируем все эти строки в "foo bar foo bar this that ". Результат будет сохранен в переменной callable. Затем, наконец, вызовем функцию callable().get() с целью начать вычисления и дождаться возвращаемых значений, чтобы вывести на экран и их. До вызова callable() не выполняется никаких вычислений, а после этого вызова и начинается вся магия.

```
    auto result (
        pconcat(
            ptwice(
                pconcat(
                    pcreate("foo "),
                    pcreate("bar "))),
            pconcat(
                pcreate("this "),
                pcreate("that "))));
    cout << "Setup done. Nothing executed yet.\n";
    cout << result().get() << '\n';
}
```

11. Компиляция и запуск программы показывают, что все вызовы create выполняются одновременно, а остальные вызовы — уже после них. Кажется, будто все они были спланированы интеллектуально. Вся программа работает 16 секунд. Если бы шаги выполнялись не параллельно, то для завершения программы потребовалось бы 30 секунд. Обратите внимание: для одновременного выполнения всех вызовов create нужна система как минимум с четырьмя ядрами ЦП. Если у системы будет меньше ядер, то некоторые вызовы должны будут делить ЦП, что увеличит время выполнения программы.

```
$ ./chains
Setup done. Nothing executed yet.
3s CREATE "foo "
3s CREATE "bar "
3s CREATE "this "
3s CREATE "that "
5s CONCAT "this " "that "
5s CONCAT "foo " "bar "
3s TWICE  "foo bar "
5s CONCAT "foo bar foo bar " "this that "
foo bar foo bar this that
```

Как это работает

Простая последовательная версия этой программы без вызовов async и объектов типа future выглядела бы так:

```
int main()
{
    string result {
        concat(
```

```
            twice(
                concat(
                    create("foo "),
                    create("bar "))),
            concat(
                create("this "),
                create("that "))) };
    cout << result << '\n';
}
```

В данном примере мы написали вспомогательные функции `async_adapter` и `asynchronize`, которые позволили создать новые функции на основе функций `create`, `concat` и `twice`. Мы назвали эти новые асинхронные функции `pcreate`, `pconcat` и `ptwice`. Сначала опустим сложность реализации `async_adapter` и `asynchronize` с целью увидеть, что они дают. Последовательная версия выглядит аналогично следующему коду:

```
string result {concat( ... )};
cout << result << '\n';
```

Распараллеленная версия выглядит аналогично этому фрагменту:

```
auto result (pconcat( ... ));
cout << result().get() << '\n';
```

Теперь перейдем к сложной части. Типом распараллеленного результата является не `string`, а вызываемый объект, возвращающий объект типа `future<string>`, для которого можно вызвать функцию `get()`. На первый взгляд это выглядит безумным.

Как и *зачем* мы работаем с объектами, которые возвращают значения типа `future`? Проблема заключается в том, что наши методы `create`, `concat` и `twice` *слишком медленные*. (Да, мы искусственно их замедлили, поскольку пытались смоделировать реальные приложения, которые потребляют много времени ЦП.) Но мы определили, что дерево зависимостей, описывающее поток данных, имеет независимые части, пригодные для параллельного выполнения. Рассмотрим два примера планов (рис. 9.5).

С левой стороны мы видим план для *одного ядра*. Все вызовы функций нужно выполнять один за другим, поскольку у нас есть только один ЦП. Это значит, что, поскольку вызов функции `create` длится 3 секунды, вызов `concat` — 5 секунд, а `twice` — 3 секунды, для получения конечного результата потребуется 30 секунд.

С правой стороны мы видим *план*, где задачи выполняются максимально распараллеленно. В идеальном мире, где все компьютеры имеют четыре ядра, можно создать все подстроки одновременно, а затем сконкатенировать их. Минимальное время получения результата с оптимальным параллельным планом равно 16 секундам. Мы не можем ускорить выполнение программы, если не сделать сами вызовы функций быстрее. Имея всего четыре ядра ЦП, можно добиться этого времени выполнения. Мы достигли оптимального расписания. Как оно работает?

Одно ядро
Время
Ядро 0
3s
create "foo"
3s
create "bar"
5s
concat
3s
twice
3s
create "this"
3s
create "that"
5s
concat
5s
concat
"foo bar foo bar this that"
30s
Несколько ядер
Время
Ядро 0
Ядро 1
Ядро 2
Ядро 3
3s
create "foo"
create "bar"
create "this"
create "that"
5s
concat
concat
3s
twice
5s
concat
"foo bar foo bar this that"
16 s

Рис. 9.5

Мы могли бы просто написать следующий код:

```
auto a (async(launch::async, create, "foo "));
auto b (async(launch::async, create, "bar "));
auto c (async(launch::async, create, "this "));
auto d (async(launch::async, create, "that "));
auto e (async(launch::async, concat, a.get(), b.get()));
auto f (async(launch::async, concat, c.get(), d.get()));
auto g (async(launch::async, twice, e.get()));
auto h (async(launch::async, concat, g.get(), f.get()));
```

Это хорошее начало для `a`, `b`, `c` и `d`, которые представляют четыре подстроки. Они создаются асинхронно в фоновом режиме. К сожалению, этот код блокируется в строке, где мы инициализируем `e`. Чтобы сконкатенировать `a` и `b`, нужно вызвать `get()` для обеих подстрок, данный код будет *заблокирован* до тех пор, пока данные значения не будут *готовы*. Очевидно, это плохая идея, поскольку распараелелливание перестает быть паралелльным после первого вызова `get()`. Требуется более хорошая стратегия.

Задействуем сложные вспомогательные функции, которые мы написали. Первая из них — это `asynchronize`:

```
template <typename F>
static auto asynchronize(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async, f, xs...);
        };
    };
}
```

При наличии функции `int f(int, int)` можно сделать следующее:

```
auto f2 ( asynchronize(f) );
auto f3 ( f2(1, 2) );
auto f4 ( f3() );
int result { f4.get() };
```

Функция `f2` — это наша асинхронная версия функции `f`. Ее можно вызвать с теми же аргументами, что и функцию `f`, поскольку `f2` *подражает* ей. Затем она возвращает вызываемый объект, который мы сохраняем в `f3`. Функция `f3` теперь захватывает `f` и аргументы `1, 2`, но пока ничего не вызывает. Это все делается ради захвата.

Теперь при вызове функции `f3()` мы наконец получаем объект типа `future`, поскольку `f3()` делает вызов `async(launch::async, f, 1, 2);`! В некотором смысле семантическое значение `f3` заключается в следующем: *«Получить захваченную функцию и аргументы, а затем передать их в вызов* `std::async`*»*.

Внутреннее лямбда-выражение, которое не принимает никаких аргументов, позволяет пойти по нестандартному пути. С его помощью можно настроить работу для параллельной отправки, но не нужно вызывать никаких блокирующих функций. Мы следуем тому же принципу в гораздо более сложной функции `async_adapter`:

```
template <typename F>
static auto async_adapter(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async, fut_unwrap(f), xs()...);
        };
    };
}
```

Данная функция также сначала возвращает функцию, которая подражает `f`, поскольку принимает те же аргументы. Затем эта функция возвращает вызываемый объект, который тоже не принимает аргументов. В результате упомянутый вызываемый объект наконец отличается от другой вспомогательной функции.

Каково значение строки `async(launch::async, fut_unwrap(f), xs()...);`? Часть `xs()...` означает предположение, что все аргументы, которые сохраняются в наборе параметров `xs`, являются вызываемыми объектами (как те, что мы постоянно создаем!) и, как следствие, вызываются без аргументов. Эти вызываемые объекты, постоянно создаваемые нами, производят значения типа `future`, для которых мы вызываем функцию `get()`. Здесь вступает в действие функция `fut_unwrap`:

```
template <typename F>
static auto fut_unwrap(F f)
{
    return [f](auto ... xs) {
        return f(xs.get()...);
    };
}
```

Функция `fut_unwrap` просто преобразует функцию `f` в объект функции, который принимает диапазон аргументов. Данный объект вызывает функцию `.get()` для них *всех* и наконец перенаправляет их к `f`.

Возможно, вам потребуется время, чтобы это переварить. В нашей функции `main` цепочка вызовов `auto result (pconcat(...));` создавала большой вызываемый объект, который содержал все функции и все аргументы. К этому моменту мы не выполняли асинхронных вызовов. Затем, вызвав функцию `result()`, мы *породили небольшую лавину* вызовов `async` и `.get()`, которые выполнялись в правильном порядке, чтобы не заблокировать друг друга. Фактически вызовы `get()` не происходят до вызовов `async`.

В конечном счете мы наконец можем вызвать функцию `.get()` для значения типа `future`, которое вернула функция `result()`, и получить финальную строку.

10 Файловая система

В этой главе:

- реализация нормализатора пути к файлу;
- получение канонических путей к файлам из относительных путей;
- составление списка всех файлов в каталоге;
- реализация средства поиска текста в стиле `grep`;
- реализация автоматического средства для переименования файлов;
- реализация счетчика использования диска;
- вычисление статистики о типах файлов;
- реализация средства, которое уменьшает размер папки путем замены дубликатов символьными ссылками.

Введение

Работа с путями файлов в файловой системе всегда утомительна, если у нас нет специальной библиотеки, поскольку при решении данной задачи необходимо учитывать множество условий.

Одни пути файлов являются *абсолютными*, другие — *относительными*, и, возможно, даже не являются прямыми, поскольку содержат косвенные адреса: `.` (текущий каталог) и `..` (родительский каталог). В то же время в различных операционных системах для разделения каталогов используется слеш / (Linux, MacOS и различные системы UNIX) или обратный слеш \ (Windows). И конечно же, существуют разные типы файлов.

Поскольку каждая вторая программа, связанная с работой в файловой системе, нуждается в таком функционале, было бы здорово иметь новую библиотеку файловой системы в C++17 STL. Самое лучшее в данной ситуации то, что принцип работы одинаков для разных операционных систем, поэтому не требуется писать

разные фрагменты кода для версий программ, поддерживающих разные операционные системы.

В этой главе мы сначала рассмотрим принцип работы класса `path`, поскольку он выступает самым важным элементом библиотеки. Затем увидим, насколько мощными, но простыми в использовании являются классы `directory_iterator` и `recursive_directory_iterator` при работе с файлами. В конце главы задействуем в примерах маленькие и простые инструменты, которые выполняют реальные задачи, связанные с файловой системой. С этого момента можно будет легко создавать более сложные инструменты.

Реализуем нормализатор пути файла

Мы начинаем эту главу с очень простого примера, иллюстрирующего работу класса `std::filesystem::path` и вспомогательной функции, которая рационально нормализует пути к файлам.

Результатом данного примера является небольшое приложение, которое принимает любой путь к файлу и возвращает его в нормализованной форме. Нормализованный путь к файлу — абсолютный путь к файлу, не содержащий косвенных адресов `.` или `..`.

При реализации примера мы также увидим, на какие детали следует обратить внимание во время работы с этой базовой частью библиотеки, связанной с файловой системой.

Как это делается

В этом примере мы реализуем программу, которая всего лишь принимает путь к файлу как аргумент командной строки, а затем отображает его в нормализованной форме.

1. Файл с кодом начинается с директив `include`, а затем мы объявляем, что используем пространства имен `std` и `filesystem`:

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. В функции `main` проверяем, предоставил ли пользователь аргумент командной строки. В случае отрицательного ответа выдаем сообщение об ошибке и отображаем на экране, как правильно работать с программой. Если путь к файлу все же был предоставлен, то создаем на его основе экземпляр объекта `filesystem::path`:

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
```

```
        cout << "Usage: " << argv[0] << " <path>\n";
        return 1;
    }
    const path dir {argv[1]};
```

3. Поскольку можно создавать объекты класса `path` из любой строки, нельзя быть уверенными в том, что полученный путь к файлу существует в файловой системе компьютера. Чтобы в этом убедиться, можем использовать функцию `filesystem::exists`. Если данного пути к файлу не существует, то просто снова отображаем сообщение об ошибке:

```
    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
```

4. Итак, на данном этапе мы вполне уверены: пользователь предоставил *существующий* путь к файлу, зная, что мы можем запросить нормализованную версию, которую затем отобразим. Функция `filesystem::canonical` возвращает другой объект класса `path`. Мы можем вывести на экран его напрямую, но перегруженный оператор `<<` класса `path` берет в кавычки пути к файлу. Чтобы этого избежать, можем отобразить путь к файлу с помощью методов `.c_str()` или `.string()`:

```
    cout << canonical(dir).c_str() << '\n';
}
```

5. Скомпилируем программу и поработаем с ней. Когда мы запустим ее в домашнем каталоге и передадим относительный путь к файлу `"src"`, она выведет на экран полный абсолютный путь к файлу.

```
$ ./normalizer src
/Users/tfc/src
```

6. Когда мы снова запускаем программу в домашнем каталоге, но даем ей запутанное относительное описание пути к файлу, в котором сначала прописывается вход в папку Desktop, потом прописывается выход из нее с помощью косвенного адреса `..`, затем входим в папку Documents и выходим из нее, чтобы в конечном итоге попасть в каталог `src`, программа отображает *тот же* путь файла, что и ранее!

```
$ ./normalizer Desktop/../Documents/../src
/Users/tfc/src
```

Как это работает

Этот начальный пример работы с `std::filesystem` мы сделали относительно коротким и прямолинейным. Мы инициализировали объект класса `path` на основе строки, которая содержит описание пути к файлу. Класс `std::filesystem::path` имеет самое первостепенное значение в тех ситуациях, когда мы используем би-

блиотеку, связанную с файловой системой, поскольку с ним связано большинство функций и классов.

Функция `filesystem::exists` позволяет проверить, существует ли указанный путь файла на самом деле. До сего момента мы не можем быть в этом уверены, поскольку возможно создавать объекты класса `path`, которые не относятся к реальному объекту файловой системы. Функция `exists` всего лишь принимает экземпляр класса `path` и возвращает значение `true` в том случае, если он действительно существует. Эта функция способна сама определить, какой путь мы ей передали (абсолютный или относительный), что делает ее очень комфортной в применении.

Наконец, мы использовали функцию `filesystem::canonical` для каталога, чтобы вывести на экран его нормализованную форму:

```
path canonical(const path& p, const path& base = current_path());
```

Функция `canonical` принимает путь к файлу и в качестве необязательного второго аргумента еще один путь к файлу. Второй путь `base` добавляется к пути файла `p` в том случае, если `p` является относительным. После этого функция `canonical` пытается убрать все косвенные адреса `.` и `..`.

Для вывода результата на экран мы использовали метод `.c_str()`, которому передали канонический путь к файлу. Мы сделали так потому, что перегруженный оператор `<<` для выходных потоков берет пути к файлам в кавычки, а это не всегда желательно.

Дополнительная информация

Функция `canonical` генерирует исключение типа `fileystem_error`, если путь, который мы хотим привести к каноническому виду, не существует. Для предотвращения этого мы проверили наш путь к файлу с помощью функции `exists`. Но было ли достаточно данной проверки, чтобы необработанные исключения не генерировались? Нет.

Обе функции, как `exists`, так и `canonical`, способны генерировать исключения типа `bad_alloc`. Если бы эти исключения сгенерировались, кто-то мог бы утверждать, что программа все равно обречена. Более важная, а также гораздо более вероятная проблема возникает, когда где-то между проверкой существования файла и процессом приведения его к каноническому виду некто переименовывает или удаляет основной файл! В этом случае функция `canonical` сгенерирует сообщение об ошибке `filesystem_error`, хотя мы ранее уже убедились в том, что файл существует.

Большая часть функций файловой системы имеет еще одну перегруженную версию, которая принимает те же аргументы, а также ссылку на `std::error_code`:

```
path canonical(const path& p, const path& base = current_path());
path canonical(const path& p, error_code& ec);
path canonical(const std::filesystem::path& p,
               const std::filesystem::path& base,
               std::error_code& ec );
```

Таким образом, можно выбрать, окружать ли запросы к функциям файловой системы конструктами `try-catch`, или же проверять наличие ошибок вручную. Обратите внимание: это изменяет только поведение *ошибок, связанных с файловой системой*! Если в системе заканчивается память, то возможна генерация более сложных исключений, таких как `bad_alloc`, которые могут иметь или не иметь параметр `ec`.

Получаем канонические пути к файлам из относительных путей

В последнем примере мы уже приводили к каноническому виду/нормализовали пути файлов. Конечно же, класс `filesystem::path` способен не только хранить и проверять пути к файлам. Это помогает легко создавать пути к файлу из строк, а также снова разбивать их на составные части.

На данном этапе класс `path` позволяет абстрагироваться от деталей работы операционной системы, но в некоторых случаях все же о них следует помнить.

Мы увидим, как обращаться с путями и их композицией/декомпозицией, на примере работы с абсолютными и относительными путями.

Как это делается

В данном примере мы будем работать с абсолютными и относительными путями к файлам, чтобы увидеть сильные стороны класса `path` и связанных с ним вспомогательных функций.

1. Сначала включаем все необходимые заголовочные файлы и объявляем, что используем пространства имен `std` и `filesystem`:

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Затем объявляем пример пути к файлу. На данный момент неважно, существует ли текстовый файл, на который он ссылается. Тем не менее есть функции, генерирующие исключения, если требуемого файла нет.

```
int main()
{
    path p {"testdir/foobar.txt"};
```

3. Сейчас мы познакомимся с четырьмя разными функциями библиотеки для работы с файловой системой. Функция `current_path` возвращает путь, в котором в данный момент выполняется программа, — так называемый *рабочий каталог*. Функция `absolute` принимает относительный путь к файлу наподобие нашего пути *p* и возвращает абсолютный, однозначный путь во всей файловой систе-

ме. Функция `system_complete` делает практически то же самое, что и функция `absolute` в Linux, MacOS или других UNIX-подобных операционных системах. В Windows мы получим абсолютный путь к файлу, только вначале будет добавлено буквенное обозначение тома диска (например, `"C:"`). Функция `canonical` опять же делает то же самое, что и функция `absolute`, но потом дополнительно убирает все косвенные адреса, такие как `"."` (сокращение для *«текущий каталог»*) или `".."` (сокращение для *«один каталог вверх»*). Мы рассмотрим работу с данными косвенными адресами в следующих шагах.

```
cout << "current_path      : " << current_path()
     << "\nabsolute_path   : " << absolute(p)
     << "\nsystem_complete : " << system_complete(p)
     << "\ncanonical(p)    : " << canonical(p)
     << '\n';
```

4. Еще одна приятная особенность класса `path` заключается в том, что он перегружает оператор `/`. Таким образом, можно сцеплять имена папок и имена файлов с помощью данного оператора и составлять из них пути файлов. Попробуем это и отобразим составной путь:

```
cout << path{"testdir"} / "foobar.txt" << '\n';
```

5. Рассмотрим работу с функцией `canonical` и составными путями. Передавая функции `canonical` относительный путь к файлу, например `"foobar.txt"`, и составной абсолютный путь `current_path() / "testdir"`, получаем существующий абсолютный путь к файлу. В следующем обращении к функции передаем наш путь `p` (то есть `"testdir/foobar.txt"`) и абсолютный путь `current_path()`, который направляет нас в каталог `"testdir"` и обратно. Это то же самое, что и `current_path()`, из-за косвенных адресов. В обоих вызовах функция `canonical` должна возвратить одинаковый абсолютный путь.

```
cout << "canonical testdir : "
     << canonical("foobar.txt",
                  current_path() / "testdir")
     << "\ncanonical testdir 2 : "
     << canonical(p, current_path() / "testdir/..")
     << '\n';
```

6. Кроме того, можно проверить эквивалентность двух путей, не являющихся каноническими. Функция `equivalence` приводит к каноническому виду пути к файлам, которые она принимает в качестве аргументов, и в конечном итоге возвращает значение `true` при условии, что они описывают один и тот же путь. Для этой проверки путь к файлу должен действительно *существовать*, в противном случае функция сгенерирует исключение.

```
    cout << "equivalence: "
         << equivalent("testdir/foobar.txt",
                       "testdir/../testdir/foobar.txt")
         << '\n';
}
```

7. Компиляция и запуск программы дадут следующий результат. Функция `current_path()` возвращается к домашнему каталогу на моем ноутбуке, поскольку я запустил приложение оттуда. К нашему относительному пути `p` был добавлен префикс, состоящий из данной папки, с помощью функций `absolute_path`, `system_complete` и `canonical`. Мы видим, что функции `absolute_path` и `system_complete` выдают абсолютно одинаковое описание пути файла в моей системе, потому что это Mac (на Linux будет так же). В компьютере с операционной системой Windows функция `system_complete` добавит префикс `"C:"` или любого другого диска, в котором расположен рабочий каталог.

```
$ ./canonical_filepath
current_path    : "/Users/tfc"
absolute_path   : "/Users/tfc/testdir/foobar.txt"
system_complete : "/Users/tfc/testdir/foobar.txt"
canonical(p)    : "/Users/tfc/testdir/foobar.txt"
"testdir/foobar.txt"
canonical testdir   :  "/Users/tfc/testdir/foobar.txt"
canonical testdir 2 : "/Users/tfc/testdir/foobar.txt"
equivalence: 1
```

8. Мы не обрабатываем никаких исключений в нашей короткой программе. При удалении файла `foobar.txt` из каталога `testdir` программа прекращает свою работу из-за исключения. Функция `canonical` требует наличия действительного пути файла. Существует также функция `weakly_canonical`, которая не предъявляет подобных требований.

```
$ ./canonial_filepath
current_path : "/Users/tfc"
absolute_path : "/Users/tfc/testdir/foobar.txt"
system_complete : "/Users/tfc/testdir/foobar.txt"
terminate called after throwing an instance of
'std::filesystem::v1:: cxx11::filesystem_error'
what(): filesystem error: cannot canonicalize:
No such file or directory [testdir/foobar.txt] [/Users/tfc]
```

Как это работает

Цель данного примера заключается в том, чтобы увидеть, как легко создавать новые пути динамически. В основном это связано с наличием в классе `path` удобного перегруженного оператора `/`. Кроме того, функции файловой системы хорошо работают с абсолютными и относительными путями к файлам, а также с путями, которые содержат косвенные адреса `.` и `..`.

Есть довольно много функций, которые возвращают части экземпляра `path`, иногда даже преобразуя их. Не будем перечислять все существующие функции, поскольку для знакомства с ними лучше всего обратиться к справочным материалам по C++.

Однако функции-члены класса `path`, возможно, стоит рассмотреть поближе. Посмотрим, каким функциям-членам класса `path` соответствуют конкретные части пути к файлу. На следующей диаграмме показано, что пути файлов в Windows несколько отличаются от путей файлов в UNIX/Linux (рис. 10.1).

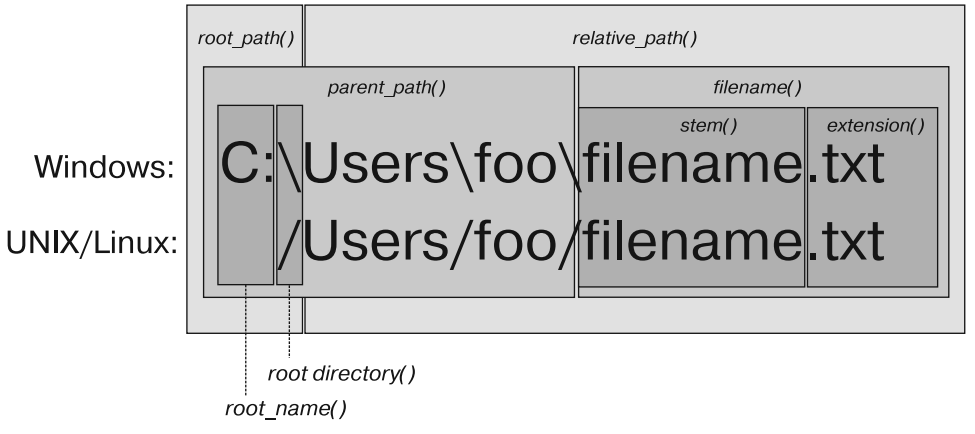

Рис. 10.1

Как видите, функции-члены класса `path` возвращаются для *абсолютного* пути. Для *относительных* путей `root_path`, `root_name` и `root_directory` пусты. `relative_path`, соответственно, возвращает путь, только если тот уже является относительным.

Составляем список всех файлов в каталоге

Конечно же, каждая операционная система, предлагающая поддержку файловой системы, также поставляется с утилитой, которая просто *перечисляет* все файлы внутри каталога в файловой системе. Самые простые примеры — команда `ls` в Linux, MacOS и других UNIX-подобных операционных системах. В DOS и Windows существует команда `dir`. Обе команды составляют список из всех файлов в каталоге и предоставляют дополнительную информацию, такую как размер файла, разрешения и т. д.

Однако переопределение такого инструмента также является хорошим примером, который позволит нам научиться выполнять обходы каталогов и файлов. Давайте просто сделаем это!

Наша собственная утилита `ls/dir` будет способна упорядочивать по имени все файлы в каталоге, их флаги разрешения доступа и отображать количество байт, которые они занимают в файловой системе.

Как это делается

В этом примере мы реализуем небольшой инструмент, который составляет список файлов в любом предоставленном пользователем каталоге. Он будет упорядочивать файлы в списке не только по имени, но и по типу, размеру и разрешениям доступа.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem` по умолчанию:

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <numeric>
#include <algorithm>
#include <vector>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Понадобится вспомогательная функция `file_info`. Она принимает ссылку на объект `directory_entry` и извлекает из нее путь, а также объект `file_status` (с помощью функции `status`), который содержит тип файла и информацию о правах. Наконец, она извлекает и размер записи, если это обычный файл. Для каталогов и особых файлов мы просто возвращаем значение `0`. Вся информация упаковывается в кортеж.

```
static tuple<path, file_status, size_t>
file_info(const directory_entry &entry)
{
    const auto fs (status(entry));
    return {entry.path(),
            fs,
            is_regular_file(fs) ? file_size(entry.path()) : 0u};
}
```

3. Кроме того, понадобится вспомогательная функция `type_char`. Путь может представлять не только каталоги и простые текстовые/бинарные файлы. Операционные системы предоставляют множество разнообразных типов, которые абстрагируют что-то еще, например интерфейсы аппаратных устройств в виде так называемых символьных/блочных файлов. В библиотеке для работы с файловой системой, расположенной в STL, есть множество функций-предикатов для них. Таким образом, можно вернуть букву `'d'` для каталогов, букву `'f'` для обычных файлов и т. д.:

```
static char type_char(file_status fs)
{
    if      (is_directory(fs))      { return 'd'; }
    else if (is_symlink(fs))        { return 'l'; }
    else if (is_character_file(fs)) { return 'c'; }
    else if (is_block_file(fs))     { return 'b'; }
    else if (is_fifo(fs))           { return 'p'; }
```

```
    else if (is_socket(fs))         { return 's'; }
    else if (is_other(fs))          { return 'o'; }
    else if (is_regular_file(fs))   { return 'f'; }
    return '?';
}
```

4. Напишем еще одну вспомогательную функцию, `rwx`. Она принимает переменную `perms` (просто возвращающую тип класса перечисления из библиотеки для работы с файловой системой) и возвращает строку наподобие `"rwxrwxrwx"`, которая описывает настройки прав для файла. Первая группа символов `"rwx"` описывает права на чтение, запись и исполнение (**r***ead,* **w***rite and* **e***xecution*) для владельца файла. Следующая группа описывает те же права для всех пользователей, являющихся частью *пользовательской группы*, к которой принадлежит файл. Последняя группа символов описывает эти же права для всех остальных. Строка `"rwxrwxrwx"` означает, что все пользователи могут получить доступ к объекту любым способом; строка `"rw-r--r--"` — что только владелец файла может читать и изменять его, а все остальные — только читать. Мы просто создадим строку на основе этих значений бит за битом. Лямбда-выражение поможет выполнить повторяющуюся работу, связанную с проверкой, содержит ли переменная `p` типа `perms` конкретный бит владельца, и возвратит символ `'-'` или соответствующую букву:

```
static string rwx(perms p)
{
    auto check ([p](perms bit, char c) {
        return (p & bit) == perms::none ? '-' : c;
    });
    return {check(perms::owner_read,   'r'),
            check(perms::owner_write,  'w'),
            check(perms::owner_exec,   'x'),
            check(perms::group_read,   'r'),
            check(perms::group_write,  'w'),
            check(perms::group_exec,   'x'),
            check(perms::others_read,  'r'),
            check(perms::others_write, 'w'),
            check(perms::others_exec,  'x')};
}
```

5. Наконец, последняя вспомогательная функция принимает целочисленный размер файла и преобразует его в читабельный вид. Мы просто проигнорируем точку при делении чисел и округлим их до ближайшего значения кило-, мега- или гигабайт.

```
static string size_string(size_t size)
{
    stringstream ss;
    if        (size >= 1000000000) {
        ss << (size / 1000000000) << 'G';
    } else if (size >= 1000000)    {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000)       {
```

```
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }
    return ss.str();
}
```

6. Теперь наконец можно реализовать функцию `main`. Начнем с проверки того, предоставил ли пользователь путь в командной строке. Если он этого не сделал, то просто возьмем текущий каталог `"."`. Затем проверим, существует ли данный каталог. При его отсутствии мы не можем создать список файлов.

```
int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};
    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
```

7. Заполним вектор кортежами, содержащими информацию о файлах, которые возвращает наша первая вспомогательная функция `file_info` из объектов `directory_entry`. Создадим `directory_iterator` и передадим в его конструктор объект пути, созданный на предыдущем шаге. При переборе с помощью итератора для каталогов преобразуем объекты типа `directory_entry` в кортежи, содержащие информацию о файлах, и вставляем их в вектор:

```
    vector<tuple<path, file_status, size_t>> items;
    transform(directory_iterator{dir}, {},
        back_inserter(items), file_info);
```

8. Мы сохранили всю необходимую информацию в элементы вектора и можем просто сохранить ее с помощью написанных вспомогательных функций:

```
    for (const auto &[path, status, size] : items) {
        cout << type_char(status)
             << rwx(status.permissions()) << " "
             << setw(4) << right << size_string(size)
             << " " << path.filename().c_str()
             << '\n';
    }
}
```

9. Компиляция и запуск проекта с путем к офлайн-версии документации C++ дадут следующий результат. Мы видим, что папка содержит только другие каталоги и простые файлы, поскольку первыми символами каждой строки являются `'d'` и `'f'`. Данные файлы имеют разные права доступа и, конечно, различаются по размеру. Обратите внимание: файлы представлены в алфавитном порядке, но мы не можем полагаться на эту особенность, поскольку это не требуется в стандарте C++17.

```
$ ./list ~/Documents/cpp_reference/en/cpp
drwxrwxr-x    0B  algorithm
```

```
frw-r--r--   88K  algorithm.html
drwxrwxr-x    0B  atomic
frw-r--r--   35K  atomic.html
drwxrwxr-x    0B  chrono
frw-r--r--   34K  chrono.html
frw-r--r--   21K  comment.html
frw-r--r--   21K  comments.html
frw-r--r--  220K  compiler_support.html
drwxrwxr-x    0B  concept
frw-r--r--   67K  concept.html
drwxr-xr-x    0B  container
frw-r--r--  285K  container.html
drwxrwxr-x    0B  error
frw-r--r--   52K  error.html
```

Как это работает

В этом примере мы проитерировали по файлам и для каждого из них проверили его статус и размер. Несмотря на то что все наши операции с файлами оказались довольно прямолинейными и простыми, обход каталога выглядит несколько непонятно.

Чтобы обойти каталог, мы просто создали итератор `directory_iterator` и проитерировали с его помощью. Обход каталога фантастически легко совершить с помощью библиотеки `filesystem`:

```
for (const directory_entry &e : directory_iterator{dir}) {
    // сделать что-то
}
```

Про этот класс можно сказать лишь следующее:

- он проверяет каждый элемент один раз;
- порядок, в котором выполняется перебор, не определен;
- элементы каталога `.` и `..` уже отфильтрованы.

Однако можно заметить, что итератор `directory_iterator` ведет себя как *итератор* и как *итерабельный диапазон* одновременно. Почему? В небольшом примере с циклом `for` мы видели, как он используется в качестве итерабельного диапазона. В самом коде примера мы применили его как итератор:

```
transform(directory_iterator{dir}, {},
          back_inserter(items), file_info);
```

Правда заключается в том, что это всего лишь класс итератора, но функции `std::begin` и `std::end` предоставляют перегруженные версии данного типа. Данное обстоятельство позволяет вызвать функции `begin` и `end` для подобного итератора, и они снова будут возвращать итераторы. Это может показаться странным на первый взгляд, но делает наш класс более полезным.

Инструмент текстового поиска в стиле grep

Большая часть операционных систем оснащена неким локальным инструментом поиска. Пользователи могут запустить его нажатием сочетания клавиш, а затем ввести имя локального файла, который хотели бы найти.

Прежде чем появилась подобная возможность, пользователи командной строки выполняли поиск файлов с помощью таких инструментов, как `grep` или `awk`. Пользователь мог просто ввести команду наподобие `"grep -r foobar ."`, и инструмент рекурсивно прошел бы по текущему каталогу и нашел бы все файлы, содержащие строку `"foobar"`.

В этом примере мы реализуем точно такое же приложение. Наш небольшой клон `grep` будет принимать шаблон из командной строки, а затем выполнять рекурсивный поиск в том каталоге, где мы находимся на момент запуска приложения. Он выведет имена всех файлов, соответствующих нашему шаблону. Проверка на совпадение с шаблоном будет применяться построчно, так что можно вывести на экран еще и номера строк в файле, соответствующих шаблону.

Как это делается

В этом разделе мы реализуем небольшой инструмент, который выполняет поиск предоставленных пользователем текстовых шаблонов в файлах. Инструмент работает точно так же, как и инструмент UNIX `grep`, но для простоты будет не таким зрелым и эффективным.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem`:

```
#include <iostream>
#include <fstream>
#include <regex>
#include <vector>
#include <string>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Потом реализуем вспомогательную функцию. Она принимает путь к файлу и объект, содержащий регулярное выражение, описывающее искомый шаблон. Затем создаем экземпляр вектора, который будет включать пары, состоящие из номера строк и их содержимого. Кроме того, создадим экземпляр объекта файлового потока ввода, из которого будем считывать содержимое и сравнивать его с шаблоном строка за строкой:

```
static vector<pair<size_t, string>>
matches(const path &p, const regex &re)
```

```
{
    vector<pair<size_t, string>> d;
    ifstream is {p.c_str()};
```

3. Пройдем по файлу строка за строкой с помощью функции `getline`. Функция `regex_search` возвращает значение `true` при условии, что строка содержит наш шаблон. Если это именно так, то поместим в вектор номер строки и саму строку. Наконец, вернем все найденные совпадения:

```
    string s;
    for (size_t line {1}; getline(is, s); ++line) {
        if (regex_search(begin(s), end(s), re)) {
            d.emplace_back(line, move(s));
        }
    }
    return d;
}
```

4. В функции `main` сначала проверим, предоставил ли пользователь аргумент командной строки, который можно задействовать как шаблон. Если нет, то сгенерируем ошибку:

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Usage: " << argv[0] << " <pattern>\n";
        return 1;
    }
```

5. Далее создадим объект регулярного выражения из входного шаблона. Невозможность создать такое регулярное выражение приведет к генерации исключения. При генерации исключения поймаем его и сгенерируем ошибку:

```
    regex pattern;
    try { pattern = regex{argv[1]}; }
    catch (const regex_error &e) {
        cout << "Invalid regular expression provided.n";
        return 1;
    }
```

6. Наконец, можно проитерировать по файловой системе и поискать совпадения с шаблоном. Воспользуемся итератором `recursive_directory_iterator` для итерации по всем файлам рабочего каталога. Он работает точно так же, как и итератор `directory_iterator` из предыдущего примера, но заходит еще и в подкаталоги. Таким образом, не нужно управлять рекурсией. Для каждой записи вызываем вспомогательную функцию `matches`:

```
    for (const auto &entry :
          recursive_directory_iterator{current_path()}) {
        auto ms (matches(entry.path(), pattern));
```

7. Для каждого совпадения (если они есть) выводим путь к файлу, номер строки и содержимое строки, содержащей совпадение:

```
        for (const auto &[number, content] : ms) {
            cout << entry.path().c_str() << ":" << number
                 << " - " << content << '\n';
        }
    }
}
```

8. Подготовим файл с именем `"foobar.txt"`, содержащий тестовые строки, по которым можно выполнить поиск:

```
foo
bar
baz
```

9. Компиляция и запуск программы дадут следующий результат. Я запустил приложение в каталоге `/Users/tfc/testdir` моего ноутбука и сначала передал ему шаблон `"bar"`. Внутри этого каталога приложение нашло вторую строку в нашем файле `"foobar.txt"` и другом файле `"text1.txt"`, который находится в каталоге `testdir/dir1`:

```
$ ./grepper bar
/Users/tfc/testdir/dir1/text1.txt:1 - foo bar bla blubb
/Users/tfc/testdir/foobar.txt:2 - bar
```

10. При повторном запуске приложения с шаблоном `"baz"` оно находит третью строку в нашем примере текстового файла:

```
$ ./grepper baz
/Users/tfc/testdir/foobar.txt:3 - baz
```

Как это работает

Создание и использование регулярного выражения с целью фильтрации содержимого файлов — основная цель данного примера. Однако рассмотрим итератор `recursive_directory_iterator`, поскольку этот особый класс итераторов мы применяли для фильтрации файлов, по которым итерируем рекурсивно.

Как и `directory_iterator`, `recursive_directory_iterator` итерирует по элементам каталога. Он делает это рекурсивно, согласно своему названию. При встрече с элементом файловой системы, который является *каталогом*, он вернет экземпляр типа `directory_entry` для данного пути, а затем зайдет в него, чтобы проитерировать по его потомкам.

Итератор `recursive_directory_iterator` имеет несколько интересных функций-членов.

- `depth()` — говорит, на сколько уровней итератор спустился в подкаталоге.
- `recursion_pending()` — сообщает, будет ли итератор спускаться дальше после элемента, на который он указывает в данный момент.

- `disable_recursion_pending()` — эту функцию можно вызвать, чтобы помешать итератору спуститься в следующий подкаталог, если сейчас он указывает на каталог, в который можно спуститься. Это значит, что вызов указанного метода ничего не даст, если мы совершим данное действие *слишком рано*.
- `pop()` — эта функция прерывает работу на текущем уровне и поднимает итератор на один уровень вверх в иерархии каталогов для продолжения работы.

Дополнительная информация

Еще одной важной деталью, о которой нужно знать, выступает класс-перечисление `directory_options`. Конструктор класса `recursive_directory_iterator` принимает значение этого типа в качестве второго аргумента. Значением по умолчанию, которое мы использовали неявно, является `directory_options::none`. Другие его значения выглядят следующим образом:

- `follow_directory_symlink` — позволяет рекурсивному итератору следовать по символьным ссылкам на каталоги;
- `skip_permission_denied` — указывает итератору пропускать каталоги, которые в противном случае вернут ошибку, поскольку файловая система не дает прав на доступ к ним.

Эти настройки можно объединять с помощью оператора `|`.

Инструмент для автоматического переименования файлов

На создание этого примера меня сподвигла ситуация, в которую я попадаю довольно часто. Скажем, при объединении в одном каталоге файлов с фотографиями от разных друзей и с разных устройств можно заметить, что расширения этих файлов различаются. Одни файлы формата JPEG имеют расширение `.jpg`, другие — `.jpeg`, а третьи — и вовсе `.JPEG`.

Некоторым людям нравится делать все расширения одинаковыми. Было бы полезно иметь возможность переименовать все файлы лишь одной командой. В то же время мы могли бы удалить все пробелы `' '` и заменить их, например, на `'_'`.

В данном примере мы реализуем такой инструмент и назовем его `renamer`. Он будет принимать диапазон входных шаблонов и их замен, это выглядит следующим образом:

```
$ renamer jpeg jpg JPEG jpg
```

В этом случае `renamer` рекурсивно проитерирует по текущему каталогу и выполнит поиск шаблонов `jpeg` и `JPEG` в именах всех файлов. Он заменит обе строки на `jpg`.

Как это делается

В этом примере мы реализуем инструмент, который рекурсивно просканирует все файлы внутри каталога и соотнесет их имена с заданным шаблоном. Все совпадения заменятся токенами, предоставленными пользователем, и найденные файлы будут переименованы соответствующим образом.

1. Сначала включим некоторые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem`:

```
#include <iostream>
#include <regex>
#include <vector>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Реализуем небольшую вспомогательную функцию, которая принимает путь к файлу в виде строки, а также диапазон пар для замены. Каждая пара для замены содержит шаблон и его замену. Проходя в цикле по диапазону, воспользуемся `regex_replace`, передавая ему входную строку и принимая преобразованную строку. После этого вернем полученную строку:

```
template <typename T>
static string replace(string s, const T &replacements)
{
    for (const auto &[pattern, repl] : replacements) {
        s = regex_replace(s, pattern, repl);
    }
    return s;
}
```

3. В функции `main` сначала проверим командную строку. Принимаем аргументы из нее *попарно*, поскольку нужно, чтобы шаблоны стояли рядом с их заменами. Первый элемент `argv` — всегда имя исполняемого файла. Это значит, что если пользователь предоставит хотя бы одну пару или больше, то `argc` должен быть *нечетным* и не меньше `3`:

```
int main(int argc, char *argv[])
{
    if (argc < 3 || argc % 2 != 1) {
        cout << "Usage: " << argv[0]
             << " <pattern> <replacement> ...\n";
        return 1;
    }
```

4. Как только мы убедились, что получили во входных значениях необходимые пары, заполним ими вектор:

```
    vector<pair<regex, string>> patterns;
    for (int i {1}; i < argc; i += 2) {
```

```
        patterns.emplace_back(argv[i], argv[i + 1]);
    }
```

5. Теперь можно проитерировать по файловой системе. Для простоты определим в качестве каталога, по которому нужно проитерировать, текущий каталог приложения. Затем возьмем только имена файлов без остальной части пути и преобразуем их соответственно списку шаблонов и замен, собранному ранее. Возьмем копию `opath`, назовем ее `rpath` и заменим часть имени файла новой:

```
    for (const auto &entry :
          recursive_directory_iterator{current_path()}) {
        path opath {entry.path()};
        string rname {replace(opath.filename().string(),
                              patterns)};
        path rpath {opath};
        rpath.replace_filename(rname);
```

6. Для всех файлов, чьи имена совпали с нашими шаблонами, выведем на экран строку, сигнализирующую о том, что мы переименовали их. Если полученное в результате переименования имя файла уже существует, то продолжать работу невозможно. Просто будем пропускать такие файлы. Вместо этого мы могли бы прикреплять какое-то число к пути или что-то еще с целью разрешить пересечение имен.

```
        if (opath != rpath) {
            cout << opath.c_str() << " --> "
                 << rpath.filename().c_str() << '\n';
            if (exists(rpath)) {
                cout << "Error: Can't rename."
                        " Destination file exists.\n";
            } else {
                rename(opath, rpath);
            }
        }
    }
}
```

7. Компиляция и запуск программы для примера каталога дадут следующий результат. Я поместил несколько картинок в формате JPEG в каталог, но задал для них разные окончания: `jpg`, `jpeg` и `JPEG`. Затем выполнил программу, передав ей шаблоны `jpeg` и `JPEG` и выбрав замену `jpg` для них обоих. В результате получил каталог с одинаковыми расширениями файлов:

```
$ ls
birthday_party.jpeg   holiday_in_dubai.jpg  holiday_in_spain.jpg
trip_to_new_york.JPEG
$ ../renamer jpeg jpg JPEG jpg
/Users/tfc/pictures/birthday_party.jpeg --> birthday_party.jpg
/Users/tfc/pictures/trip_to_new_york.JPEG --> trip_to_new_york.jpg
$ ls
birthday_party.jpg   holiday_in_dubai.jpg holiday_in_spain.jpg
trip_to_new_york.jpg
```

Создаем индикатор эксплуатации диска

Мы уже реализовали инструмент, который работает как `ls` в Linux/MacOS или `dir` в Windows, но, подобно этим утилитам, не выводит размер файлов в *каталогах*.

Чтобы получить эквивалент размера каталога, нужно зайти во все подкаталоги и суммировать размеры всех файлов, содержащихся в них.

В данном примере мы реализуем инструмент, который делает именно это. Инструмент можно запустить для любого каталога, он определит размер всех его записей.

Как это делается

В этом примере мы реализуем приложение, которое итерирует по каталогу и перечисляет размеры файлов для каждой записи. Это просто для обычных файлов, но если мы смотрим на запись каталога, которая сама по себе является каталогом, то нужно заглянуть в него и суммировать размеры всех файлов, хранящихся в нем.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem`:

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <numeric>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Затем реализуем вспомогательную функцию, которая принимает в качестве аргумента `directory_entry` и возвращает его размер в файловой системе. Это не каталог, мы просто вернем размер файла, вычисленный с помощью `file_size`:

```
static size_t entry_size(const directory_entry &entry)
{
    if (!is_directory(entry)) { return file_size(entry); }
```

3. Если это каталог, то нужно проитерировать по всем его записям и подсчитать их размер. Мы будем вызывать вспомогательную функцию `entry_size` рекурсивно при повторной встрече с подкаталогами:

```
    return accumulate(directory_iterator{entry}, {}, 0u,
        [](size_t accum, const directory_entry &e) {
            return accum + entry_size(e);
        });
}
```

4. Для повышения читабельности воспользуемся функцией `size_string`, которая уже встречалась в этой главе. Она просто сокращает большие размеры файлов, делая их «аккуратнее» и добавляя префиксы «кило», «мега» или «гига»:

```
static string size_string(size_t size)
{
    stringstream ss;
    if        (size >= 1000000000) {
        ss << (size / 1000000000) << 'G';
    } else if (size >= 1000000)    {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000)       {
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }
    return ss.str();
}
```

5. Первое, что нужно сделать в функции `main`, — проверить, предоставил ли пользователь путь к файлу в командной строке. Если это не так, то возьмем текущий каталог. Прежде чем продолжить, проверим, существует ли данный каталог:

```
int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};
    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
```

6. Теперь можно проитерировать по всем записям каталога и вывести на экран их имена и размер:

```
    for (const auto &entry : directory_iterator{dir}) {
        cout << setw(5) << right
             << size_string(entry_size(entry))
             << " " << entry.path().filename().c_str()
             << '\n';
    }
}
```

7. Компиляция и запуск программы дадут следующий результат. Я запустил ее для каталога, в котором находится офлайн-справка по C++. Поскольку он содержит и подкаталоги, наша вспомогательная функция, суммирующая размер файла, очень пригодится:

```
$ ./file_size ~/Documents/cpp_reference/en/
  19M c
  12K c.html
 147M cpp
  17K cpp.html
  22K index.html
  22K Main_Page.html
```

Как это работает

Вся программа строится на использовании функции `file_size` для обычных файлов. Если программа увидит каталог, то рекурсивно спустится в него и вызовет функцию `file_size` для всех его записей.

Единственное, что мы сделали для определения того, можем ли вызвать непосредственно `file_size` или нужно рекурсивно спуститься дальше, — реализовали предикат `is_directory`. Он работает для каталогов, которые содержат только обычные файлы и каталоги.

Поскольку наша программа довольно проста, она даст сбой при следующих условиях.

- Функция `file_size` работает только для обычных файлов и символьных ссылок. Она генерирует исключение во всех других случаях.
- Несмотря на то что функция `file_size` работает для символьных ссылок, она *все еще* сгенерирует исключение, если мы вызовем ее для *неработающей* символьной ссылки.

Чтобы сделать программу из нашего примера более надежной, следует воспользоваться защитным программированием в случаях неверных типов файлов и обработки исключений.

Подбиваем статистику о типах файлов

В предыдущем примере мы реализовали инструмент, который выводит на экран размер всех членов каталога.

В текущем примере тоже будем определять размер рекурсивно, но в этот раз объединим размеры всех файлов с одинаковым *расширением*. Таким образом, сможем вывести для пользователя таблицу, в которой перечисляется количество и тип файлов, а также средний размер файлов каждого типа.

Как это делается

В этом примере мы реализуем небольшой инструмент, который рекурсивно итерирует по заданному каталогу. В процессе он определяет, файлы с какими расширениями находятся в данном каталоге, сколько файлов присутствует для каждого расширения, а также их средний размер.

1. Сначала включим необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem`:

```
#include <iostream>
#include <sstream>
```

```
#include <iomanip>
#include <map>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Функция `size_string` была полезна в предыдущих примерах. Она преобразует размеры файлов в читабельные строки:

```
static string size_string(size_t size)
{
    stringstream ss;
    if        (size >= 1000000000) {
        ss << (size / 1000000000) << 'G';
    } else if (size >= 1000000)    {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000)       {
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }
    return ss.str();
}
```

3. Затем реализуем вспомогательную функцию, которая принимает объект пути и итерирует по всем файлам данного пути. При этом она собирает всю информацию в ассоциативный массив, в котором соотносятся расширения файлов и пары, содержащие общее количество файлов и их суммарный размер.

```
static map<string, pair<size_t, size_t>> ext_stats(const path &dir)
{
    map<string, pair<size_t, size_t>> m;
    for (const auto &entry :
          recursive_directory_iterator{dir}) {
```

4. Если запись каталога тоже является каталогом, то ее можно опустить. Это не значит, что мы не будем рекурсивно спускаться в каталог. Итератор `recursive_directory_iterator` все равно совершит данное действие, но мы не хотим сами заходить в эти подкаталоги.

```
        const path        p  {entry.path()};
        const file_status fs {status(p)};
        if (is_directory(fs)) { continue; }
```

5. Далее извлекаем расширения из строки, представляющей запись каталога. Если у нее нет расширения, то просто опускаем ее:

```
        const string ext {p.extension().string()};
        if (ext.length() == 0) { continue; }
```

6. Подсчитаем размер текущего файла. Затем найдем агрегатный объект в ассоциативном массиве для этого расширения. Если такого объекта нет, то неявно

создадим его. Просто увеличим счетчик количества файлов и добавим размер файла в переменную-аккумулятор:

```
        const size_t size {file_size(p)};

        auto &[size_accum, count] = m[ext];
        size_accum += size;
        count      += 1;
    }
```

7. После этого вернем ассоциативный массив:

```
    return m;
}
```

8. В функции `main` примем путь, предоставленный пользователем в командной строке. Конечно, нужно проверить, существует ли он, поскольку в противном случае продолжать не имеет смысла.

```
int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};
    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
```

9. Можно мгновенно проитерировать по ассоциативному массиву, предоставляемому `ext_stats`. Поскольку элементы типа `accum_size` этого массива содержат сумму всех файлов с одинаковым расширением, разделим эту сумму на общее количество таких файлов и выведем ее на экран:

```
    for (const auto &[ext, stats] : ext_stats(dir)) {
        const auto &[accum_size, count] = stats;
        cout << setw(15) << left << ext << ": "
             << setw(4) << right << count
             << " items, avg size "
             << setw(4) << size_string(accum_size / count)
             << '\n';
    }
}
```

10. Компиляция и запуск программы дадут следующий результат. Я предоставил ей в качестве аргумента командной строки каталог, содержащий офлайн-справку по C++.

```
$ ./file_type ~/Documents/cpp_reference/
.css            :    2 items, avg size  41K
.gif            :    7 items, avg size 902B
.html           : 4355 items, avg size  38K
.js             :    3 items, avg size   4K
.php            :    1 items, avg size 739B
.png            :   34 items, avg size   2K
.svg            :   53 items, avg size   6K
.ttf            :    2 items, avg size 421K
```

Инструмент для уменьшения размера папки путем замены дубликатов символьными ссылками

Существует множество инструментов, сжимающих данные разными способами. Наиболее известными примерами таких алгоритмов/форматов являются ZIP и RAR. Подобные инструменты уменьшают размер файлов, снижая их внутреннюю избыточность.

Прежде чем сжать файлы в архив, можно довольно просто снизить использование диска, просто *удалив повторяющиеся* файлы. В данном примере мы реализуем небольшой инструмент, который рекурсивно проходит по каталогу и в процессе ищет файлы с одинаковым содержимым. Если он найдет такие файлы, то удалит все дубликаты, кроме одного. Все удаленные файлы будут заменены символьными ссылками, которые указывают на оставшийся уникальный файл. Это позволяет сэкономить место, не выполняя сжатия и сохраняя данные.

Как это делается

В данном примере мы реализуем небольшой инструмент, который определяет, какие файлы в каталоге дублируют друг друга. Зная это, он удалит все файлы, кроме одного, и заменит их символьными ссылками, что уменьшит размер каталога.

Убедитесь, что создали резервные копии системных данных. Мы будем работать с функциями STL, которые удаляют файлы. Неверно указанный путь для такой программы может привести к тому, что программа жадно удалит нежелательным способом слишком много файлов.

1. Сначала включим все необходимые заголовочные файлы и объявим об использовании пространств имен `std` и `filesystem` по умолчанию:

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. Чтобы определить, какие файлы являются дубликатами друг друга, создадим ассоциативный массив, в котором соотносятся хеши файлов и путь к первому файлу, из которого был получен этот хеш. Для получения таких хешей следует использовать популярный алгоритм, такой как MD5 или SHA. В целях сохранения данного примера чистым и простым просто считаем весь файл в строку, а затем задействуем объект хеш-функции, уже применяемый `unordered_map` для подсчета хешей строк:

```
static size_t hash_from_path(const path &p)
{
```

```
    ifstream is {p.c_str(),
                 ios::in | ios::binary};
    if (!is) { throw errno; }
    string s;
    is.seekg(0, ios::end);
    s.reserve(is.tellg());
    is.seekg(0, ios::beg);
    s.assign(istreambuf_iterator<char>{is}, {});
    return hash<string>{}(s);
}
```

3. Затем реализуем функцию, которая создает такой ассоциативный массив, основанный на хешах, и удаляет дубликаты. Она рекурсивно итерирует по каталогу и его подкаталогам:

```
static size_t reduce_dupes(const path &dir)
{
    unordered_map<size_t, path> m;
    size_t count {0};
    for (const auto &entry :
          recursive_directory_iterator{dir}) {
```

4. Для каждой записи каталога функция проверяет, является ли эта запись каталогом. Все каталоги опускаются. Для каждого файла генерируем значение хеша и пробуем вставить его в ассоциативный массив. Если последний уже содержит такой хеш, то это значит, что файл с таким хешем уже был добавлен. Соответственно, мы нашли дубликат! В случае конфликтов во время вставки второе значение в паре, которую возвращает `try_emplace`, равно `false`.

```
        const path p {entry.path()};
        if (is_directory(p)) { continue; }
        const auto &[it, success] =
            m.try_emplace(hash_from_path(p), p);
```

5. Задействуя значения, возвращаемые `try_emplace`, мы можем сказать пользователю, что мгновение назад вставили файл, поскольку встретили этот хеш в первый раз. Если мы нашли дубликат, то сообщаем пользователю о том, что второй файл является дубликатом, и удаляем его. После удаления создаем символьную ссылку, которая заменяет дубликат.

```
        if (!success) {
            cout << "Removed " << p.c_str()
                 << " because it is a duplicate of "
                 << it->second.c_str() << '\n';
            remove(p);
            create_symlink(absolute(it->second), p);
            ++count;
        }
```

6. После перебора в файловой системе возвращаем количество файлов, которые мы удалили и заменили файловыми ссылками.

```
    }
    return count;
}
```

7. В функции `main` убеждаемся, что пользователь передал каталог в командной строке и что этот каталог существует:

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Usage: " << argv[0] << " <path>\n";
        return 1;
    }
    path dir {argv[1]};
    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
```

8. Единственное, что нам осталось сделать, — вызвать функцию `reduce_dupes` для этого каталога и вывести на экран информацию о том, сколько файлов мы удалили:

```
    const size_t dupes {reduce_dupes(dir)};
    cout << "Removed " << dupes << " duplicates.\n";
}
```

9. Компиляция и запуск программу, для примера каталога, содержащего дубликаты, выглядит следующим образом. Я использовал инструмент `du`, чтобы проверить размер каталога до и после запуска нашей программы, с целью продемонстрировать работоспособность нашего подхода:

```
$ du -sh dupe_dir
1.1M dupe_dir
$ ./dupe_compress dupe_dir
Removed dupe_dir/dir2/bar.jpg because it is a duplicate of
dupe_dir/dir1/bar.jpg
Removed dupe_dir/dir2/base10.png because it is a duplicate of
dupe_dir/dir1/base10.png
Removed dupe_dir/dir2/baz.jpeg because it is a duplicate of
dupe_dir/dir1/baz.jpeg
Removed dupe_dir/dir2/feed_fish.jpg because it is a duplicate of
dupe_dir/dir1/feed_fish.jpg
Removed dupe_dir/dir2/foo.jpg because it is a duplicate of
dupe_dir/dir1/foo.jpg
Removed dupe_dir/dir2/fox.jpg because it is a duplicate of
dupe_dir/dir1/fox.jpg
Removed 6 duplicates.
$ du -sh dupe_dir
584K dupe_dir
```

Как это работает

Мы использовали функцию `create_symlink`, чтобы создать входную точку в другой файл в файловой системе. Это позволит избежать наличия дубликатов. Кроме того, можно создать жесткую ссылку с помощью функции `create_hard_link`. Семантически оба этих подхода похожи друг на друга, но жесткие ссылки имеют другие технические последствия, нежели мягкие. Некоторые форматы файловых систем и вовсе могут не поддерживать жесткие ссылки или, например, лишь определенное количество жестких ссылок, ссылающихся на один и тот же файл. Еще одна проблема заключается в том, что жесткие ссылки не могут указывать из одной файловой системы на другую.

Однако помимо деталей реализации существует еще один источник ошибок, проявляющийся при использовании `create_symlink` или `create_hard_link`. В следующих строках содержится ошибка. Можете ли вы ее заметить сразу?

```
path a {"some_dir/some_file.txt"};
path b {"other_dir/other_file.txt"};
remove(b);
create_symlink(a, b);
```

При выполнении этой программы ничего плохого не случится, но символьная ссылка будет *нерабочей*. Она указывает на `"some_dir/some_file.txt"`, а это неверно. Проблема заключается в том, что она должна указывать либо на `"/absolute/path/some_dir/some_file.txt"`, либо на `"../some_dir/some_file.txt"`. Вызов `create_symlink` использует корректный абсолютный путь, если мы напишем следующий код:

```
create_symlink(absolute(a), b);
```

Функция create_symlink не проверяет, корректен ли путь, на который мы создаем ссылку.

Дополнительная информация

Как можно заметить, наша функция определения хеша довольно незамысловата. Мы реализовали ее такой, чтобы пример оставался простым и не имел внешних зависимостей.

В чем заключается проблема нашей хеш-функции? На самом деле есть даже две проблемы.

1. Мы считываем в строку весь файл. Это будет иметь катастрофические последствия для файлов, которые крупнее нашей системной памяти.
2. Типаж хеш-функции `hash<string>`, представленный в C++, скорее всего, не поддерживает такие хеши.

При необходимости найти более качественную функцию подсчета хеша следует выбрать ту, которая работает быстро, безопасно для памяти и позволяет убедиться, что крупные файлы с разным содержимым не получат одинаковый хеш. Последнее требование, возможно, является самым важным. Если мы решим, что один файл является дубликатом другого и притом они не содержат одинаковые данные, то это может привести к *потере данных* после удаления одного из них.

Существуют более качественные алгоритмы хеширования, например MD5 или одна из разновидностей SHA. Чтобы получить доступ к таким функциям в нашей программе, можно использовать, скажем, криптографический API для OpenSSL.

Об авторе

Яцек Галовиц (Jacek Galowicz) получил степень магистра наук в области электротехники и вычислительной техники в Рейнско-Вестфальском техническом университете Ахена (Германия). Во время учебы он работал ассистентом, занимаясь как преподавательской, так и научной деятельностью, а также выступил соавтором нескольких научных публикаций. В то же время в качестве фрилансера Яцек создавал приложения и драйверы ядер на языках C и C++, задействованные в разных областях, таких как графическое программирование в 3D, базы данных, обмен данными по сети и моделирование физической среды. В последнее время он занимается программированием микроядерных операционных систем, предназначенных для виртуализации Intel x86 в Intel и FireEye и предъявляющих особые требования к производительности и безопасности. Яцек обожает реализации низкоуровневого ПО на современном C++ и старается обеспечить его высокую производительность, не усложняя при этом код. В последние годы он изучал чистое функциональное программирование и Haskell, что побудило его переключиться на создание обобщенного кода с помощью метапрограммирования.

Написание книги и в то же самое время основание компании — это интересный опыт, мне даже понравилось. Последнее, однако, оказалось возможно только благодаря поддержке и терпению моей замечательной подруги Виктории (Viktoria), моих коллег и всех моих друзей. Особую благодарность выражаю Арне Мерцу (Arne Mertz) за его бесценные замечания, а также Торстену Робицки (Torsten Robitzki) и Оливеру Брансу (Oliver Bruns) из пользовательской группы C++ Ганновера за обратную связь.

О рецензенте

Арне Мерц (Arne Mertz) — специалист по C++ с более чем десятилетним опытом. Он изучал физику в Гамбургском университете (Германия), а затем стал разработчиком ПО. В основном он работал с финансовыми приложениями, написанными на C++. Арне работает в компании Zühlke Engineering в Германии. Кроме того, он широко известен в кругах программистов благодаря своему блогу Simplify C++! (https://arne-mertz.de), посвященному написанию чистого и удобного в сопровождении кода на C++.

Яцек Галовиц

C++17 STL. Стандартная библиотека шаблонов

Перевел с английского *Е. Зазноба*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,
улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.

Дата изготовления: 03.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 01.03.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*