Язык программирования C++

Краткий курс, второе издание

Бьярне Страуструп

Серия C++ In-Depth · Бьярне Страуструп

Язык программирования C++

Краткий курс, второе издание

A Tour of C++

Second Edition

Bjarne Stroustrup

ADDISON–WESLEY

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Язык программирования C++

Краткий курс, второе издание

Бьярне Страуструп

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
С83
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Страуструп, Бьярне.

С83 Язык программирования C++. Краткий курс, 2-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 320 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-12-5 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Бьярне Страуструп

Язык программирования C++. Краткий курс

2-е издание

Подписано в печать 18.03.2019. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 25,8. Уч.-изд. л. 14,0.
Тираж 1000 экз. Заказ № 2337.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-12-5 (рус.)
ISBN 978-0-13-499783-4 (англ.)

Оглавление

Содержание

Предисловие

Уча, будьте кратки.
— Цицерон

C++ выглядит как совершенно новый язык. Сегодня я могу выразить свои идеи более четко, более просто и более непосредственно, чем мог бы сделать это с помощью C++98. Кроме того, получаемые в результате программы лучше проверяются компилятором и быстрее работают.

В этой книге приведен обзор C++ по состоянию C++17 (текущий стандарт ISO C++, реализованный основными производителями компиляторов C++). Кроме того, в ней упоминаются концепты и модули, определенные в технической спецификации ISO и используемые в настоящее время, но не входящие в стандарт до C++20.

Как и другие современные языки программирования, C++ весьма велик и содержит большое количество библиотек, необходимых для эффективного использования языка. Эта тонкая книга призвана передать опытному программисту идею о том, из чего состоит новейший C++. Она охватывает наиболее фундаментальные возможности языка и компоненты стандартной библиотеки. Книга может быть прочитана всего лишь за несколько часов, но, очевидно, что требуется гораздо больше времени, чтобы научиться писать хорошие программы на C++. Однако цель автора — не сделать программиста мастером, а дать ему обзор языка, привести ключевые примеры и помочь начать работать.

Предполагается, что вы программировали ранее. Если нет, просьба сначала прочесть какой-нибудь учебник наподобие *Программирование: принципы и практика с использованием C++ (второе издание)* [55], прежде чем продолжить чтение данной книги. Но даже если вы программировали ранее, язык, который вы использовали, или приложения, которые вы создавали, могут существенно отличаться от представленного здесь стиля C++.

Представьте себе экскурсию по большому городу, такому как Киев, Москва или Нью-Йорк. В течение всего лишь нескольких часов вы под руководством экскурсовода бросаете беглый взгляд на основные достопримечательности, выслушиваете несколько историй, связанных с каждой из них, и получаете советы, что вам еще следует посмотреть в этом городе. Знаете ли вы город после такой экскурсии? Конечно, нет. Вы не помните и не понимаете все, что видели и слышали. Вы не знаете, как ориентироваться среди официальных и неофициальных правил, которые регулируют жизнь жителей этого города. Чтобы действительно узнать город, вам придется пожить в нем подольше,

зачастую не один год. Однако вы получили определенное, при небольшом везении достаточно полное и объективное представление о том, что же привлекательно в этом городе, и идеи о том, что в нем может представлять для вас интерес. После такой экскурсии можно начинать реальное изучение города.

В ходе экскурсии, в которой экскурсоводом работает данная книга, вам демонстрируются основные языковые возможности C++ и поддерживаемые ими стили программирования, такие как объектно-ориентированное и обобщенное программирование. В ней нет подробного справочного руководства по возможностям языка программирования. В лучших традициях учебников я пытаюсь объяснить каждую возможность, прежде чем ее использовать, но это не всегда возможно, да и не каждый читатель читает книгу строго последовательно. Поэтому читателю рекомендуется использовать перекрестные ссылки и предметный указатель.

Точно так в книге стандартные библиотеки представлены только с точки зрения конкретных примеров, но отнюдь не исчерпывающе. Здесь не описаны библиотеки, которые не определены стандартом ISO. При необходимости читатель может найти необходимый материал самостоятельно. (Примерами таких материалов являются [59] и [60], но в Интернете имеется огромное количество информации (разного качества), например [13]. Когда я упоминаю, например, стандартную библиотечную функцию или класс, вы легко можете найти их определения и изучить документацию по их применению.)

В нашем туре C++ представлен как единое целое, а не как некий слоеный пирог. Поэтому в книге нет деления: вот это — возможности языка, которые доступны в C, вот эта часть доступна в C++98, а это — новинки из стандартов C++11, C++14 или C++17. Такую информацию вы найдете в главе 16, "История и совместимость". Я же сосредоточиваюсь на основах и стараюсь быть насколько можно кратким, но полностью противостоять искушению побольше поговорить о новинках вряд ли сумею. Впрочем, это только на руку тем читателям, кто уже знаком с рядом старых версий C++.

В справочном руководстве или стандарте языка программирования говорится о том, что можно сделать с помощью этого языка, но программисты заинтересованы не просто в том, чтобы использовать язык, но использовать его эффективно и грамотно. Этот аспект частично учтен при подборе охватываемых в книге тем, и в особенности в разделах советов и консультаций. О том, что собой представляет хороший современный C++, можно прочесть в [61]. Базовые рекомендации могут стать хорошей отправной точкой для дальнейшего изучения идей, представленных в этой книге. Вы можете заметить замечательное сходство формулировок (и даже нумерации) рекомендаций из упомянутой книги и книги, которую вы держите в руках. Одна из причин этого

заключается в том, что первое издание данной книги стало также главным исходным материалом при подготовке [61].

Благодарности

Некоторые из представленных здесь материалов взяты из [59], поэтому я благодарю всех, кто помог мне завершить эту книгу.

Спасибо всем, кто помог доработать и исправить первое издание данной книги.

Я также признателен Моргану Стенли (Morgan Stanley) за то, что он дал мне возможность выделить время для написания второго издания книги. Благодаря курсу "Проектирование с использованием C++" в Колумбийском университете удалось выявить множество опечаток и ошибок в раннем варианте этой книги; там же мною было получено много конструктивных предложений по ее улучшению.

Я благодарен Полу Андерсону (Paul Anderson), Чаку Эллисону (Chuck Allison), Питеру Готтшлингу (Peter Gottschling), Уильяму Монсу (William Mons), Чарльзу Уилсону (Charles Wilson) и Сергею Зубкову (Sergey Zubkov) за просмотр книги и внесение предложений по ее улучшению.

Манхэттен, Нью-Йорк

Бьярне Страуструп

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

1

Основы

Первым делом мы перебьем
всех языковых законников.[1]
— Шекспир, "Генрих VI", часть II

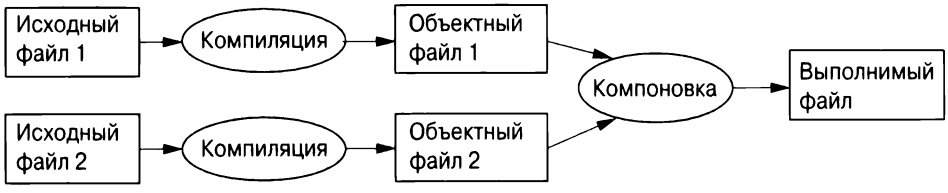
1.1. Введение

В этой главе содержится неформальное представление нотации языка C++, его модель памяти и вычисления, а также основные механизмы организации кода в программе. Все эти языковые средства поддерживают стили программирования, наиболее часто встречающиеся в C и иногда именуемые *процедурным программированием*.

[1] Несколько измененная цитата: у Шекспира упоминаются только законники (без упоминания языков). — *Примеч. пер.*

1.2. Программы

C++ является компилируемым языком. Для работы программы ее исходный текст должен быть обработан с помощью компилятора, который создает объектные файлы, объединяемые компоновщиком в выполнимую программу. Обычно программы на языке C++ состоят из многих файлов с исходными текстами (обычно именуемыми просто *исходными файлами*).



Выполнимая программа создается для определенной комбинации аппаратного обеспечения и операционной системы; ее нельзя просто перенести, скажем, из компьютера Mac в компьютер с Windows. Говоря о *переносимости* программ C++, мы обычно имеем в виду переносимость исходного кода, т.е. исходный код может быть успешно скомпилирован и выполняться в разных системах.

Стандарт ISO C++ определяет два типа сущностей.

- *Фундаментальные возможности языка*, такие как встроенные типы (например, `char` и `int`) или циклы (например, инструкции `for` и `while`).
- *Компоненты стандартных библиотек*, такие как контейнеры (например, `vector` и `map`) или операции ввода-вывода (например, `<<` и `getline()`).

Компоненты стандартной библиотеки представляют собой совершенно обычный код C++, предоставляемый каждой реализацией языка. То есть стандартная библиотека C++ может быть реализована в самом C++ (и реализуется — с очень небольшим использованием машинного кода для таких вещей, как переключение контекста потока). Это означает, что C++ достаточно выразителен и эффективен для самых сложных задач системного программирования.

C++ является статически типизированным языком, т.е. тип каждой сущности (например, объекта, значения, имени или выражения) должен быть известен компилятору в точке использования. Тип объекта определяет набор применимых к нему операций.

1.2.1. Hello, World!

Минимальная программа на C++ выглядит следующим образом:

```
int main() { } // Минимальная программа C++
```

Здесь определена функция `main`, не принимающая аргументов и не выполняющая никаких действий.

Фигурные скобки `{}` группируют выражения в C++. Здесь они указывают начало и конец тела функции. Двойная косая черта `//` начинает комментарий, который продолжается до конца строки. Комментарии предназначены для чтения человеком и полностью игнорируются компилятором.

Каждая программа на C++ должна иметь ровно одну глобальную функцию с именем `main()`. Программа начинается с выполнения этой функции. Целочисленное значение `int`, возвращаемое функцией `main()`, если таковое имеется, является значением, возвращаемым программой "системе". Если значение не возвращается, система получает значение, указывающее на успешное завершение. Ненулевое значение, возвращаемое `main()`, указывает на сбой. Не каждая операционная система и среда выполнения используют это возвращаемое значение: так поступают среды на основе Linux/Unix, но среды на базе Windows делают это редко.

Как правило, программа производит некоторый вывод. Вот программа, которая выводит на консоль текст `Hello, World!`:

```
#include <iostream>
int main()
{
    std::cout << "Hello, World!\n";
}
```

Строка `#include <iostream>` указывает компилятору на необходимость *включения* объявлений стандартных потоковых средств ввода-вывода из файла `iostream`. Без таких объявлений выражение

```
std::cout << "Hello, World!\n"
```

не будет иметь смысла. Оператор << ("вывести") записывает свой второй аргумент в первый. В данном случае строковый литерал `"Hello, World!\n"` записывается в стандартный поток вывода `std::cout`. Строковый литерал представляет собой последовательность символов, окруженную двойными кавычками. В строковом литерале обратная косая черта \, за которой следует еще один символ, обозначает единый "специальный символ". В нашем случае `\n` является символом новой строки, так что после вывода символов, составляющих текст `Hello, World!`, выполняется переход на новую строку.

Префикс `std::` указывает, что имя `cout` находится в пространстве имен стандартной библиотеки (§3.4). Как правило, при обсуждении стандартных функций я оставляю `std::`; в §3.4 показано, как сделать имя видимым без явной квалификации пространства имен.

По сути, весь выполнимый код размещен в функциях и прямо или косвенно вызывается из `main()`. Например:

```
#include <iostream>      // Включение ("импорт") объявлений
                         // библиотеки ввода-вывода
using namespace std;     // Делаем видимыми имена из пространства
                         // имен std без применения std:: (§3.4)

double square(double x) // Функция возведения в квадрат числа с
{                        // плавающей точкой двойной точности
    return x*x;
}

void print_square(double x)
{
    cout << "Квадрат " << x << " равен " << square(x) << "\n";
}

int main()
{
    print_square(1.234); // Вывод: Квадрат 1.234 равен 1.52276
}
```

"Возвращаемый тип" `void` указывает, что функция не возвращает никакого значения.

1.3. Функции

Основной способ получить что-то в программе на C++ — это вызвать соответствующую функцию. Определение функции — это способ указать, как должны быть выполнены необходимые действия. Функция не может быть вызвана, если она не была объявлена ранее.

Объявление функции дает имя функции, тип возвращаемого значения (если таковое имеется), а также количество и типы аргументов, которые должны быть указаны в вызове. Например:

```
Elem*  next_elem();  // Аргументов нет; возвращает
                     // указатель на Elem (Elem*)
void   exit(int);    // Аргумент типа int; не возвращает ничего
double sqrt(double); // Аргумент типа double; возвращает double
```

В объявлении функции возвращаемый тип указывается перед именем функции, а типы аргументов — после имени, заключенными в круглые скобки.

Семантика передачи аргументов идентична семантике инициализации (§3.6.1), т.е. выполняется проверка типов аргументов и при необходимости происходит неявное преобразование типов аргументов (§1.4). Например:

```
double s2 = sqrt(2);       // Вызов sqrt() с аргументом double{2}
double s3 = sqrt("three"); // Ошибка: sqrt() требует
                           // аргумент типа double
```

Значение такой проверки времени компиляции и преобразования типов нельзя недооценивать.

Объявление функции может содержать имена аргументов. Это может помочь читателю программы, но только если объявление не является одновременно определением функции, компилятор просто игнорирует такие имена. Например:

```
double sqrt(double d); // Возвращает квадратный корень от d
double square(double); // Возвращает квадратный корень от аргумента
```

Тип функции включает возвращаемый тип и последовательность типов аргументов. Например:

```
double get(const vector<double>& vec, int index); // Тип функции:
                                // double(const vector<double>&,int)
```

Функция может быть членом класса (§2.3, §4.2.1). Для такой *функции-члена* имя ее класса также является частью типа функции. Например:

```
char& String::operator[](int index); // Тип: char& String::(int)
```

Мы хотим, чтобы наш код был понятным, потому что это первый шаг на пути к его поддерживаемости и сопровождаемости. Первым шагом к облегчению понимания кода является разбиение вычислительных задач на значимые именованные фрагменты (представленные в виде функций и классов). Затем такие функции составляют базовый словарь вычислений, так же как типы (встроенные и определяемые пользователем) составляют базовый словарь данных. Стандартные алгоритмы C++ (например, `find`, `sort` или `iota`) обеспечивают хорошую стартовую позицию (глава 12, "Алгоритмы"). Затем мы можем объединять функции, представляющие распространенные или специализированные задачи, в более крупные вычисления.

Количество ошибок в коде сильно коррелирует с количеством и сложностью кода. Обе проблемы можно решить, используя более короткие функции. Использование функции для выполнения конкретной задачи часто спасает нас от написания определенной части кода в середине другого кода; делая его функцией, мы именуем нашу деятельность и документируем ее зависимости.

Если определены две функции с одним и тем же именем, но с разными типами аргументов, компилятор выберет для каждого вызова наиболее подходящую функцию. Например:

```
void print(int);          // Принимает целочисленный аргумент
void print(double);       // Принимает аргумент с плавающей точкой
void print(string);       // Принимает строковый аргумент
void user()
{
    print(42);            // Вызов print(int)
    print(9.65);          // Вызов print(double)
    print("Barcelona");   // Вызов print(string)
}
```

Если могут быть вызваны две альтернативные функции, но ни одна из них не лучше другой, вызов считается неоднозначным и компилятор выдает сообщение об ошибке. Например:

```
void print(int,double);
void print(double ,int);
void user2()
{
    print(0,0); // Ошибка: неоднозначность
}
```

Определение нескольких функций с одним и тем же именем называется *перегрузкой функций* и является одной из важных частей обобщенного программирования (§7.2). При перегрузке функций каждая функция с одним и тем же именем должна реализовывать одну и ту же семантику. Примером этого являются функции `print()` — каждая функция `print()` выводит свой аргумент.

1.4. Типы, переменные и арифметика

Каждое имя и каждое выражение имеет тип, который определяет, какие операции могут быть над ним выполнены. Например, объявление

```
int inch;
```

гласит, что `inch` имеет тип `int`, т.е. `inch` представляет собой целочисленную переменную.

Объявление представляет собой инструкцию, которая вводит объявляемую сущность в программу. Оно определяет тип этой сущности.

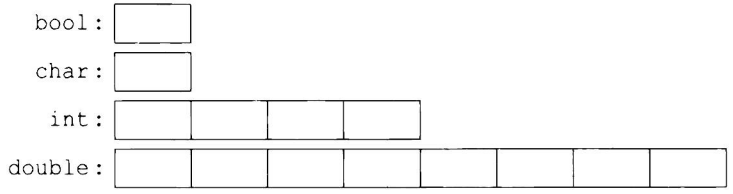
- *Тип* определяет множество возможных значений и множество операций (для объекта).

- *Объект* представляет собой некоторое местоположение в памяти, хранящее значение некоторого типа.
- *Значение* представляет собой множество битов, интерпретируемых в соответствии с типом.
- *Переменная* представляет собой именованный объект.

C++ предлагает небольшой зоопарк фундаментальных типов, но так как я не зоолог, я не буду перечислять их все. Вы можете найти их в справочниках, например в [59], или в Интернете [13]. Например:

```
bool      // Логическое значение - true или false
char      // Символ, например 'a', 'z' или '9'
int       // Целое число, например -273, 42 или 1066
double    // Число с плавающей точкой двойной точности,
          // например -273.15, 3.14 или 6.626e-34
unsigned  // Неотрицательное целое число, например 0, 1 или 999
          // (используется для побитовых логических операций)
```

Каждый фундаментальный тип непосредственно соответствует аппаратным возможностям и имеет фиксированный размер, который определяет диапазон значений, которые могут в нем храниться.



Переменная типа `char` имеет естественный для хранения символа на данной машине размер (обычно это 8-битный байт), а размеры всех прочих типов кратны размеру `char`. Размер типа зависит от реализации (т.е. может меняться от машины к машине) и может быть получен с помощью оператора `sizeof`. Например, `sizeof(char)` равен 1, а `sizeof(int)` часто равен 4.

Числа могут быть с плавающей точкой или целочисленные.

- Числа с плавающей точкой распознаются по наличию десятичной точки (например, `3.14`) или показателю степени (например, `3e-2`).
- Целочисленные литералы — по умолчанию десятичные (например, `42` означает "сорок два"). Префикс `0b` указывает двоичный (по основанию 2) целочисленный литерал (например, `0b10101010`). Префикс `0x` указывает шестнадцатеричный (по основанию 16) целочисленный литерал (например, `0xBAD1234`). Префикс `0` указывает восьмеричный (по основанию 8) целочисленный литерал (например, `0334`).

Чтобы сделать длинные литералы более удобочитаемыми для людей, можно использовать одинарные кавычки (') в качестве разделителя между цифрами. Например, π приближенно равно `3.14159'26535'89793'23846'26433'83279'50288` (или, если вы предпочитаете шестнадцатеричную запись —`0x3.243F'6A88'85A3'08D3`).

1.4.1. Арифметика

Арифметические операторы могут использоваться для создания соответствующих выражений в комбинации с фундаментальными типами:

```
x+y   // Плюс
+x    // Унарный плюс
x-y   // Минус
-x    // Унарный минус
x*y   // Умножение
x/y   // Деление
x%y   // Остаток от деления (для целых чисел)
```

То же возможно и для операторов сравнения:

```
x==y  // Равно
x!=y  // Не равно
x<y   // Меньше
x>y   // Больше
x<=y  // Меньше или равно
x>=y  // Больше или равно
```

Кроме того, имеются логические операторы:

```
x&y   // Побитовое и
x|y   // Побитовое или
x^y   // Побитовое исключающее или
~x    // Побитовое дополнение
x&&y  // Логическое и
x||y  // Логическое или
!x    // Логическое не (отрицание)
```

Побитовый логический оператор дает результат типа операнда, для которого операция была выполнена с каждым битом. Логические операторы `&&` и `||` просто возвращают значение `true` или `false` в зависимости от значений их операндов.

В присваиваниях и арифметических операциях C++ выполняет все необходимые преобразования между фундаментальными типами, так что они могут свободно сочетаться в различных операциях:

```
void some_function() // Функция, не возвращающая значение
{
    double d = 2.2;  // Инициализация числа с плавающей точкой
    int i = 7;       // Инициализация целого числа
```

```
    d = d+i;          // Присваивание суммы переменной d
    i = d*i;          // Присваивание произведения переменной i, с
}                     // обрезанием значения d*i типа double до int
```

Преобразования, используемые в выражениях, называются *обычными арифметическими преобразованиями*, и призваны обеспечить вычисление выражений с точностью, соответствующей наивысшей точности его операндов. Например, сложение типов `double` и `int` вычисляется с помощью арифметики с плавающей точкой двойной точности.

Обратите внимание, что = является оператором присваивания, а оператором проверки на равенство является оператор ==.

В дополнение к обычным арифметическим и логическим операторам C++ предлагает более специфичные операторы для изменения переменных:

```
x+=y   // x = x+y
++x    // Инкремент: x = x+1
x-=y   // x = x-y
--x    // Декремент: x = x-1
x*=y   // Масштабирование: x = x*y
x/=y   // Масштабирование: x = x/y
x%=y   // x = x%y
```

Эти операторы ясны, удобны и очень часто используются.

Порядок вычисления выражений — слева направо, за исключением присваиваний, которые вычисляются справа налево. К сожалению, порядок вычисления аргументов функции не определен.

1.4.2. Инициализация

Прежде чем можно будет использовать объект, ему должно быть присвоено значение. C++ предлагает различные записи для выражения инициализации, например используемую выше запись с =, и универсальную форму, основанную на списках инициализации в фигурных скобках:

```
double d1 = 2.3;          // Инициализация d1 значением 2.3
double d2 {2.3};          // Инициализация d2 значением 2.3
double d3 = {2.3};        // Инициализация d3 значением 2.3
                          // (= при {...} необязателен)
complex<double> z = 1;    // Комплексное число с компонентами double
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2}; // = при {...} необязателен
vector<int> v {1,2,3,4,5,6};  // Вектор значений типа int
```

Применение = традиционно и восходит к C, но если у вас есть сомнения, используйте универсальную форму со списком в фигурных скобках {}. Это по крайней мере спасет вас от преобразований с потерей информации:

```
int i1 = 7.8; // i1 становится равным 7 (вы не удивлены?)
int i2 {7.8}; // Ошибка: преобразование числа с плавающей точкой в целое
```

К сожалению, допускаются и неявно применяются *сужающие преобразования*, которые теряют информацию, такие как `double` в `int` или `int` в `char`. Проблемы, вызываемые неявными сужающими преобразованиями, — это цена, заплаченная за совместимость с C (§16.3).

Константы (§1.6) не могут быть оставлены неинициализированными, а переменные могут оставаться неинициализированными лишь в крайне редких случаях. Не вводите в программу имя, пока у вас не будет для него подходящего значения. Определяемые пользователем типы (такие, как `string`, `vector`, `Matrix`, `Motor_controller` или `Orc_warrior`) могут быть определены как неявно инициализируемые (§4.2.1).

При определении переменной вам не нужно явно указывать тип, если он может быть выведен из инициализатора:

```
auto b  = true;     // bool
auto ch = 'x';      // char
auto i  = 123;      // int
auto d  = 1.2;      // double
auto z  = sqrt(y);  // z имеет тип, возвращаемый функцией sqrt(y)
auto bb {true};     // bb имеет тип bool
```

При использовании `auto` мы склонны применять =, поскольку не имеется никакого потенциально проблематичного преобразования типов, но если вы предпочитаете последовательно использовать только { }, то можете это делать.

Мы используем `auto` там, где у нас нет конкретной причины упоминать тип явно. "Конкретные причины" включают следующее.

- Определение находится в большой области видимости, где хотелось бы, чтобы тип был четко виден читателям кода.
- Мы хотим точно указать диапазон или точность переменной (например, `double`, а не `float`).

Используя `auto`, мы избегаем избыточности и записи длинных имен типов. Это особенно важно в обобщенном программировании, в котором определение точного типа объекта может быть трудным для программиста, а имена типов могут быть довольно длинными (§12.2).

1.5. Область видимости и время жизни

Объявление вносит объявляемое имя в область видимости.

- *Локальная область видимости*: имя, объявленное в функции (§1.3) или лямбда-выражении (§6.3.2), называется *локальным именем*. Его область видимости простирается от точки объявления до конца блока, в котором

находится объявление. *Блок* определяется парой фигурных скобок { }. Имена аргументов функции рассматриваются как локальные имена.

- *Область видимости класса*: имя называется *именем члена* (или *именем члена класса*), если оно определено в классе (§2.2, §2.3, глава 4, "Классы"), вне любой функции (§1.3), лямбда-выражения (§6.3.2) или перечисления `enum class` (§2.5). Его область видимости простирается от открывающей фигурной скобки { охватывающего объявления до конца этого объявления.
- *Область видимости пространства имен*: имя называется *именем члена пространства имен*, если оно определено в пространстве имен (§3.4) вне любой функции, лямбда-выражения (§6.3.2), класса (§2.2, §2.3, глава 4, "Классы") или перечисления `enum class` (§2.5). Его область видимости простирается от точки объявления до конца его пространства имен.

Имя, объявленное вне прочих конструкций, называется *глобальным именем* и считается находящимся в *глобальном пространстве имен*.

Кроме того, могут существовать объекты без имени, такие как временные объекты и объекты, создаваемые с использованием оператора `new` (§4.2.2). Например:

```
vector<int> vec; // vec – глобальное имя (глобальный вектор int)

struct Record
{
    string name; // name является членом Record (член-строка)
    // ...
};

void fct(int arg) // fct – глобальное имя (глобальная функция)
                  // arg – локальное имя (аргумент функции)
{
    string motto {"Who dares wins"}; // motto – локальное имя
    auto p = new Record{"Hume"};     // p указывает на безымянный
                                     // объект Record (создан new)
    // ...
}
```

Объект должны быть построен (инициализирован), прежде чем он будет использован, а при выходе из области видимости — уничтожен. Для объекта пространства имен точкой его уничтожения является конец программы. Для члена точка уничтожения определяется моментом уничтожения объекта, членом которого он является. Объект, созданный с использованием оператора `new`, "живет" до тех пор, пока не будет уничтожен оператором `delete` (§4.2.2).

1.6. Константы

C++ поддерживает два понятия неизменяемости.

- `const`: грубо говоря, это означает "я обещаю не изменять это значение". Это ключевое слово используется главным образом для указания интерфейсов, так что данные могут передаваться в функции с использованием указателей и ссылок, не боясь, что они будут модифицированы. Компилятор заставляет выполнять обещание, сделанное с помощью `const`. Значение `const` может быть вычислено во время выполнения.
- `constexpr`: грубо говоря, это означает "вычисляется во время компиляции". Это ключевое слово используется главным образом для определения констант, разрешая размещение данных в памяти, предназначенной только для чтения (где они вряд ли окажутся повреждены), и для повышения производительности. Значение `constexpr` должно быть вычислено компилятором.

Например:

```
constexpr int dmv = 17;             // dmv — именованная константа
int var = 17;                       // var — не константа
const double sqv = sqrt(var);       // sqv — именованная константа,
                                    // возможно, вычисленная
                                    // во время выполнения
double sum(const vector<double>&); // Функция sum не изменяет
                                    // свой аргумент (§1.7)
vector<double> v {1.2, 3.4, 4.5};  // v не является константой
const double s1 = sum(v);           // OK: sum(v) вычисляется
                                    // во время выполнения
constexpr double s2 = sum(v);       // Ошибка: sum(v) — не
                                    // константное выражение
```

Чтобы функция могла использоваться в *константном выражении*, т.е. в выражении, которое будет вычисляться компилятором, она должна быть определена как `constexpr`. Например:

```
constexpr double square(double x) { return x*x; }

// OK: 1.4*square(17) — константное выражение:
constexpr double max1 = 1.4*square(17);

// Ошибка: var — не константное выражение:
constexpr double max2 = 1.4*square(var);

// OK: может быть вычислено во время выполнения:
const     double max3 = 1.4*square(var);
```

Функция, объявленная как constexpr, может применяться к неконстантным аргументам, но когда это происходит, результат не является константным выражением. Функция, объявленная как constexpr, может вызываться с аргументами, не являющимися константными выражениями, в контекстах, которые не требуют константных выражений. Таким образом, нам не приходится определять, по сути, одну и ту же функцию дважды: один раз — для константных выражений и второй — для переменных.

Чтобы быть объявленной как constexpr, функция должна быть достаточно простой, не должна иметь побочных действий и может использовать только информацию, переданную ей в качестве аргументов. В частности, она не может изменять нелокальные переменные, но может содержать циклы и использовать собственные локальные переменные. Например:

```
constexpr double nth(double x, int n) // Предполагается, что 0 <= n
{
    double res = 1;
    int i = 0;
    while (i<n)
    {   // Цикл while: выполняется, пока истинно его условие (§1.7.1)
        res*=x;
        ++i;
    }
    return res;
}
```

В ряде мест константные выражения требуются правилами языка (например, для границ массивов (§1.7), в метках case (§1.8), аргументах — значениях шаблонов (§6.2) и константах, объявленных с использованием constexpr). В других случаях вычисления времени компиляции оказываются важными с точки зрения производительности программы. Независимо от проблем с производительностью понятие неизменности (объекта с неизменным состоянием) является важной концепцией проектирования.

1.7. Указатели, массивы и ссылки

Наиболее фундаментальной коллекцией данных является смежно расположенная последовательность элементов одного и того же типа, именуемая *массивом*. Фундаментально это именно то, что может предоставить аппаратное обеспечение. Массив элементов типа char может быть объявлен следующим образом:

```
char v[6]; // Массив из 6 символов
```

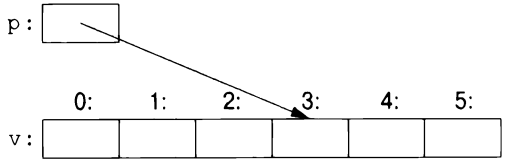
Аналогично указатель может быть объявлен следующим образом:

```
char*p;    // Указатель на символ
```

В объявлениях [] означает "массив из", а * — "указатель на". Все массивы в качестве нижней границы имеют 0, поэтому упомянутый выше массив v имеет шесть элементов — от v[0] до v[5]. Размер массива должен быть константным выражением (§1.6). Переменная указателя может хранить адрес объекта соответствующего типа:

```
char*p = &v[3]; // p указывает на четвертый элемент v
char x = *p;    //*p - объект, на который указывает p
```

В выражении унарный префикс * означает "содержимое", а & — "адрес". Результат показанного выше инициализированного определения графически может быть представлен следующим образом.



Рассмотрим копирование десяти элементов из одного массива в другой:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                 // Чтобы стать копией v1,
    for (auto i=0; i!=10; ++i) // копируем элементы
        v2[i]=v1[i];
    // ...
}
```

Эта инструкция for может быть прочтена как "установить i равным нулю; пока i не равно 10, копировать i-й элемент и увеличить i". При применении к целочисленной переменной или к переменной с плавающей точкой оператор инкремента ++ просто добавляет 1. C++ предлагает также более простую инструкцию for, которая называется циклом for для диапазона, для простейшего обхода последовательности:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto x : v)      // Для каждого x из v
        cout << x << '\n';
    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

Первый цикл для диапазона можно прочитать как "для каждого элемента v от первого до последнего поместить его копию в x и распечатать ее". Об-

ратите внимание, что не нужно указывать границы массива, когда мы инициализируем его списком. Цикл `for` для диапазона может использоваться для любой последовательности элементов (§12.1).

Если мы не хотим копировать значения из `v` в переменную `x` и хотим, чтобы вместо этого `x` была ссылкой, ссылающейся на элемент, можем написать:

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto& x : v) // Добавление 1 к каждому x из v
        ++x;
    // ...
}
```

Здесь в объявлении унарный суффикс `&` означает "ссылка на". Ссылка похожа на указатель, за исключением того, что вам не нужно использовать префикс `*` для доступа к значению, на которое указывает ссылка. Кроме того, после инициализации *ссылка не может быть перенацелена* (не может ссылаться на другой объект).

Ссылки особенно полезны для указания аргументов функции. Например:

```
void sort(vector<double>& v); // Сортировка v (v — вектор double)
```

Используя ссылку, мы гарантируем, что при вызове `sort(my_vec)` мы не копируем `my_vec` и что будет отсортирован именно `my_vec`, а не его копия.

Если мы не хотим изменять аргумент, но при этом хотим избежать затрат на копирование, мы используем константную ссылку (§1.6). Например:

```
double sum(const vector<double>&)
```

Функции, принимающие константные ссылки, весьма распространены.

При использовании в объявлениях операторы (такие, как `&`, `*` или `[]`) называются *операторами деклараторов*:

```
T a[n] // T[n]: a является массивом из n элементов T
T*p    // T*: p является указателем на T
T& r   // T&: r является ссылкой на T
T f(A) // T(A): f — функция, принимающая аргумент типа A
       // и возвращающая результат типа T
```

1.7.1. Нулевой указатель

Мы пытаемся обеспечить, чтобы указатель всегда указывал на объект, так что его можно разыменовывать. Если у нас нет такого объекта или если нужно представить понятие "нет объекта" (например, для конца списка), мы даем указателю значение `nullptr` ("нулевой указатель"). Существует только единственный `nullptr`, используемый всеми типами указателей:

```
double* pd = nullptr;         // Указатель на double
Link<Record>* lst = nullptr; // Указатель на Link<Record>
int x = nullptr;              // Ошибка: nullptr - указатель, а не int
```

Зачастую оказывается разумной проверка, указывает ли аргумент-указатель на что-либо:

```
int count_x(const char* p, char x) // Подсчет количества вхождений
{   // x в p[]. Предполагается, что p указывает на массив char
    // с завершающим нулевым символом (или не указывает ни на что)
    if (p==nullptr)
        return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

Обратите внимание, как мы можем перемещать указатель с использованием оператора ++ так, чтобы он указывал на следующий элемент массива, и что мы можем убрать инициализатор в инструкции `for`, если он нам не нужен.

Определение `count_x()` предполагает, что `char*` является строкой в стиле языка программирования C, т.е. указатель указывает на массив с завершающим нулевым символом. Символы в строковом литерале неизменяемы, поэтому для возможности вызова `count_x("Hello!")` я объявил функцию `count_x()` с аргументом `const char*`.

В более старом коде вместо `nullptr` обычно используется `0` или `NULL`. Однако применение `nullptr` устраняет потенциальную путаницу между целыми числами (такими, как `0` или `NULL`) и указателями (такими, как `nullptr`).

В примере с `count_x()` мы не используем инициализирующую часть цикла `for`, поэтому можем использовать более простой цикл `while`:

```
int count_x(const char* p, char x) // Подсчет количества вхождений
{   // x в p[]. Предполагается, что p указывает на массив char
    // с завершающим нулевым символом (или не указывает ни на что)
    if (p==nullptr)
        return 0;
    int count = 0;
    while (*p)
    {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

Инструкция while выполняется до тех пор, пока условие не станет ложным.

Проверка числового значения (например, while(*p) в count_x()) эквивалентна сравнению значения с 0 (например, while(*p!=0)). Проверка значения указателя (например, if(p)) эквивалентна сравнению значения с nullptr (например, if(p!=nullptr)).

"Нулевой ссылки" не существует. Ссылка должна ссылаться на действительный объект (и реализации предполагают, что это так и есть). Есть запутанные и хитрые способы нарушить это правило, но так поступать не следует.

1.8. Проверки

C++ предоставляет обычный набор операторов для выражения выбора и циклов, таких как инструкции if, switch, while и for. Например, вот простая функция, которая выполняет запрос ввода пользователя и возвращает логическое значение, указывающее его ответ:

```
bool accept()
{
    cout << "Работаем дальше (y или n)?\n";  // Вывод запроса
    char answer = 0;  // Инициализация значением, которого не
                      // может быть во вводе пользователя
    cin >> answer;    // Чтение ответа пользователя
    if (answer == 'y')
        return true;
    return false;
}
```

Оператору вывода << соответствует оператор >> для ввода; cin — переменная стандартного потока ввода (глава 10, "Ввод и вывод"). Тип правого операнда оператора >> определяет, какой вход принимается, а сам правый операнд является целевым объектом операции ввода, принимающим вводимую информацию. Символ \n в конце выводимой строки является символом новой строки (§1.2.1).

Обратите внимание, что определение answer появляется там, где оно необходимо (а не до того). Объявление может находиться везде, где может находиться инструкция языка.

Пример можно было бы улучшить, если учесть ответ n (для "нет"):

```
bool accept2()
{
    cout << "Работаем дальше (y или n)?\n";  // Вывод запроса
    char answer = 0;  // Инициализация значением, которого не
                      // может быть во вводе пользователя
    cin >> answer;    // Чтение ответа пользователя
    switch (answer)
    {
```

```
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "Считаю это ответом \"нет\".\n";
        return false;
    }
}
```

Инструкция `switch` сравнивает значение с набором констант. Эти константы, именуемые метками `case`, должны быть различными, а если проверяемое значение не равно ни одной из них, выбирается метка `default`. Если проверяемое значение не равно ни одной из меток `case`, а метка `default` отсутствует, не выполняются никакие действия.

Мы не обязаны выходить из `case`-метки с помощью оператора `return` для выхода из функции, содержащей инструкцию `switch`. Зачастую мы хотим продолжать работу, выполняя инструкцию, следующую за инструкцией `switch`. Это можно сделать, применив инструкцию `break`. В качестве примера рассмотрим примитивный анализатор тривиальных команд видеоигры:

```
void action()
{
    while (true) {
        cout << "Действие:\n"; // Запрос действия
        string act;
        cin >> act;            // Чтение символов в строку
        Point delta {0,0};     // Point хранит пару {x,y}
        for (char ch : act) {
            switch (ch) {
            case 'u':          // Вверх (up)
            case 'n':          // Север (north)
                ++delta.y;
                break;
            case 'r':          // Вправо (right)
            case 'e':          // Восток (east)
                ++delta.x;
                break;
            // ... Другие действия ...
            default:
                cout << "Заморожен!\n";
            }
            move(current+delta*scale);
            update_display();
        }
    }
}
```

Так же, как и инструкция `for` (§1.7), инструкция `if` может вводить переменную и тестировать ее. Например:

```
void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0)
    {
        // ... Попадаем сюда, если n!=0 ...
    }
    // ...
}
```

Здесь целое значение `n` определяется для использования внутри инструкции `if`, инициализируется значением `v.size()` и тут же, после точки с запятой, проверяется условие `n!=0`. Имя, объявленное в условии, находится в области видимости обеих ветвей инструкции `if`.

Как и для инструкции `for`, цель объявления имени в условии инструкции `if` заключается в том, чтобы ограничить область видимости переменной, улучшить читаемость и минимизировать ошибки.

Наиболее распространенный случай — проверка переменной на равенство нулю (или `nullptr`). Для этого просто удалите явное упоминание о сравнении. Например:

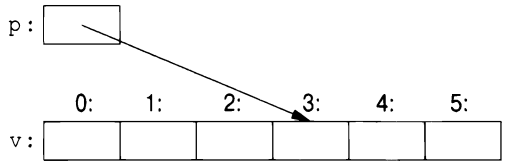
```
void do_something(vector<int>& v)
{
    if (auto n = v.size())
    {
        // ... Оказываемся здесь, если n!=0 ...
    }
    // ...
}
```

Предпочтительно использование по возможности этой более краткой и простой формы записи.

1.9. Отображение на аппаратные средства

C++ предлагает непосредственное сопоставление с оборудованием. Когда вы используете одну из базовых операций, реализация представляет собой то, что предлагает оборудование (как правило, отдельную машинную операцию). Например, сложение двух `int`, `x+y`, выполняет машинную команду целочисленного сложения.

Реализация C++ рассматривает память компьютера как последовательность ячеек памяти, в которых можно размещать (типизированные) объекты и адресовать их с помощью указателей.



Указатель представлен в памяти как машинный адрес, поэтому числовое значение p на этом рисунке равно 3. Если изображение кажется вам похожим на массив (§1.7), то это потому, что массив является базовой абстракцией C++ для "непрерывной последовательности объектов в памяти".

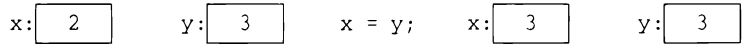
Простое сопоставление фундаментальных языковых конструкций с аппаратными средствами имеет решающее значение для высокой производительности на низком уровне, которой десятилетиями славятся C и C++. Основная модель машины в C и C++ основана на компьютерном оборудовании, а не на некоторой математической абстракции.

1.9.1. Присваивание

Присваивание встроенных типов представляет собой простую машинную операцию копирования. Взгляните:

```
int x = 2;
int y = 3;
x = y;     // x становится равным 3
           // Примечание: x==y
```

Это очевидно. Графически это можно представить следующим образом.

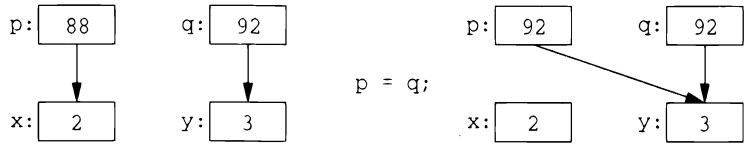


Обратите внимание, что эти два объекта независимы. Мы можем изменить значение y, не влияя на значение x. Например, присваивание x=99 не изменит значение y. В отличие от Java, C# и других языков (но подобно языку C) это верно для всех типов, а не только для int.

Если мы хотим, чтобы разные объекты ссылались на одно и то же (совместно используемое) значение, мы должны это указать. Так, мы могли бы использовать указатели:

```
int  x = 2;
int  y = 3;
int* p = &x;
int* q = &y; // Теперь p!=q и *p!=*q
p = q;       // p становится равным &y; теперь p==q,
             // так что (очевидно) *p == *q
```

Графически это можно представить следующим образом.

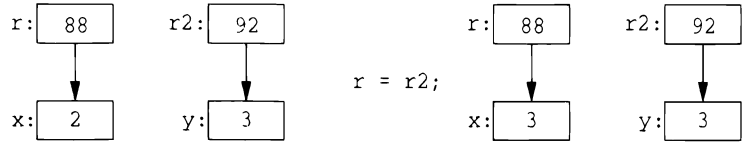


Я произвольно выбрал значения 88 и 92 в качестве адресов int. Здесь мы видим, что целевой объект присваивания получает значение от исходного объекта, что дает нам два независимых объекта (в данном случае — два указателя) с одинаковым значением. То есть присваивание p=q приводит к p==q. После присваивания p=q оба указателя указывают на переменную y.

И ссылка, и указатель ссылаются/указывают на объект, и оба они представлены в памяти в виде машинного адреса. Однако языковые правила их использования различны. Присвоение ссылки изменяет *не то, на что ссылается ссылка* (т.е. ссылка не перенацеливается на другой объект), а сам объект, на который она ссылается:

```
int x = 2;
int y = 3;
int& r = x;    // r ссылается на x
int& r2 = y;   // r2 ссылается на y
r = r2;        // Чтение через r2, запись через r:
               // x становится равным 3
```

Графически это можно представить следующим образом.



Для обращения к значению, на которое указывает указатель, используется оператор разыменования *; для ссылок это делается автоматически (неявно).

После присваивания x=y для всех встроенных типов выполняется равенство x==y (как и для корректно спроектированных пользовательских типов (глава 2, "Пользовательские типы"), которые предоставляют операторы присваивания = и сравнения на равенство ==.

1.9.2. Инициализация

Инициализация отличается от присваивания. В общем случае, чтобы присваивание работало правильно, целевой объект присваивания должен иметь значение. С другой стороны, задачей инициализации является превращение неинициализированного фрагмента памяти в корректный объект. Для почти всех типов результат чтения или записи неинициализированной переменной не определен. Для встроенных типов это наиболее очевидно для ссылок:

```
int x = 7;
int& r {x}; // Привязка r к x (r ссылается на x)
r = 7;      // Присваивание тому, на что ссылается r
int& r2;    // Ошибка: неинициализированная ссылка
r2 = 99;    // Присваивание тому, на что ссылается r2
```

К счастью, невозможно получить неинициализированную ссылку — в противном случае присваивание `r2=99` сохраняло бы `99` в некотором неопределенном месте в памяти; в конечном итоге это могло бы привести к неверным результатам или краху программы.

Можно использовать для инициализации ссылки оператор =, но не дайте ему вас запутать! Например:

```
int& r = x; // Привязка r к x (r ссылается на x)
```

Это все еще инициализация, привязывающая `r` к `x`, а не копирование значения.

Различие между инициализацией и присваиванием имеет решающее значение и для многих пользовательских типов, таких как `string` и `vector`, где целевой объект присваивания владеет ресурсом, который в конечном итоге должен быть освобожден (§5.3).

Базовая семантика передачи аргументов и возврата значения из функции представляет собой инициализацию (§3.6). Например, именно так мы получаем возможность передачи аргументов по ссылке.

1.10. Советы

Приведенные здесь советы в основном представляют собой подмножество советов из *C++ Core Guidelines* [61]. Ссылки на них выглядят как [CG:ES.23], что означает 23-е правило из раздела *Expressions and Statement*. Обычно советы по ссылке более подробны, с обоснованием и примерами.

[1] Не паникуй! Со временем все станет понятнее; §1.1; [CG: In.0].

[2] Не используйте встроенные возможности сами по себе. Основные (встроенные) возможности обычно лучше всего используются косвенно, через библиотеки, такие как стандартная библиотека ISO C++ (главы 8–15); [CG:P.10].

[3] Для написания хороших программ не требуется знание каждой мелочи языка программирования.

[4] Сосредоточьтесь на методах программирования, а не на возможностях языка программирования.

[5] Новейшие возможности языка приведены в стандарте ISO C++; §16.1.3; [CG:P.2].

[6] "Упаковывайте" значимые операции в тщательно именованные функции; §1.3; [CG:F.1].

[7] Функция должна выполнять единственную логическую операцию; §1.3 [CG:F.2].

[8] Функции должны быть краткими; §1.3; [CG:F.3].

[9] Используйте перегрузку, когда функции выполняют концептуально одинаковые задачи с разными типами; §1.3.

[10] Если может потребоваться вычисление функции во время компиляции, объявите ее как `constexpr`; §1.6; [CG:F.4].

[11] Следует понимать, как примитивы языка отображаются на аппаратное обеспечение; §1.4, §1.7, §1.9, §2.3, §4.2.2, §4.4.

[12] Применение разделителей между цифрами числа делает литералы более удобочитаемыми; §1.4; [CG:NL.11].

[13] Избегайте усложненных выражений; [CG:ES.40].

[14] Избегайте сужающих преобразований; §1.4.2; [CG:ES.46].

[15] Минимизируйте область видимости переменной; §1.5.

[16] Избегайте "магических констант"; используйте символические константы; §1.6; [CG:ES.45].

[17] Предпочитайте неизменяемые данные; §1.6; [CG:P.10].

[18] Объявляйте в каждом объявлении (только) одно имя; [CG:ES.10].

[19] Поддерживайте часто используемые и локальные имена короткими, а редкие и нелокальные — длинными; [CG:ES.7].

[20] Избегайте одинаково выглядящих имен; [CG:ES.8].

[21] Избегайте имен `ПРОПИСНЫМИ_БУКВАМИ`; [CG:ES.9].

[22] Предпочитайте синтаксис инициализации с использованием фигурных скобок `{}` для объявлений с именованными типами; §1.4; [CG:ES.23].

[23] Используйте `auto`, чтобы избежать повторения имен типов; §1.4.2; [CG:ES.11].

[24] Избегайте неинициализированных переменных; §1.4; [CG:ES.20].

[25] Поддерживайте области видимости малыми; §1.5; [CG:ES.5].

[26] При объявлении переменной в условии инструкции `if` предпочитайте версии с неявной проверкой на равенство `0`; §1.8.

[27] Применяйте `unsigned` только для работы с битами; §1.4; [CG:ES.101], [CG:ES.106].

[28] Применение указателей должно быть простым и понятным; §1.7; [CG:ES.42].

[29] Работая с указателями, используйте `nullptr` вместо `0` или `NULL`; §1.7; [CG:ES.47].

[30] Не объявляйте переменную до тех пор, пока у вас не будет значения для ее инициализации; §1.7, §1.8; [CG:ES.21].

[31] Не пишите в комментариях то, что очевидным образом выражено в коде; [CG:NL.1].

[32] Указывайте в комментариях свои намерения; [CG:NL.2].

[33] Поддерживайте согласованный стиль отступов; [CG:NL.4].

2

Пользовательские типы

— Не паникуй!
Дуглас Адамс

- **Введение**
- **Структуры**
- **Классы**
- **Объединения**
- **Перечисления**
- **Советы**

2.1. Введение

Мы называем типы, которые могут быть построены из фундаментальных типов (§1.4), модификатора `const` (§1.6) и операторов деклараторов (§1.7), *встроенными типами*. C++ имеет богатый набор встроенных типов и операций, который сознательно ограничен низким уровнем. Они непосредственно и эффективно отражают возможности обычной компьютерной техники, но при этом не предоставляют программисту средства высокого уровня для создания сложных приложений. Вместо этого C++ дополняет встроенные типы и операции сложным набором *механизмов абстракции*, на основе которых программисты могут построить необходимые возможности высокого уровня.

Механизмы абстракции C++ разработаны прежде всего для того, чтобы позволить программистам проектировать и реализовывать их собственные типы с соответствующими представлениями и операциями и изящно использовать такие типы в своих приложениях. Типы, построенные из других типов с помощью механизмов абстракции C++, называют *пользовательскими типами* (или типами, определенными пользователями). Таковыми типами являются *классы* и *перечисления*. Пользовательские типы могут быть построены как из встроенных типов, так и из других пользовательских типов. Большая часть этой книги посвящена дизайну, реализации и использованию пользовательских типов. Пользовательские типы зачастую оказываются предпочтительнее встроенных, потому что их легче использовать, они менее подвержены ошибкам и

обычно так же эффективны для выполнения действий, для которых они предназначены, как и встроенные типы (или оказываются еще более быстрыми).

Оставшаяся часть этой главы представляет простейшие и наиболее фундаментальные средства определения и использования типов. Главы 4–7 содержат более полное описание механизмов абстракции и стилей программирования, которые они поддерживают. Главы 8–15 представляют обзор стандартной библиотеки, а так как стандартная библиотека главным образом состоит из пользовательских типов, эти главы предоставляют примеры того, что может быть построено с помощью языковых средств и методов программирования, представленных в главах 1–7.

2.2. Структуры

Первым шагом в создании нового типа часто является организация необходимых ему элементов в структуре данных `struct`:

```
struct Vector {
    int sz;         // Количество элементов
    double* elem;   // Указатель на элементы
};
```

Эта, первая, версия `Vector` состоит из `int` и `double*`.

Переменная типа `Vector` может быть определена следующим образом:

```
Vector v;
```

Однако само по себе это не слишком полезно, потому что указатель `elem` в `v` не указывает ни на что. Для того чтобы объект был полезным, у объекта `v` должны быть некоторые элементы, на которые он мог бы указывать. Например, мы можем построить `Vector` следующим образом:

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s]; // Выделение памяти для s double
    v.sz = s;
}
```

То есть элемент `elem` объекта `v` получает значение указателя, созданного оператором `new`, а член `sz` объекта `v` получает количество элементов. Знак `&` в `Vector&` указывает, что мы передаем `v` как неконстантную ссылку (§1.7). Таким образом, функция `vector_init()` может модифицировать переданный ей вектор.

Оператор `new` выделяет память из области, именуемой *свободной памятью* (известной также как *динамическая память* или *куча*). Выделенные в свободной памяти объекты не зависят от области видимости, в которой они

создаются, и продолжают существовать до тех пор, пока не будут уничтожены с помощью оператора `delete` (§4.2.2).

Простое применение `Vector` имеет следующий вид:

```
double read_and_sum(int s) // Читает s целых чисел из cin и
{                          // возвращает их сумму; предполагается,
    Vector v;              // что значение s положительно
    vector_init(v,s);      // Выделение памяти для s элементов в v
    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];    // Чтение элементов
    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];    // Вычисление суммы элементов
    return sum;
}
```

Нам нужно пройти долгий путь до того, как наш `Vector` станет таким же элегантным и гибким, как `vector` из стандартной библиотеки. В частности, пользователь `Vector` должен знать каждую деталь представления `Vector`. В оставшейся части этой главы и в следующих двух происходит постепенное улучшение `Vector`, как пример языковых возможностей и технологий. В главе 11, "Контейнеры", представлен `vector` стандартной библиотеки, который содержит множество улучшений.

Я использую `vector` и другие компоненты стандартной библиотеки как примеры

- для иллюстрации возможностей языка программирования и методов проектирования и
- чтобы помочь изучить и использовать компоненты стандартной библиотеки.

Не следует изобретать велосипед и компоненты стандартной библиотеки наподобие `vector` или `string`; просто воспользуйтесь ими.

Для обращения к членам `struct` посредством имени (или ссылки) мы используем точку (`.`), а для обращения через указатель — оператор `->`. Например:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;   // Обращение посредством имени
    int i2 = rv.sz;  // Обращение посредством ссылки
    int i3 = pv->sz; // Обращение посредством указателя
}
```

2.3. Классы

У определения данных отдельно от операций над ними есть определенные преимущества, такие как возможность использовать данные произвольным

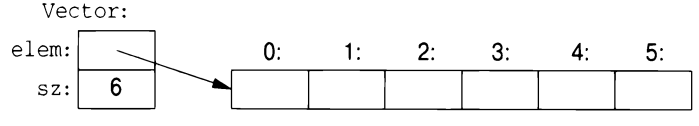
образом. Однако для пользовательского типа необходима более тесная связь между представлением данных и операциями над ними, обеспечивающая все свойства, ожидаемые от "реального типа". В частности, часто требуется сохранить представление данных недоступным для пользователей, что упрощает использование типа, гарантирует согласованное использование данных и позволяет внести улучшения в представление позже. Чтобы сделать это, мы должны различать интерфейс типа (который общедоступен) и его реализацию (которая имеет доступ к недоступным иным образом данным). Соответствующий языковый механизм называют *классом*. У класса есть ряд членов, которые могут быть данными, функциями или членами-типами. Интерфейс определяется открытыми членами класса (объявленными как `public`), а закрытые (`private`) члены доступны только через интерфейс. Например:

```
class Vector
{
public:
  Vector(int s) :elem{new double[s]},sz{s}{} // Конструирование Vector
  double& operator[](int i){return elem[i];} // Обращение по индексу
  int size() { return sz; }
private:
  double*elem; // Указатель на элементы
  int    sz;   // Количество элементов
};
```

При этом можно определить переменную нашего нового типа `Vector` следующим образом:

```
Vector v(6); // Vector с 6 элементами
```

Графически объект `Vector` можно представить следующим образом.



В принципе объект `Vector` является "дескриптором", содержащим указатель на элементы (`elem`) и количество элементов (`sz`). Количество элементов (в примере — 6) может меняться от объекта к объекту `Vector`, а сам объект в разное время может иметь различное количество элементов (§4.2.3). Однако сам объект `Vector` всегда имеет один и тот же размер. Это основной способ обработки различного количества информации в C++: дескриптор фиксированного размера, ссылающийся на переменное количество данных "в другом месте" (например, в области памяти, выделенной с помощью оператора `new`; §4.2.2). Как создавать и использовать такие объекты — основная тема главы 4, "Классы".

Здесь представление вектора (члены `elem` и `sz`) доступно только через интерфейс, предоставляемый с помощью открытых членов: `Vector()`, `operator[]()` и `size()`. Пример `read_and_sum()` из §2.2 упрощается:

```
double read_and_sum(int s)
{
    Vector v(s);    // Создание вектора из s элементов
    for(int i=0; i!=v.size(); ++i)
        cin>>v[i];  // Чтение элементов
    double sum = 0;
    for(int i=0; i!=v.size(); ++i)
        sum+=v[i]; // Вычисление суммы элементов
    return sum;
}
```

"Функция"-член с тем же именем, что и имя класса, называется *конструктором*, т.е. функцией, используемой для конструирования (создания) объектов класса. Таким образом, конструктор `Vector()` заменяет функцию `vector_init()` из §2.2. В отличие от обычной функции конструктор гарантированно используется для инициализации объектов своего класса. Таким образом, определение конструктора решает проблему неинициализированных переменных класса.

`Vector(int)` определяет, как будут создаваться объекты типа `Vector`. В частности, он указывает, что для этого требуется целое число. Это целое число используется в качестве количества элементов. Конструктор инициализирует члены `Vector`, используя список инициализаторов членов:

```
:elem{new double[s]}, sz{s}
```

То есть сначала выполняется инициализация `elem` указателем на память для `s` элементов типа `double`, полученную из свободной памяти. Затем выполняется инициализация `sz` значением `s`.

Обращение к элементам выполняется с помощью функции индекса, именуемой `operator[]`. Она возвращает ссылку на соответствующий элемент (`double&`, которая позволяет читать и записывать этот элемент).

Функция `size()` возвращает пользователям количество хранящихся элементов.

Очевидно, что обработка ошибок здесь полностью отсутствует, но мы вернемся к этому позже, в §3.5. Точно так же мы не предоставляем механизм "возврата" в свободную память массива `double`, полученного с помощью оператора `new`; в §4.2.2 показано, как элегантно это сделать с использованием деструктора.

Между структурой и классом нет никакой принципиальной разницы; `struct` — это просто класс с членами, открытыми (`public`) по умолчанию.

Например, вы можете определить конструкторы и другие функции-члены для структуры.

2.4. Объединения

Объединение (union) представляет собой структуру (struct), в которой все члены располагаются по одному и тому же адресу, так что union занимает столько же памяти, сколько и его наибольший член. Естественно, union может хранить одновременно значение только одного члена. Например, рассмотрим запись таблицы символов, которая хранит имя и значение. Значение может иметь тип либо Node*, либо int:

```
enum Type { ptr, num }; // Type может хранить значения ptr и num (§2.5)

struct Entry
{
    string name;        // string – тип стандартной библиотеки
    Type t;
    Node* p;            // Используем p, если t==ptr
    int i;              // Используем i, если t==num
};

void f(Entry* pe)
{
    if (pe->t == num)
        cout << pe->i;
    // ...
}
```

Члены p и i никогда не используются одновременно, так что получается пустая трата памяти. Ее легко избежать, указав, что оба члена являются членами union:

```
union Value
{
    Node* p;
    int   i;
};
```

Язык не отслеживает, какие значения хранятся в объединении, так что это должен делать программист :

```
struct Entry
{
    string name;
    Type t;
    Value v;    // Используем v.p, если t==ptr; и v.i, если t==num
};
```

```
void f(Entry* pe)
{
    if (pe->t == num)
        cout << pe->v.i;
    // ...
}
```

Поддержание соответствия между *полем типа* (здесь — `t`) и типом, содержащимся в объединении, чревато ошибками. Чтобы избежать ошибок, можно обеспечить это соответствие, инкапсулируя объединение и поле типа в класс и предлагая доступ только через функции-члены, которые гарантируют корректное использование объединения. На уровне приложений абстракции, основанные на таких *маркированных объединениях* (tagged unions), являются достаточно распространенными и полезными. Использование же "голых" объединений лучше всего свести к минимуму.

Тип стандартной библиотеки `variant` может использоваться для устранения большинства прямых применений объединений. `variant` сохраняет значение одного из множества альтернативных типов (§13.5.1). Например, `variant<Node*,int>` может содержать либо `Node*`, либо `int`.

С помощью `variant` пример `Entry` может быть записан следующим образом:

```
struct Entry
{
    string name;
    variant<Node*,int> v;
};

void f(Entry* pe)
{
    if (holds_alternative<int>(pe->v)) // *pe хранит int? (§13.5.1)
        cout << get<int>(pe->v);       // Получаем этот int
    // ...
}
```

Для множества применений `variant` проще и безопаснее, чем `union`.

2.5. Перечисления

В дополнение к классам C++ поддерживает простую разновидность пользовательских типов, для которой мы можем перечислить значения:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Обратите внимание, что перечислители (например, red) находятся в области видимости своего enum class, так что они могут повторно использоваться в разных enum class без какой-либо путаницы. Например, Color::red является перечислителем red в перечислении Color, который отличается от Traffic_light::red.

Перечисления используются для представления небольших множеств целочисленных значений. Они применяются с целью сделать код более удобочитаемым и менее подверженным ошибкам, чем в ситуации, когда такие символические (и мнемонические) имена не использовались бы.

class после enum указывает, что перечисление строго типизировано и что его перечислители имеют ограниченную область видимости. Будучи отдельными типами, перечисления enum class помогают предотвратить случайное злоупотребление константами. В частности, мы не можем смешивать значения Traffic_light и Color:

```
Color x = red;                // Ошибка: какой red?
Color y = Traffic_light::red; // Ошибка: этот red не из Color
Color z = Color::red;         // OK
```

Точно так же мы не можем неявно смешивать Color и целочисленные значения:

```
int i = Color::red; // Ошибка: Color::red не является int
Color c = 2;        // Ошибка инициализации: 2 не является Color
```

Перехват попытки преобразования в enum является хорошей защитой от ошибок, но часто мы хотим инициализировать enum значением его базового типа (которым по умолчанию является int), так что это допускается как явное преобразование из базового типа:

```
Color x = Color{5}; // OK, но длинно
Color y {6};        // Также OK
```

По умолчанию в enum class определены только присваивание, инициализация и сравнения (например, == и <; §1.4). Однако *перечисление является пользовательским типом*, так что мы можем определить для него операторы:

```
Traffic_light& operator++(Traffic_light&t) // Префиксный инкремент ++
{
    switch (t) {
        case Traffic_light::green:  return t=Traffic_light::yellow;
        case Traffic_light::yellow: return t=Traffic_light::red;
        case Traffic_light::red:    return t=Traffic_light::green;
    }
}

Traffic_light next = ++light;      // next == Traffic_light::green
```

Если вы не хотите явно квалифицировать имена перечислителей и хотите, чтобы их значения были целыми числами (без необходимости явного преобразования), можете удалить слово `class` из `enum class` и получить "обычный" `enum`. Перечислители такого "обычного" `enum` находятся в той же области видимости, что и имя их перечисления, и неявно преобразуются в свои целые значения. Например:

```
enum Color { red, green, blue };
int col = green;
```

Здесь `col` получает значение `1`. По умолчанию целочисленные значения счетчиков начинаются с `0` и увеличиваются на единицу для каждого очередного перечислителя. "Обычное" перечисление было в C++ (и C) с самых первых дней, так что несмотря на то, что они менее хорошо себя ведут, они широко распространены в современном коде.

2.6. Советы

[1] Предпочитайте хорошо определенные пользовательские типы встроенным, если последние оказываются слишком низкоуровневыми; §2.1.

[2] Организовывайте связанные данные в структуры (`struct` или `class`); §2.2; [CG:C.1].

[3] Представляйте различие между интерфейсом и реализацией с помощью `class`; §2.3; [CG:C.3].

[4] `struct` представляет собой просто `class`, все члены которого по умолчанию являются `public`; §2.3.

[5] Определите конструкторы для гарантии и простоты инициализации классов; §2.3; [CG:C.2].

[6] Избегайте "голых" объединений; заворачивайте их в класс вместе с полем типа; §2.4; [CG:C.181].

[7] Используйте перечисления для представления множеств именованных констант; §2.5; [CG:Enum.2].

[8] Предпочитайте перечисления `class enum` "обычным" перечислениям `enum` для минимизации неожиданностей; §2.5; [CG:Enum.3].

[9] Определите операции над перечислениями для безопасного и простого использования; §2.5; [CG:Enum.4].

3

Модульность

— Не прерывайте меня,
пока я прерываю вас.

Уинстон Черчилль

3.1. Введение

Программа на языке программирования C++ состоит из множества отдельно разработанных частей, таких как функции (§1.2.1), пользовательские типы (глава 2, "Пользовательские типы"), иерархии классов (§4.5) и шаблоны (глава 6, "Шаблоны"). Ключом к управлению всеми этими частями является четкое определение взаимодействия между ними. Первым и самым важным шагом является разграничение интерфейса части и ее реализации. На уровне языка C++ представляет интерфейсы с помощью объявлений. *Объявление* определяет все, что необходимо для использования функции или типа. Например:

```
double sqrt(double); // Функция вычисления квадратного корня,
                     // получающая double и возвращающая double

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;    // elem указывает на массив из sz double
    int sz;
};
```

Ключевым моментом здесь является то, что тела функций, т.е. их *определения*, находятся "где-то в ином месте". В данном примере нам может хотеться, чтобы представление `Vector` тоже было "в другом месте", но мы вернемся к этому вопросу позже (говоря об абстрактных типах в §4.3). Определение `sqrt()` выглядит следующим образом:

```
double sqrt(double d) // Определение sqrt()
{
    // ... алгоритм вычисления квадратного корня ...
}
```

В случае `Vector` мы должны определить все три функции-члена:

```
Vector::Vector(int s)              // Определение конструктора
    :elem{new double[s]}, sz{s}    // Инициализация членов
{
}

double& Vector::operator[](int i) // Определение оператора индекса
{
    return elem[i];
}

int Vector::size()                // Определение size()
{
    return sz;
}
```

Мы должны определить функции-члены `Vector`, но не функцию `sqrt()`, так как она является частью стандартной библиотеки. Однако реальной разницы нет: библиотека просто является некоторым кодом, который мы применяем в своих программах и который написан с использованием тех же языковых возможностей, которыми пользуемся мы.

Может иметься несколько объявлений некоторой сущности, такой как функция, но определение должно быть только одно.

3.2. Раздельная компиляция

C++ поддерживает понятие раздельной компиляции, когда пользовательский код видит только объявления типов и функций. Определения этих типов и функций содержатся в отдельных исходных файлах и скомпилированы отдельно. Это разрешает организацию программы в виде набора полунезависимых фрагментов кода. Такое разделение может использоваться для минимизации времени компиляции и строгого разделения логически различных частей программы (таким образом минимизируя вероятность ошибок). Библиотека часто представляет собой сборник отдельных скомпилированных фрагментов кода (например, функций).

Как правило, мы размещаем объявления, определяющие интерфейс модуля, в файле с именем, указывающим его предполагаемое применение. Например:

```
// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;  // elem указывает на массив из sz double
    int sz;
};
```

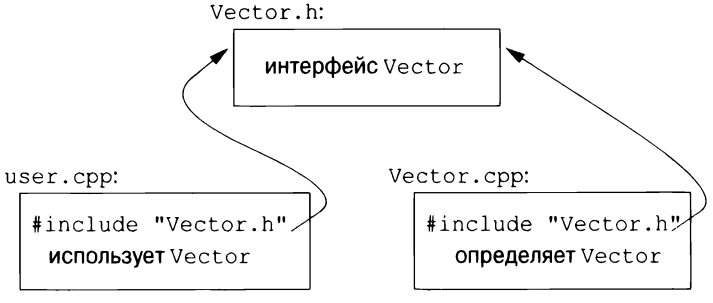
Это объявление размещено в файле `Vector.h`. Пользователи *включают* этот заголовочный файл для доступа к данному интерфейсу. Например:

```
// user.cpp:

#include "Vector.h"  // Интерфейс Vector
#include <cmath>     // Интерфейс математических функций
                     // стандартной библиотеки, включая sqrt()

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for(int i=0; i!=v.size(); ++i)
    sum+=std::sqrt(v[i]); // Сумма квадратных корней
    return sum;
}
```

Чтобы помочь компилятору обеспечить согласованность, `.cpp`-файл с реализацией `Vector` также включает `.h`-файл с его интерфейсом:

```
// Vector.cpp:

#include "Vector.h"                 // Интерфейс Vector

Vector::Vector(int s)               // Определение конструктора
    :elem{new double[s]}, sz{s}     // Инициализация членов
{
}

double& Vector::operator[](int i) // Определение оператора индекса
{
    return elem[i];
}

int Vector::size()                // Определение size()
{
    return sz;
}
```

Код в `user.cpp` и `Vector.cpp` совместно использует информацию интерфейса `Vector`, представленную в `Vector.h`, но в остальном эти два файла независимы и могут быть скомпилированы отдельно. Графически фрагменты программы могут быть представлены следующим образом.



Строго говоря, использование раздельной компиляции не является языковым вопросом; это вопрос о том, как лучше всего использовать конкретную реализацию языка. Однако он имеет большое практическое значение. Наилучший подход к организации программы состоит в том, чтобы рассматривать программу как набор модулей с точно определенными зависимостями, логически представить модульность с помощью языковых средств, а затем использовать физическую модульность с помощью файлов для эффективной раздельной компиляции.

Файл `.cpp`, компилируемый сам по себе (включая файлы `.h`, которые он включает с помощью директивы `#include`), именуется *единицей трансляции*. Программа может состоять из многих тысяч единиц трансляции.

3.3. Модули (C++20)

Использование директивы включения файлов `#include` имеет долгую историю, чревато ошибками и представляет собой довольно дорогостоящий способ составления программ из частей. Если вы примените директиву `#include header.h` в 101 единице трансляции, текст `header.h` будет обработан компилятором 101 раз. Если вы включаете `#include header1.h` до `header2.h`, то объявления и макросы в `header1.h` могут влиять на смысл кода в `header2.h`. Если же вы включаете `#include header2.h` до `header1.h`, то уже `header2.h` может влиять на код в `header1.h`. Очевидно, что это решение далеко от идеального и на самом деле является основным источником стоимости компиляции и ошибок с 1972 года, когда этот механизм был впервые введен в C.

И вот, наконец, мы надеемся получить лучший способ выражения физических модулей на C++. Возможность языка программирования, именуемая модулями, пока еще не является частью стандарта ISO C++, но уже входит в техническую спецификацию ISO [34]. Имеются готовые реализации, и поэтому я рискую рекомендовать модули в этой книге, несмотря на то что, вероятнее всего, детали этой возможности изменятся, и может пройти несколько лет до того, как каждый сможет использовать ее в своем производственном коде. Старый код (имеется в виду код с `#include`) может "жить" еще в течение очень долгого времени, поскольку соответствующее обновление может быть очень дорогостоящим и трудоемким.

Рассмотрим, как выразить пример с `Vector` и `sqrt_sum()` из §3.2 с использованием модулей:

```
// Файл Vector.cpp:

module;   // Здесь определен модуль

// ... указываем все, что может потребоваться реализации Vector ...
export module Vector;   // Определение модуля с именем "Vector"

export class Vector
{
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;       // elem указывает на массив из sz double
    int sz;
};
```

```
Vector::Vector(int s)
    :elem{new double[s]}, sz{s}   // Инициализация членов
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}

export int size(const Vector& v) { return v.size(); }
```

Здесь определен модуль с именем `Vector`, который экспортирует класс `Vector`, все его функции-члены, а также свободную функцию `size()`.

Способ применения модулей состоит в их импорте при необходимости. Например:

```
// Файл user.cpp:
import Vector;    // Получение интерфейса Vector
#include <cmath>  // Интерфейс математических функций, включая sqrt()
double sqrt_sum(Vector& v)
{
    double sum = 0;
    for(int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]); // Сумма квадратных корней
    return sum;
}
```

Я мог бы импортировать и математические функции стандартной библиотеки, но я использовал старомодный `#include`, чтобы показать, что вы можете смешивать старые и новые директивы в одной программе. Такое смешение необходимо для постепенного перевода старого кода с `#include` на `import`.

Различие между заголовочными файлами и модулями не просто синтаксическое.

- Модуль компилируется только один раз (а не в каждой единице трансляции, в которой он используется).
- Два модуля могут импортироваться в любом порядке без изменения их смысла.
- Если вы импортируете нечто в модуль, то пользователи этого модуля не получают неявный доступ к тому, что вы импортируете (и импортированные сущности никак не влияют на вашу работу): `import` не является транзитивным.

Влияние модулей на надежность и производительность во время компиляции может быть просто захватывающим.

3.4. Пространства имен

В дополнение к функциям (§1.3), классам (§2.3) и перечислениям (§2.5) C++ предлагает *пространства имен* в качестве механизма для выражения того факта, что некоторые объявления связаны одно с другим и что их имена не должны конфликтовать с другими именами. Например, я могу захотеть поэкспериментировать с собственным типом комплексных чисел (§4.2.1, §14.4):

```
namespace My_code {
    class complex
    {
        // ...
    };

    complex sqr t(complex);

    // ...

    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}
```

Помещая свой код в пространство имен `My_code`, я гарантирую, что мои имена не конфликтуют с именами стандартной библиотеки (в пространстве имен `std`) (§3.4). Эта предосторожность разумна, поскольку стандартная библиотека обеспечивает поддержку арифметики комплексных чисел (§4.2.1, §14.4).

Простейший способ получить доступ к имени в другом пространстве имен — квалифицировать его с помощью имени пространства имен (например, `std::cout` или `My_code::main`). "Настоящая" функция `main()` определена в глобальном пространстве имен, т.е. не является локальной для некоторого пространства имен, класса или функции.

Если многократная квалификация имени становится утомительной или отвлекающей, можно ввести имя в область видимости с помощью using-объявления:

```
void my_code(vector<int>& x, vector<int>& y)
{
    using std::swap;  // Использование swap из стандартной библиотеки
    // ...
    swap(x,y);        // Вызов std::swap()
    other::swap(x,y); // Некоторая другая swap()
    // ...
}
```

Объявление using делает имя из пространства имен используемым так же, как если бы оно было объявлено в области видимости, в которой к нему выполняется обращение. Действие директивы using std::swap таково, как если бы функция swap была объявлена в my_code().

Для получения доступа ко всем именам в пространстве имен стандартной библиотеки можно использовать следующую директиву:

```
using namespace std;
```

Директива using делает неквалифицированные имена из именованного пространства имен доступными из области видимости, в которую мы поместили эту директиву. Поэтому после директивы using для std мы можем писать просто cout, а не std::cout. Используя директиву using, мы теряем способность выборочно использовать имена из указанного пространства имен, поэтому ее следует использовать осторожно, как правило, для библиотеки, которая широко применяется в приложении (например, std), или при переделке приложения, которое не использовало пространства имен.

Пространства имен в основном используются для организации более крупных программных компонентов, таких как библиотеки. Они упрощают составление программы из отдельно разработанных частей.

3.5. Обработка ошибок

Обработка ошибок — это большая и сложная тема со своими проблемами и последствиями, которые выходят далеко за пределы языковых средств в методы и инструменты программирования. Однако C++ предоставляет несколько возможностей, которые могут нам помочь. Основным инструментом является сама система типов. Вместо того чтобы тщательно составлять свои приложения на основе встроенных типов (например, char, int или double) и инструкций (например, if, while или for), мы строим пользовательские типы (такие, как string, map или regex) и алгоритмы (такие, как sort(),

`find_if()` и `draw_all()`), подходящие для наших целей и наших приложений. Такие конструкции более высокого уровня упрощают программирование, ограничивают возможности внесения ошибок (например, вы вряд ли попытаетесь применить алгоритм обхода дерева к диалоговому окну) и увеличивают способность компилятора к обнаружению ошибок. Большинство конструкций языка C++ предназначены для проектирования и реализации изящных и эффективных абстракций (например, пользовательских типов и алгоритмов, их использующих). Одним из результатов такой абстракции является то, что точка, в которой может быть обнаружена ошибка времени выполнения, отделена от точки, в которой она может обрабатываться. По мере роста программ, в особенности при широком использовании библиотек, особую важность приобретают стандарты обработки ошибок, так что сформулировать стратегию обработки ошибок еще на ранней стадии разработки программы — неплохая идея.

3.5.1. Исключения

Вновь рассмотрим наш пример с использованием `Vector`. Что *должно* быть сделано, если мы пытаемся получить доступ к элементу, который находится вне диапазона вектора из §2.3?

- Автор `Vector` не знает, что пользователь собирался сделать в этом случае (автор обычно даже не знает, в какой программе будет работать его творение).
- Пользователь `Vector` не может последовательно обнаруживать данную проблему (если бы он мог это делать, обращения за пределами диапазона не было бы вовсе).

В предположении, что обращение за пределами диапазона является ошибкой, после которой мы хотим восстановить работу нашего приложения, создатель `Vector` может принять следующее решение: он обнаруживает попытку обращения к элементу вне доступного диапазона и сообщает об этом пользователю. Затем пользователь может предпринять необходимые действия по обработке ошибки и восстановлению. Например, `Vector::operator[]()` может обнаружить попытку такого обращения и сгенерировать исключение `out_of_range`:

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Инструкция `throw` передает управление обработчику исключений типа `out_of_range` в некоторой функции, которая непосредственно или опосредованно вызывает `Vector::operator[]()`. Для этого реализация *сворачивает* стек вызовов функций так, чтобы вернуться в контекст этой вызывающей функции (т.е. механизм обработки исключений выходит из областей видимости и функций, чтобы вернуться в вызывающую функцию, которая выразила заинтересованность в обработке исключения данного вида, вызывая при этом все необходимые деструкторы (§4.2.2). Например:

```
void f(Vector& v)
{
    // ...
    try   // Исключения в этом блоке обрабатываются
    {     // определенным ниже обработчиком
        v[v.size()] = 7;        // Попытка обращения за границами v
    }
    catch (out_of_range& err) // Ошибка out_of_range!
    {
        // ... Обработка ошибки выхода за границы диапазона ...
        cerr << err.what() << '\n';
    }
    // ...
}
```

Мы помещаем код, в обработке исключений в котором мы заинтересованы, в блок `try`. Попытка присваивания `v[v.size()]` приводит к обращению за границами массива, поэтому управление будет передано в предоставленный блок `catch` с обработчиком исключений типа `out_of_range`. Тип `out_of_range` определен в стандартной библиотеке (в заголовочном файле `<stdexcept>`) и используется рядом функций обращения к контейнерам стандартной библиотеки.

Исключение перехватывается по ссылке, чтобы избежать копирования, и в нем используется функция `what()` для вывода сообщения об ошибке.

Использование механизмов обработки исключений может сделать обработку ошибок более простой, более систематической и более удобочитаемой. Чтобы достичь этого, не злоупотребляйте конструкциями `try`. Основной метод сделать обработку ошибок простой и систематической называется *захват ресурса является инициализацией* (Resource Acquisition Is Initialization — RAII) и объясняется в §4.2.2. Основная идея RAII заключается в том, чтобы все ресурсы, необходимые для работы класса, получал конструктор, а деструктор освобождал все захваченные ресурсы, тем самым гарантируя (неявное) освобождение ресурсов.

Функция, которая никогда не должна генерировать исключения, может быть объявлена как `noexcept`. Например:

```
void user(int sz) noexcept
{
    Vector v(sz);
    iota(&v[0],&v[sz],1); // Заполняет v значениями 1,2,3,4... (§14.3)
    // ...
}
```

Если все благие намерения остались лишь намерениями и функция `user()` по-прежнему генерирует исключение, вызывается функция `std::terminate()`, которая немедленно прекращает выполнение программы.

3.5.2. Инварианты

Использование исключений для сигнализации о выходе за границы массива является примером функции, проверяющей свои аргументы и отказывающейся работать, когда не выполняется ее основное *предусловие*. Если бы мы формально описывали оператор индекса `Vector`, мы бы указали что-то наподобие "индекс должен быть в диапазоне `[0:size())`", и это фактически и есть то условие, которое мы проверяем в нашей реализации `operator[]()`. Обозначение `[a:b)` означает полуоткрытый диапазон, в котором `a` является частью диапазона, а `b` — нет. Всякий раз, определяя функцию, мы должны рассмотреть, каковы ее предусловия и следует ли их тестировать (§3.5.3). Для большинства приложений проверка простых инвариантов является хорошей идеей; см. также §3.5.4.

Однако `operator[]()` работает с объектами типа `Vector`, и ничто из того, что он делает, не имеет смысла, пока элементы `Vector` не будут иметь "разумные" значения. В частности, мы говорили, что "`elem` указывает на массив из `sz` элементов типа `double`", но это было сказано только в комментарии. Такое утверждение о том, что предполагается истинным для класса, называется *инвариантом класса* или просто *инвариантом*. Задача конструктора — установить инвариант своего класса (так, чтобы функции-члены могли полагаться на то, что он выполняется), а задача функций-членов — гарантировать выполнение инварианта после их завершения. К сожалению, наш конструктор `Vector` только частично выполнил свою работу. Он правильно инициализировал элементы `Vector`, но не убедился, что переданные ему аргументы имели смысл. Что будет при следующем вызове?

```
Vector v(-27);
```

Похоже, это приведет к неприятностям.

Вот более надежное определение:

```
Vector::Vector(int s)
{
    if (s<0)
        throw length_error{"Vector: отрицательный размер"};
```

```
    elem = new double[s];
    sz = s;
}
```

Для сообщения о некорректном числе элементов я использую исключение `length_error` стандартной библиотеки, потому что некоторые операции стандартной библиотеки используют его для сообщения о подобных проблемах. Если оператор `new` не может выделить необходимую память, он генерирует исключение `std::bad_alloc`. Теперь мы можем написать:

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error& err) {
        // Обработка отрицательного размера
    }
    catch (std::bad_alloc& err) {
        // Обработка исчерпания памяти
    }
}
```

Вы можете определить собственные классы, которые будут использоваться в качестве исключений, и заставить их переносить любую информацию из точки, где обнаружена ошибка, в точку, в которой она может быть обработана (§3.5.1).

Часто функция не имеет возможности завершить назначенную ей задачу после генерации исключения. В таком случае "обработка" исключения означает выполнение минимальной локальной очистки и повторную генерацию того же исключения. Например:

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error&) { // Некоторые действия и
                                 // повторная генерация исключения
        cerr << "Сбой test(): ошибка размера вектора\n";
        throw;                   // Повторная генерация
    }
        catch (std::bad_alloc&){ // Наша программа не в состоянии об-
                                 // работать ошибку исчерпания памяти
        std::terminate();        // Завершение программы
    }
}
```

В тщательно спроектированном коде блоки `try` редки. Избегайте излишних блоков `try`, систематически используя идиому RAII (§4.2.2, §5.3).

Понятие инвариантов занимает центральное место в разработке классов, а предусловия играют аналогичную роль в разработке функций. Инварианты

- помогают точно понять, чего мы хотим;
- заставляют быть точными; это дает нам больше шансов получить правильный код (конечно, после отладки и тестирования).

Понятие инвариантов лежит в основе концепции управления ресурсами C++, поддерживаемой конструкторами (глава 4, "Классы") и деструкторами (§4.2.2, §13.2).

3.5.3. Альтернативные варианты обработки ошибок

Обработка ошибок — важная проблема в реальном программировании, поэтому имеется множество различных подходов к ее решению. Если обнаружена ошибка, которая не может быть обработана локально в функции, то функция должна каким-то образом передать информацию о проблеме некоторой вызывающей функции. Генерация исключения — наиболее общий механизм C++ для этого.

Существуют языки, в которых исключения предназначены просто для предоставления альтернативного механизма возврата значений. C++ таким языком не является: исключения предназначены для сообщения о невозможности выполнить задание. Исключения интегрированы с конструкторами и деструкторами для обеспечения согласованной схемы обработки ошибок и управления ресурсами (§4.2.2, §5.3). Компиляторы оптимизированы таким образом, чтобы вернуть значение было намного дешевле, чем сгенерировать то же значение в виде исключения.

Генерация исключения — не единственный способ сообщить об ошибке, которая не может быть обработана локально. Функция может указать, что она не в состоянии выполнить поставленную перед ней задачу, разными путями:

- генерируя исключение;
- возвращая некоторое значение, указывающее на ошибку;
- завершая работу программы (вызывая функцию наподобие `exit()`, `terminate()` или `abort()`).

Индикатор ошибки ("код ошибки") возвращается в следующих случаях.

- Когда ошибка — нормальное, ожидаемое явление. Например, вполне нормальной является ошибка открытия файла (возможно, такого файла нет или он не может быть открыт из-за проблем с правами доступа).

- Ожидается, что непосредственная вызывающая функция в состоянии обработать такую ошибку.

Исключения генерируются в следующих ситуациях.

- Ошибка настолько редкая, что программист, скорее всего, забудет выполнить ее проверку. Например, когда вы в последний раз проверяли значение, возвращаемое функцией `printf()`?
- Ошибка не может быть обработана непосредственной вызывающей функцией. Вместо этого ошибка должна быть передана конечной вызывающей функции. Например, невозможно, чтобы каждая функция приложения надежно обрабатывала все ошибки выделения памяти или обрыва соединения сети.
- В модулях более низкого уровня в приложении могут быть добавлены новые типы ошибок, при том что модули более высокого уровня не были написаны с учетом необходимости обработки таких ошибок (например, если бывшее однопоточным приложение было изменено для работы с несколькими потоками или с удаленными ресурсами с доступом по сети).
- Отсутствует подходящий путь возврата для кодов ошибок. Например, конструктор не имеет возвращаемого значения, которое могла бы проверить вызывающая функция[1]. Кроме того, конструкторы могут быть вызваны для нескольких локальных переменных или в частично сконструированном сложном объекте, так что очистка на основе кодов ошибок окажется довольно сложной.
- Путь возврата значения из функции оказывается более сложным или дорогостоящим из-за необходимости возвращать как значение, так и индикатор ошибки (например, в виде `pair`; §13.4.3), что может привести к использованию лишних параметров и нелокальных указателей ошибки или к другим обходным путям.
- Ошибка должна быть передана цепочке вызовов "конечной вызывающей функции". Многократная проверка кода ошибки утомительна, дорогостояща и чревата ошибками.
- Восстановление после ошибки зависит от результатов нескольких вызовов функций, что ведет к необходимости поддерживать локальное состояние между вызовами и к усложненным управляющим структурам.

[1] Возможность возврата кода ошибки из конструктора через параметр-ссылку (или указатель), а также через глобальную переменную сути дела не меняет. — *Примеч. ред.*

- Функция, обнаружившая ошибку, является функцией обратного вызова (функциональный аргумент), так что непосредственная вызывающая функция может даже не знать, что функция была вызвана.
- Ошибка требует выполнения некоторой "отмены действий".

Программа завершается в следующих случаях.

- Восстановление после ошибки данного вида невозможно. Например, во многих — но не во всех — системах нет разумного способа восстановления после исчерпания доступной памяти.
- В данной системе обработка нетривиальных ошибок требует перезапуска потока, процесса или компьютера.

Одним из способов гарантировать завершение работы приложения является добавление `noexcept` к функции, с тем чтобы генерация исключения с помощью `throw` в любом месте реализации функции приводила к вызову `terminate()`. Обратите внимание, что существуют приложения, для которых безусловное завершение работы неприемлемо, поэтому необходимо использовать альтернативные варианты.

К сожалению, перечисленные ситуации не всегда оказываются логически непересекающимися и простыми в применении. Имеют значение такие факторы, как размер и сложность программы. Иногда по мере развития приложения происходит изменение приоритетов и компромиссов. Для принятия верного решения требуется опыт. Если у вас есть сомнения, предпочитайте исключения, потому что их использование лучше масштабируется и не требует внешних инструментов для проверки того, что все ошибки обработаны.

Не верьте, что все коды ошибок или все исключения плохи; для каждого из механизмов есть своя достаточно точно определенная ниша. Кроме того, не верьте мифу о том, что обработка исключений плохо влияет на производительность; зачастую она быстрее, чем корректная обработка сложных или редких условий ошибки, а также повторных проверок кодов ошибок.

Идиома RAII (§4.2.2, §5.3) имеет жизненное значение для простой и эффективной обработки ошибок с использованием исключений. Код, переполненный `try`-блоками, зачастую просто является отражением худших аспектов стратегий обработки ошибок на основе кодов ошибок.

3.5.4. Контракты

В настоящее время нет общего и стандартного способа написания дополнительных тестов времени выполнения для инвариантов, предусловий и т.п. Для включения в C++20 предложен механизм контракта [20, 21]. Его цель состоит в том, чтобы поддержать пользователей, которые для обеспечения корректной работы программы хотели бы положиться на тестирование с дли-

тельными проверками времени выполнения, но при развертывании производственной версии кода получить код с минимальными проверками. Такой подход популярен в высокопроизводительных приложениях в организациях, которые в своей работе опираются на обширную систематическую проверку.

До сих пор для этого приходилось использовать различные механизмы, разработанные отдельно для каждого конкретного случая. Например, можно воспользоваться макросом командной строки для управления проверкой времени выполнения:

```
double& Vector::operator[](int i)
{
    if (RANGE_CHECK && (i<0 || size()<=i))
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Стандартная библиотека предлагает отладочный макрос `assert()` для проверки выполнения условий во время выполнения, например:

```
void f(const char* p)
{
    assert(p!=nullptr); // p не должен быть равен nullptr
    // ...
}
```

Если условие `assert()` не выполняется в "отладочном режиме", программа завершается. Если программа работает не в отладочном режиме, `assert()` не выполняет никаких проверок. Это очень грубая и негибкая возможность, которой, тем не менее, часто оказывается вполне достаточно.

3.5.5. Статические проверки

Исключения сообщают об ошибках, обнаруженных во время выполнения. Если можно найти ошибку во время компиляции, это обычно гораздо предпочтительнее (вот почему так важна система типов и средства определения интерфейсов пользовательских типов). Кроме того, можно выполнить некоторые простые проверки свойств, известных во время компиляции, и сообщить о сбоях в виде сообщений компилятора об ошибках. Например:

```
static_assert(4<=sizeof(int), "int слишком мал"); // Проверка размера
```

Этот код выведет сообщение `int слишком мал`, если во время компиляции не будет выполнено условие `4<=sizeof(int)` (т.е. если `int` в этой системе не будет иметь размер как минимум 4 байта). Такие инструкции мы называем статическими утверждениями.

Механизм `static_assert` может использоваться для чего угодно, что может быть выражено в терминах константных выражений (§1.6). Например:

```
constexpr double C = 299792.458;    // км/с
void f(double speed)
{
    // 160 км/ч == 160.0/(60*60) км/с
    constexpr double local_max = 160.0/(60*60);

    // Ошибка: speed должна быть константой
    static_assert(speed<C, "Такая скорость запрещена физикой");

    static_assert(local_max < C, "Недостижимая скорость"); // ОК
    // ...
}
```

В общем случае `static_assert(A,S)` выводит сообщение `S` в виде сообщения компилятора об ошибке, если условие `A` ложно. Если вы не хотите выводить специальный текст сообщения об ошибке, уберите `S` — и компилятор выведет сообщение по умолчанию:

```
static_assert(4<=sizeof(int)); // Выводит сообщение по умолчанию
```

Сообщение по умолчанию обычно содержит информацию о местоположении `static_assert` в исходном тексте и символьное представление проверяемого предиката.

Наиболее важное применение `static_assert` — когда мы делаем те или иные предположения о типах, используемых в обобщенном программировании (§7.2, §13.9).

3.6. Аргументы и возвращаемые значения функций

Основной и рекомендуемый способ передачи информации из одной части программы в другую — через вызов функции. Информация, необходимая для выполнения задачи, передается в качестве аргументов функции, а полученные результаты передаются обратно как возвращаемые значения. Например:

```
int sum(const vector<int>& v)
{
    int s = 0;
    for (const int i : v)
        s += i;
    return s;
}

vector fib = {1,2,3,5,8,13,21};
int x = sum(fib); // x становится равным 53
```

Существуют и другие пути передачи информации между функциями, такие как глобальные переменные (§1.5), указатели и ссылки в качестве параме-

тров (§3.6.1) и совместно используемое состояние в объекте класса (глава 4, "Классы"). Глобальные переменные крайне не рекомендуются к применению как известный источник ошибок, а разделяемое состояние обычно должно использоваться только функциями, совместно реализующими точно определенную абстракцию (например, функциями-членами класса, §2.3).

Учитывая важность передачи информации в функции и из них, неудивительно, что имеется множество способов ее осуществления. Ключевыми вопросами при этом являются следующие.

- Объект копируется или совместно используется?
- Если объект совместно используется, является ли он изменяемым?
- Если объект перемещается, остается ли после этого "пустой объект" (§5.2.2)?

Поведение по умолчанию для передачи аргументов в функцию и возврата значений из функций — "копирование" (§1.9), но некоторые копирования могут быть неявно оптимизированы и превращены в перемещения.

В примере `sum()` результирующий `int` копируется из `sum()`, но копировать потенциально очень большой вектор в `sum()` было бы неэффективно и бессмысленно, поэтому аргумент в эту функцию передается по ссылке (на что указывает символ `&`; §1.7).

У функции `sum()` нет никаких причин изменять свой аргумент. Эта неизменяемость указывается путем объявления аргумента `vector` как `const` (§ 1.6), так что `vector` передается в функцию посредством константной ссылки.

3.6.1. Передача аргументов

Сначала рассмотрим, как передать значение в функцию. По умолчанию мы выполняем копирование ("передача по значению"), но если мы хотим ссылаться на объект в среде вызывающей функции, то используем ссылку ("передача по ссылке"). Например:

```
void test(vector<int> v, vector<int>& rv) // v передается по значению;
{                                         // rv передается по ссылке
    v[1]  = 99; // Изменение v (локальная переменная)
    rv[2] = 66; // Изменение того, на что ссылается rv
}

int main()
{
    vector fib = {1,2,3,5,8,13,21};
    test(fib,fib);
    cout << fib[1] << ' ' << fib[2] << '\n'; // Выводит 2 66
}
```

Когда нас интересует производительность, мы обычно передаем малые значения по значению, а более крупные — по ссылке. Здесь "маленький" означает "что-то, что действительно дешево копируется". Точное значение термина "малый" зависит от архитектуры машины; хорошим эмпирическим правилом оказывается "малый размер не превышает размера двух-трех указателей".

Если мы хотим передавать аргументы по ссылке по соображениям производительности, но нам не нужно изменять аргумент, мы передаем константную ссылку, как это сделано в примере `sum()`. В обычном хорошем коде это, безусловно, наиболее распространенный случай: он быстр и не подвержен ошибкам.

Не так уж редко аргумент функции имеет значение по умолчанию, т.е. значение, которое считается предпочтительным или наиболее распространенным. Мы можем указать такое значение с помощью *аргумента функции по умолчанию*. Например:

```
void print(int value, int base =10); // Вывод значения в системе
                                     // счисления с основанием base
print(x,16);                         // Шестнадцатеричный вывод
print(x,60);                         // Шестидесятеричный (шумерский)
print(x);                            // По умолчанию - десятичный
```

Вот более простая альтернатива перегрузки:

```
void print(int value, int base); // Вывод значения в системе
                                 // счисления с основанием base
void print(int value)            // Десятичный вывод
{
    print(value ,10);
}
```

3.6.2. Возврат значения

Как только мы вычислили результат, его нужно вывести из функции и вернуть вызывающей функции. И вновь по умолчанию возврат значения выполняется с помощью копирования, и для небольших объектов это оказывается идеальным решением. Мы возвращаем результат "по ссылке" только тогда, когда хотим предоставить вызывающей функции доступ к сущности, не являющейся локальной для этой функции. Например:

```
class Vector {
public:
    // ...
    double& operator[](int i) { return elem[i]; } // Возврат ссылки
                                                  // на i-й элемент
private:
    double* elem;                                 // elem указывает
    // ...                                  // на массив из sz элементов
};
```

i-й элемент Vector существует независимо от вызова оператора индекса, поэтому мы можем вернуть ссылку на него.

С другой стороны, по завершении функции ее локальные переменные исчезают, поэтому мы не должны возвращать указатель или ссылку на них:

```
int& bad()
{
    int x;
    // ...
    return x; // Плохо: возврат ссылки на локальную переменную x
}
```

К счастью, все основные компиляторы C++ в состоянии обнаружить эту очевидную ошибку в bad().

Возврат ссылки или значения "малого" типа эффективно, но как же передавать из функции большие количества информации? Рассмотрим следующий код:

```
Matrix operator+(const Matrix& x, const Matrix& y)
{
    Matrix res;
    // ... Для всех res[i,j] имеем res[i,j] = x[i,j]+y[i,j]...
    return res;
}

Matrix m1, m2;
// ...
Matrix m3 = m1+m2; // Без копирования
```

Матрица Matrix может быть *очень* большой, с дорогостоящим копированием даже на современном аппаратном обеспечении. Поэтому, чтобы избежать копирования, мы создаем перемещающий конструктор Matrix (§5.2.2), а операция перемещения Matrix из оператора operator+() оказывается очень дешевой. Нам *не* требуется возвращаться к ручному управлению памятью:

```
Matrix* add(const Matrix& x, const Matrix& y) // Сложный и чреватый
                                              // ошибками стиль XX века
{
    Matrix* p = new Matrix;
    // ... Для всех *p[i,j], *p[i,j] = x[i,j]+y[i,j] ...
    return p;
}

Matrix m1, m2;
// ...
Matrix* m3 = add(m1,m2);    // Копируем только указатель
// ...
delete m3;                  // Легко забыть удалить...
```

К сожалению, возврат больших объектов путем возврата указателя на них весьма распространен в старом коде и является основным источником трудно обнаруживаемых ошибок. Не пишите такой код! Обратите внимание, что `operator+()` так же эффективен, как и `add()`, но гораздо проще для определения, проще в использовании и менее подвержен ошибкам.

Если функция не может выполнить требуемую задачу, она может сгенерировать исключение (§3.5.1). Это может помочь коду избежать переполненности проверками кодов ошибок для "исключительных проблем".

Тип возвращаемой функции можно вывести из возвращаемого значения. Например:

```
auto mul(int i, double d){return i*d;} // здесь "auto" означает
                                       // "вывести возвращаемый тип"
```

Это может быть удобно, в особенности для обобщенных функций (шаблонов функций, §6.3.1) и лямбда-выражений (§6.3.3), но их следует использовать осторожно, потому что выведенный тип не предлагает стабильного интерфейса: изменение в реализации функции (или лямбда-выражения) может изменить этот тип.

3.6.3. Структурное связывание

Функция может возвращать только одно значение, но это значение может быть объектом класса со многими членами. Это позволяет нам эффективно возвращать из функции несколько значений. Например:

```
struct Entry
{
    string name;
    int value;
};

Entry read_entry(istream& is) // Наивная функция чтения (лучший
{                             // вариант приведен в §10.5)
    string s;
    int i;
    is >> s >> i;
    return {s,i};
}

auto e = read_entry(cin);
cout << "{ " << e.name << ", " << e.value << " }\n";
```

Здесь `{s,i}` используется для создания возвращаемого значения `Entry`. Аналогично можно "распаковать" члены `Entry` в локальные переменные:

```
auto [n,v] = read_entry(is);
cout << "{ " << n << ", " << v << " }\n";
```

Конструкция `auto[n,v]` объявляет две локальные переменные `n` и `v` с типами, выведенными из возвращаемого типа `read_entry()`. Этот механизм придания локальных имен членам объекта класса называется *структурным связыванием* (structured binding).

Рассмотрим еще один пример:

```
map<string,int> m;
// ... Заполнение m ...
for (const auto [key,value] : m)
cout << "{" << key "," << value << "}\n";
```

Как обычно, можно декорировать `auto` с помощью `const` и `&`. Например:

```
void incr(map<string,int>&m) // Увеличение значения каждого элемента m
{
    for (auto& [key,value] : m)
        ++value;
}
```

Когда структурное связывание используется для класса без закрытых данных, легко увидеть, как выполняется такое связывание: для связывания должно быть определено столько же имен, сколько имеется нестатических членов-данных класса, и каждое введенное имя связывается с соответствующим членом. В качестве объектного кода по сравнению с явным использованием составного объекта нет никакого отличия; использование структурного связывания — просто иное выражение все той же концепции.

Можно также работать и с классами, в которых доступ осуществляется через функции-члены. Например:

```
complex<double> z = {1,2};
auto [re,im] = z+2; // re=3; im=2
```

Объект типа `complex` имеет два члена-данных, но его интерфейс состоит из функций доступа, таких как `real()` и `imag()`. Отображение `complex <double>` на две локальные переменные, такие как `re` и `im`, осуществимо и эффективно, но техника, позволяющая это сделать, выходит за рамки данной книги.

3.7. Советы

[1] Различайте объявления (используемые в качестве интерфейсов) и определения (используемые как реализации); §3.1.

[2] Используйте заголовочные файлы для представления интерфейсов и для подчеркивания логической структуры; §3.2; [CG:SF.3].

[3] Включайте заголовочные файлы с помощью директив `#include` в исходные файлы, которые реализуют их функции; §3.2; [CG:SF.5].

[4] Избегайте в заголовочных файлах определений функций, не являющихся `inline`; §3.2; [CG:SF.2].

[5] Предпочитайте модули заголовочным файлам (где имеется поддержка `module`); §3.3.

[6] Используйте пространства имен для выражения логической структуры; §3.4; [CG:SF.20].

[7] Используйте директивы `using` при переделке приложений, для базовых библиотек (таких, как `std`) или в локальной области видимости; §3.4; [CG:SF.6] [CG:SF.7].

[8] Не помещайте директивы `using` в заголовочный файл; §3.4; [CG:SF.7].

[9] Генерируйте исключения для указания того, что вы не в состоянии выполнить определенную задачу; §3.5; [CG:E.2].

[10] Используйте исключения только для обработки ошибок; §3.5.3; [CG:E.3].

[11] Применяйте коды ошибок там, где предполагается, что обрабатывать ошибки будет непосредственно вызывающая функция; §3.5.3.

[12] Генерируйте исключения, если предполагается, что ошибка должна передаваться через несколько вызовов функций; §3.5.3.

[13] Если вы не знаете, что лучше использовать, исключения или коды ошибок, предпочитайте исключения; §3.5.3.

[14] Разрабатывайте стратегию обработки ошибок как можно раньше при проектировании; §3.5; [CG:E.12].

[15] Используйте для исключений специально разработанные пользовательские типы (не встроенные типы); §3.5.1.

[16] Не пытайтесь перехватывать все исключения в каждой функции; §3.5; [CG:E.7].

[17] Предпочитайте идиому RAII явным `try`-блокам; §3.5.1, §3.5.2; [CG:E.6].

[18] Если ваша функция не может генерировать исключения, объявите ее как `noexcept`; §3.5; [CG:E.12].

[19] Конструктор должен обеспечивать выполнение инварианта и генерировать исключение, если не в состоянии это сделать; §3.5.2; [CG:E.5].

[20] Проектируйте свою стратегию обработки ошибок вокруг инвариантов; §3.5.2; [CG:E.4].

[21] То, что можно проверить во время компиляции, обычно лучше проверять во время компиляции; §3.5.5 [CG:P.4] [CG:P.5].

[22] Передавайте "малые" значения в функции по значению, а "большие" — по ссылке; §3.6.1; [CG:F.16].

[23] Предпочитайте передачу по константной ссылке передаче по обычной, неконстантной ссылке; [CG:F.17].

[24] Возвращайте информацию из функций в виде возвращаемых функцией значений (а не с помощью выходных параметров); §3.6.2; [CG:F.20] [CG:F.21].

[25] Не злоупотребляйте выводом типа возвращаемого значения; §3.6.2.

[26] Не злоупотребляйте структурным связыванием; именованные возвращаемые типы зачастую являются более ясной документацией; §3.6.3.

4

Классы

— Эти типы не абстрактны — они реальны
так же, как и типы `int` *и* `double`.
Дуг МакИлрой

4.1. Введение

Цель этой и следующих трех глав — дать вам представление о поддержке абстракции и управления ресурсами C++ без погружения в детали.

- В этой главе неформально представлены способы определения и использования новых типов (*пользовательских типов*). В частности, здесь представлены основные свойства, методы реализации и языковые средства, используемые для *конкретных классов*, *абстрактных классов* и *иерархий классов*.
- В главе 5, "Основные операции", представлены операции, которые имеют определенный смысл в C++, такие как конструкторы, деструкторы и присваивания. В ней излагаются правила их совместного использования для управления жизненным циклом объектов и поддержки простого, эффективного и полного управления ресурсами.

- В главе 6, "Шаблоны", представлены шаблоны как механизм для параметризации типов и алгоритмов (другими) типами и алгоритмами. Вычисления над пользовательскими и встроенными типами представлены в виде функций, иногда обобщенных до *шаблонных функций* и *функциональных объектов*.
- В главе 7, "Концепты и обобщенное программирование", представлен обзор концепций, технологий и языковых возможностей, лежащих в основе обобщенного программирования. Основное внимание уделяется определению и использованию *концептов* для точных определений интерфейсов шаблонов и проектирования. Вводятся шаблоны *с переменным количеством параметров* (*вариативные шаблоны*) для определения наиболее общих и наиболее гибких интерфейсов.

Это языковые средства, поддерживающие стили программирования, известные как *объектно-ориентированное программирование* и *обобщенное программирование*. В главах 8–15 приводятся примеры возможностей стандартной библиотеки и их использования.

Центральной особенностью языка C++ является *класс*. Класс — это пользовательский тип, представляющий в коде программы некоторую концепцию. Всякий раз, когда наш дизайн программы содержит полезную концепцию, идею, сущность и тому подобное, мы стараемся представить ее в программе в виде класса, чтобы идея содержалась в коде, а не только в наших головах, проектной документации или в некоторых комментариях. Программу, построенную из хорошо подобранного набора классов, намного легче понять и сделать безошибочной, чем программу, которая строится непосредственно на фундаменте фундаментальных встроенных типов. В частности, библиотеки часто предлагают именно наборы классов.

По сути, все языковые средства (помимо основных типов, операторов и инструкций) существуют для того, чтобы помочь определить лучшие классы или более удобно их использовать. Под "лучшими" классами я имею в виду более правильные, более легкие в обслуживании, более эффективные, более элегантные, более простые в применении, более удобочитаемые и надежные. Большинство методов программирования опирается на разработку и реализацию конкретных видов классов. Потребности и вкусы программистов очень разнообразны, а потому и поддержка классов очень обширна. Здесь мы рассмотрим только базовую поддержку трех важных разновидностей классов:

- конкретные классы (§4.2);
- абстрактные классы (§4.3);
- классы в иерархиях классов (§4.5).

Поразительное количество полезных классов оказываются принадлежащими одной из этих трех разновидностей. Еще больше классов можно рассматривать как простые варианты этих разновидностей или реализуемые с использованием сочетаний используемых для них методов.

4.2. Конкретные типы

Основная идея *конкретных классов* заключается в том, что они ведут себя, "как встроенные типы". Например, тип комплексного числа или целое число с бесконечной точностью очень похожи на встроенный `int`, за исключением того, что они обладают собственной семантикой и собственными наборами операций. Аналогично `vector` и `string` очень похожи на встроенные массивы, но ведут себя куда лучше (§9.2, §10.3, §11.2).

Определяющей характеристикой конкретного типа является то, что его представление является частью его определения. Во многих важных случаях, таких как `vector`, это представление является лишь одним или несколькими указателями на данные, хранящиеся в другом месте, но это представление присутствует в каждом объекте конкретного класса. Это позволяет обеспечить максимальную эффективность во времени и пространстве. В частности, это позволяет:

- помещать объекты конкретных типов в стек, в статически выделенную память и в другие объекты (§1.5);
- обращаться к объектам непосредственно (а не только с помощью указателей или ссылок);
- непосредственно и полностью инициализировать объекты (например, с использованием конструкторов; §2.3);
- копировать и перемещать объекты (§5.2).

Представление может быть закрытым (как для `Vector`, § 2.3) и доступным только через функции-члены, но оно присутствует. Поэтому, если представление изменяется каким-либо существенным образом, пользователь должен перекомпилировать исходные тексты. Это цена того, что конкретные типы ведут себя точно так же, как встроенные типы. Для типов, которые меняются нечасто и в которых локальные переменные обеспечивают столь необходимую ясность и эффективность, это вполне приемлемо и часто просто идеально. Чтобы повысить гибкость, конкретный тип может хранить основные части своего представления в свободной памяти (динамической памяти, куче) и получать доступ к ним через часть, хранящуюся в самом объекте класса. Так реализованы `vector` и `string`; их можно считать дескрипторами ресурсов с тщательно отработанными интерфейсами.

4.2.1. Арифметический тип

"Классическим пользовательским арифметическим типом" является тип `complex`:

```
class complex
{
    double re, im;          // Представление: два double
public:
    // Построение комплексного числа из двух скаляров:
    complex(double r, double i):re{r},im{i}{}

    // Построение комплексного числа из одного скаляра:
    complex(double r):re{r},im{0}{}

    complex():re{0},im{0}{} // Комплексное число по умолчанию: {0,0}

    double real() const { return re; }

    void real(double d) { re=d;      }

    double imag() const { return im; }

    void imag(double d) { im=d;      }

    complex& operator+=(complex z)
    {
        re+=z.re;     // Сложение re и im
        im+=z.im;
        return *this; // и возврат результата
    }

    complex& operator-=(complex z)
    {
        re-=z.re;
        im-=z.im;
        return *this;
    }

    complex& operator*=(complex); // Определена вне класса
    complex& operator/=(complex); // Определена вне класса
};
```

Это немного упрощенная версия комплексных чисел `complex` стандартной библиотеки (§14.4). Само определение класса содержит только операции, требующие доступа к представлению. Представление же простое и традиционное. По практическим соображениям оно должно быть совместимо с тем, что предоставлял Fortran 60 лет назад, и нам нужен обычный набор операторов. В дополнение к логическим требованиям класс `complex` должен быть эффективным, иначе он не будет востребован пользователями. Это означает,

что простые операции должны быть встраиваемыми (inline), т.е. эти операции (такие, как конструкторы, += или imag()) должны быть реализованы без вызовов функций в сгенерированном машинном коде. Функции, определенные в классе, являются встраиваемыми по умолчанию. Можно явно запросить встраиваемость функции, предварив ее объявление ключевым словом inline. Реализация промышленного типа complex (наподобие имеющегося в стандартной библиотеке) тщательно настраивается для выполнения встраивания.

Конструктор, который может быть вызван без аргумента, называется *конструктором по умолчанию*. Таким образом, complex() является конструктором complex по умолчанию. Определив конструктор по умолчанию, вы исключаете возможность наличия неинициализированных переменных данного типа.

Спецификаторы const для функций, возвращающих действительную и мнимую части, указывают, что эти функции не изменяют объект, для которого вызываются. Константная функция-член может быть вызвана как для константных, так и для неконстантных объектов, но неконстантная функция-член может быть вызвана только для неконстантных объектов. Например:

```
complex z = {1,0};
const complex cz {1,3};
z =cz;                // OK: присваивание неконстантной переменной
cz = z;               // Ошибка: complex::operator=() неконстантная
double x = z.real(); // OK: complex::real() — константная
```

Многие полезные операции не требуют непосредственного доступа к представлению complex и поэтому могут быть определены отдельно от определения класса:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
// Унарный минус:
complex operator-(complex a) { return {-a.real(), -a.imag()}; }
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

Здесь я использовал тот факт, что переданный по значению аргумент копируется, так что я могу модифицировать аргумент без воздействия на копию в вызывающей функции и использовать результат как возвращаемое значение.

Определения операторов == и != незамысловаты:

```
bool operator==(complex a, complex b) // Равенство
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

```
bool operator!=(complex a, complex b) // Неравенство
{
    return !(a==b);
}
complex sqr t(complex);               // Определение в другом месте
// ...
```

Использоваться класс `complex` может следующим образом:

```
void f(complex z)
{
    complex a {2.3};   // Построение {2.3,0.0} из 2.3
    complex b {1/a};
    complex c {a+z*complex{1,2.3}};
    // ...
    if (c != b)
    c = -(b/a)+2*b;
}
```

Компилятор преобразует операторы для работы с числами `complex` в соответствующие вызовы функций. Например, `c!=b` означает `operator!=(c,b)`, а `1/a` означает `operator/(complex{1},a)`.

Пользовательские операторы ("перегруженные операторы") должны использоваться разумно и осторожно. Синтаксис операторов фиксируется языком, так что вы не можете определить унарный оператор `/`. Кроме того, невозможно изменить значение оператора для встроенных типов, поэтому вы не можете переопределить + так, чтобы он работал для `int` как вычитание.

4.2.2. Контейнер

Контейнер представляет собой объект, содержащий коллекцию элементов. Мы называем класс `Vector` контейнером, потому что объекты типа `Vector` являются контейнерами. Как определено в §2.3, `Vector` является продуманным контейнером `double`: он прост для понимания, устанавливает полезный инвариант (§3.5.2), обеспечивает проверку выхода обращения к элементу за диапазон (§3.5.1) и обеспечивает функцию `size()`, что позволяет выполнять обход его элементов. Однако у него есть фатальный недостаток: он выделяет память для элементов, используя оператор `new`, но никогда ее не освобождает. Это не очень хорошая идея, потому что, хотя C++ и определяет интерфейс сборщика мусора (§5.3), не гарантируется его наличие для того, чтобы сделать неиспользуемую память доступной для новых объектов. Во многих средах вы не можете использовать сборку мусора, и часто по логическим причинам или из соображений производительности предпочтительнее использовать более точное управление уничтожением объектов. Нам нужен механизм, освобождающий память, выделенную конструктором; таким механизмом является *деструктор*:

```
class Vector
{
public:
    // Конструктор: захват ресурсов:
    Vector(int s) :elem{new double[s]}, sz{s}
    {
        for (int i=0; i!=s; ++i) // Инициализация элементов
            elem[i]=0;
    }

    // Деструктор: освобождение ресурсов
    ~Vector()
    {
        delete[] elem;
    }

    double& operator[](int i);
    int size() const;
private:
    // elem указывает на массив из sz элементов типа double:
    double* elem;
    int sz;
};
```

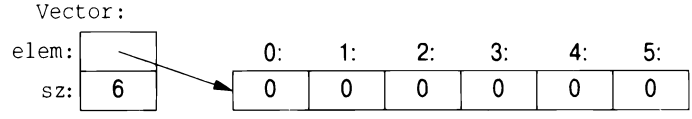
Имя деструктора состоит из оператора дополнения ~, за которым следует имя класса; деструктор — это дополнение к конструктору. Конструктор `Vector` выделяет некоторую память в свободной памяти (именуемой также кучей или динамической памятью) с использованием оператора `new[]`. Деструктор освобождает эту память с помощью оператора `delete[]`. Обычный оператор `delete` удаляет отдельный объект, `delete[]` удаляет массив.

Все это делается без вмешательства пользователей класса `Vector`. Пользователи просто создают и используют объекты `Vector` так же, как и переменные встроенных типов. Например:

```
void fct(int n)
{
    Vector v(n);
    // ... Использование v ...
    {
        Vector v2(2*n);
        // ... Использование v и v2 ...
    }   // Здесь уничтожается v2
    // ... Использование v ..
}   // Здесь уничтожается v
```

`Vector` подчиняется тем же правилам именования, области видимости, выделения памяти, времени жизни и так далее (§1.5), что и встроенные типы, такие как `int` и `char`. Этот `Vector` представляет собой сильно упрощенный вариант из-за отсутствия обработки ошибок; см. §3.5.

Комбинация конструктора/деструктора является основой многих элегантных технологий. В частности, она является основой большинства методов управления ресурсами в C++ (§5.3, §13.2). Рассмотрим графическую иллюстрацию `Vector`.



Конструктор выделяет память для элементов и соответствующим образом инициализирует элементы `Vector`. Деструктор освобождает память, выделенную для элементов. Эта *модель управления данными* очень часто используется для управления данными, которые могут изменяться в размерах в процессе жизненного цикла объекта. Методика получения ресурсов в конструкторе и освобождение их в деструкторе известна как идиома "*Захват ресурсов есть инициализация*" (Resource Acquisition Is Initialization — RAII) и позволяет избежать операций "с голым `new`", т.е. избежать выделения памяти в общем коде и сохранить его только в реализациях хорошо ведущих себя абстракций. Аналогично следует избегать "голых операций `delete`". Все это делает код намного менее подверженным ошибкам и позволяет намного проще избежать утечек ресурсов (§13.2).

4.2.3. Инициализация контейнеров

Контейнер существует для хранения элементов, поэтому очевидно, что нам нужны удобные способы добавления элементов в контейнер. Мы можем создать `Vector` с соответствующим количеством элементов, а затем выполнить их присваивание, но обычно другие способы более элегантны. Здесь я просто упомяну двух фаворитов.

- *Конструктор со списком инициализации*: инициализация с помощью списка элементов.
- `push_back()`: добавление новых элементов в конец последовательности.

Они могут быть объявлены следующим образом:

```
class Vector
{
public:
    // Инициализация списком элементов типа double:
    Vector(std::initializer_list<double>);

    // ...
```

```
    // Добавление элемента в конец с увеличением размера на 1:
    void push_back(double);
    // ...
};
```

Функция `push_back()` полезна для добавления произвольного количества элементов. Например:

```
Vector read(istream& is)
{
    Vector v;
    for(double d; is>>d; ) // Чтение значений с плавающей точкой в d
        v.push_back(d);    // Добавление d в вектор v
    return v;
}
```

Входной цикл завершается при достижении конца файла или при ошибке форматирования. До тех пор, пока это не произойдет, каждое прочитанное число добавляется в `Vector`, так что в конце работы функции размер вектора `v` равен количеству прочитанных элементов. Я использовал цикл `for`, а не более удобный цикл `while`, чтобы ограничить область видимости переменной `d`, ограниченную циклом. Предоставление классу `Vector` конструктора перемещения, обеспечивающего дешевый возврат потенциально огромного количества данных из функции `read()`, объясняется в §5.2.2:

```
Vector v = read(cin); // Копирования элементов Vector нет
```

Способ представления `std::vector` для того, чтобы сделать эффективными операцию `push_back()` и другие операции, изменяющие размер `vector`, представлен в §11.2.

Класс `std::initializer_list` используется для определения конструктора на основе списка инициализации и представляет собой тип стандартной библиотеки, известный компилятору: когда мы используем список в фигурных скобках (`{}`), такой как `{1,2,3,4}`, компилятор создает объект типа `initializer_list` для предоставления его программе. Так что можно написать:

```
Vector v1 = {1,2,3,4,5};          // v1 имеет 5 элементов
Vector v2 = {1.23, 3.45, 6.7, 8}; // v2 имеет 4 элемента
```

Конструктор со списком инициализации класса `Vector` может быть определен следующим образом:

```
// Инициализация списком:
Vector::Vector(std::initializ er_list<double> lst)
:elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
```

```
    // Копирование в elem из lst (§12.6):
    copy(lst.begin(),lst.end(),elem);
}
```

К сожалению, стандартная библиотека использует для размеров и индексов беззнаковые целые числа, поэтому мне нужно использовать уродливый `static_cast` для явного преобразования размера списка инициализаторов в `int`. Это избыточно педантично, потому что вероятность того, что количество элементов в рукописном списке больше, чем наибольшее целое число (32 767 для 16-разрядных целых чисел и 2 147 483 647 для 32-разрядных целых чисел), очень невелика. Однако система типов здравым смыслом не обладает. Она знает о возможных значениях переменных, а не об их действительных значениях, поэтому может жаловаться, даже если фактическое нарушение отсутствует. Однако такие предупреждения иногда могут спасать программиста от нехороших ошибок.

`static_cast` не проверяет значение, которое конвертирует; считается, что программист знает, что делает, и использует его правильно. Это предположение не всегда правильное, поэтому, если у вас есть сомнения, проверьте значение. Явных преобразований типов (часто называемых *приведениями*, чтобы напоминать вам, что они могут привести не туда[1]) лучше избегать. Попытайтесь использовать непроверяемые приведения только на самом низком уровне системы. Они чреваты ошибками.

Другими приведениями являются `reinterpret_cast` для трактовки объекта как простой последовательности байтов и `const_cast` для "отбрасывания `const`". Разумное использование системы типов и хорошо продуманных библиотек позволяет исключить непроверяемые приведения в программном обеспечении более высокого уровня.

4.3. Абстрактные типы

Такие типы, как `complex` и `Vector`, называются *конкретными типами*, потому что их представление является частью их определения. В этом они напоминают встроенные типы. В противоположность им *абстрактный тип* — это тип, который полностью изолирует пользователя от деталей реализации. Для этого мы отделяем интерфейс от представления и отказываемся от реальных локальных переменных. Поскольку мы ничего не знаем о представлении абстрактного типа (даже о его размере), мы должны выделить объекты в свободной памяти (§4.2.2) и получить доступ к ним через ссылки или указатели (§1.7, §13.2.1).

[1] В оригинале — "чтобы напоминать вам, что они используются для починки чего-то сломанного". Игра слов, основанная на том, что на английском "приведение типов" записывается как *cast*, а одно из значений этого слова — "отливка, литье (металла)". — *Примеч. пер.*

Сначала мы определим интерфейс класса `Container`, который будем разрабатывать как более абстрактную версию нашего класса `Vector`:

```
class Container
{
public:
    // Чисто виртуальная функция:
    virtual double& operator[](int) = 0;

    // Константная функция-член (§4.2.1):
    virtual int size() const = 0;

    // Деструктор (§4.2.2):
    virtual ~Container() {}
};
```

Этот класс является чистым интерфейсом для некоторых контейнеров, которые будут определены позже. Слово `virtual` означает "может быть переопределено позже в классе, производном от данного". Неудивительно, что объявленная как `virtual` функция называется *виртуальной функцией*. Класс, производный от `Container`, обеспечивает реализацию интерфейса `Container`. Любопытный синтаксис `=0` говорит о том, что функция является *чисто виртуальной*, т.е. некоторый класс, производный от `Container`, *обязан* ее определить. Таким образом, невозможно определить объект, который имеет тип просто `Container`. Например:

```
// Ошибка: объект абстрактного класса существовать не может:
Container c;
// ОК: Container представляет собой интерфейс:
Container* p = new Vector_container(10);
```

`Container` может служить только интерфейсом класса, который реализует функции `operator[]()` и `size()`. Класс с чисто виртуальной функцией называется *абстрактным классом*.

Данный `Container` может использоваться следующим образом:

```
void use(Container& c)
{
    const int sz = c.size();
    for(int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Обратите внимание, как функция `use()` использует интерфейс контейнера при полном незнании деталей реализации. Она использует `size()` и `[]` без какого-либо представления о том, какой именно тип обеспечивает их реализацию. Класс, который предоставляет интерфейс для множества других классов, часто называют *полиморфным типом*.

Как это часто бывает у абстрактных классов, класс Container не имеет конструктора. В конце концов, у него нет никаких данных для инициализации. С другой стороны, Container имеет деструктор и этот деструктор является виртуальным, поэтому классы, производные от Container, могут обеспечить соответствующую реализацию. Это также обычное явление для абстрактных классов, потому что работа с ними, как правило, выполняется с помощью ссылок или указателей, а при уничтожении Container через указатель неизвестно, какие именно ресурсы принадлежат его реализации; см. также §4.5.

Абстрактный класс Container определяет только интерфейс (и не определяет никакой реализации). Чтобы класс Container был полезен, мы должны написать контейнер, который реализует функции, требуемые его интерфейсом. Для этого мы могли бы использовать конкретный класс Vector:

```
// Vector_container реализует Container:
class Vector_container : public Container
{
public:
    Vector_container(int s) : v(s) { } // Vector с s элементами
    ~Vector_container() {}

    double& operator[](int i) override { return v[i]; }
    int size() const override { return v.size(); }
private:
    Vector v;
};
```

Запись :public может быть прочитана как "порожден из" или "является подтипом". Класс Vector_container называется *производным* от класса Container, а класс Container является *базовым* классом для Vector_container. Альтернативная терминология называет Vector_container *подклассом*, а Container — *суперклассом*. Говорят, что *производный* класс наследует члены из своего базового класса, поэтому использование базового и производного классов обычно называется *наследованием*.

Члены operator[]() и size() *перекрывают* (override) соответствующие элементы базового класса Container. Я явно использовал ключевое слово override, чтобы пояснить, что это было сделано преднамеренно. Использование override не является обязательным, но его применение позволяет компилятору обнаруживать ошибки, такие как опечатки в именах функций или небольшие различия между типом виртуальной функции и ее предполагаемым перекрытием. Явное использование override особенно полезно в больших высокоуровневых иерархиях, в которых в противном случае трудно понять, что должно было быть перекрыто и чем.

Деструктор ~Vector_container() перекрывает деструктор базового класса ~Container(). Обратите внимание, что деструктор члена ~Vector() неявно вызывается деструктором его класса ~Vector_container().

Чтобы такая функция, как use(Container&), могла использовать Container при полном незнании деталей реализации, некоторая другая функция должна создать объект, с которым она сможет работать. Например:

```
void g()
{
    Vector_container vc(10); // Вектор из 10 элементов
    // ... Заполнение vc ...
    use(vc);
}
```

Поскольку use() не знает о Vector_container, а осведомлена только об интерфейсе Container, она будет одинаково хорошо работать с различными реализациями Container. Например:

```
// List_container реализует Container:
class List_container : public Container
{
public:
    List_container() { }  // Пустой список
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}
    double& operator[](int i) override;
    int size() const override { return ld.size(); }
private:
    std::list<double> ld; // Список double из стандартной
};                        // библиотеки (§11.3)

double& List_container::operator[](int i)
{
    for (auto& x : ld)
    {
        if (i==0)
            return x;
        --i;
    }
    throw out_of_range{"List container"};
}
```

Здесь представление принимает вид класса стандартной библиотеки list <double>. В обычной ситуации я бы не реализовывал контейнер с операцией индексирования, используя список, потому что производительность индексирования списка существенно хуже по сравнению с таковой у вектора. Однако здесь я просто хотел показать реализацию, которая радикально отличается от обычной.

Функция может создавать List_container и использовать его в функции use():

```
void h()
{
    List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    use(lc);
}
```

Главное в том, что use(Container&) не знает, имеет ли его аргумент тип Vector_container, List_container или какой-либо иной вид Container; ему не нужно это знать. Он может использовать любой вид Container. Он знает только интерфейс, определяемый Container. Следовательно, use(Container&) не нужно перекомпилировать, если изменяется реализация List_container или используется какой-то новый класс, производный от Container.

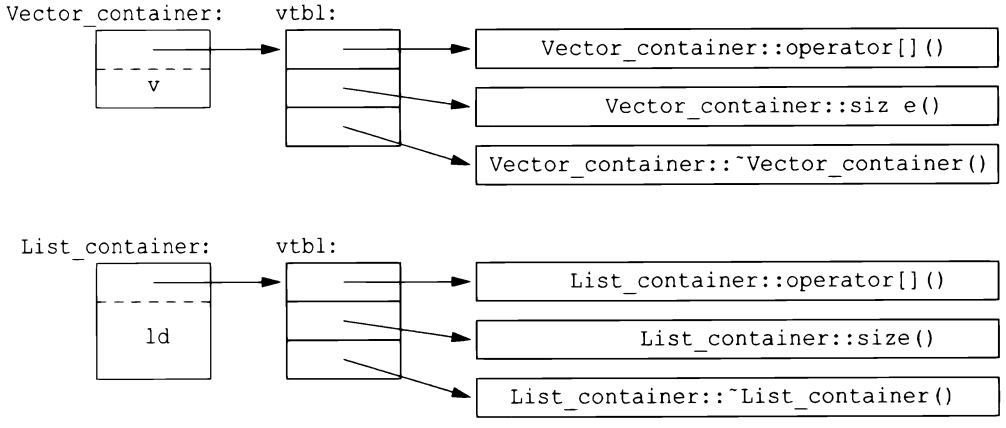
Оборотной стороной этой гибкости является то, что работать с такими объектами требуется только по ссылке или через указатель (§5.2, §13.2.1).

4.4. Виртуальные функции

Вернемся вновь к использованию Container:

```
void use(Container& c)
{
    const int sz = c.size();
    for(int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

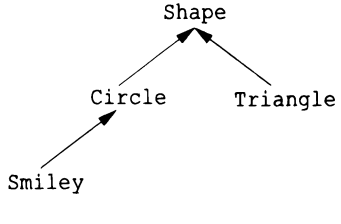
Каким образом вызов c[i] в use() обращается к правильному оператору operator[]()? Когда h() вызывает use(), должен вызываться operator[]() класса List_container. Когда же use() вызывается из g(), необходимо вызывать оператор operator[]() класса Vector_container. Чтобы суметь разрешить вызов operator[](), объект Container должен содержать информацию, позволяющую выбрать нужную функцию для вызова во время выполнения. Обычный способ реализации заключается преобразовании компилятором имени виртуальной функции в индекс в таблице указателей на функции. Такую таблицу обычно называют *таблицей виртуальных функций* или просто vtbl. Каждый класс с виртуальными функциями имеет собственную vtbl, определяющую его виртуальные функции. Графически это можно представить следующим образом.



Функции в `vtbl` позволяют правильно использовать объект, даже если его размер и схема размещения его данных неизвестны вызывающей функции. Реализация вызывающей функции должна знать только местоположение указателя на `vtbl` в `Container` и индекс, используемый для каждой виртуальной функции. Этот механизм виртуального вызова можно сделать почти таким же эффективным, как и механизм "нормального вызова функции" (в пределах 25%). Его накладные расходы в смысле памяти — один указатель в каждом объекте класса с виртуальными функциями плюс один `vtbl` для каждого такого класса.

4.5. Иерархии классов

Пример `Container` — очень простой пример иерархии классов. *Иерархия классов* представляет собой множество классов, упорядоченных в виде сетки, созданной наследованием классов (например, `public`). Мы используем иерархии классов для представления концепций, которые имеют иерархические отношения, такие как "пожарная машина — это разновидность автомобиля, который, в свою очередь, является разновидностью транспортного средства", или "смайлик — разновидность круга, который, в свою очередь, является разновидностью геометрической фигуры". Нередки огромные глубокие и широкие иерархии с сотнями классов. В качестве полуреалистического классического примера рассмотрим геометрические фигуры на экране.



Стрелки указывают отношение наследования. Например, класс `Circle` является производным от класса `Shape`. Иерархия классов обычно изображается растущей вниз от "самого базового" класса — от корня к (определяемым позже) производным классам. Для представления такой простой диаграммы в коде сначала нужно создать класс, который определяет общие свойства всех геометрических фигур:

```
class Shape
{
public:
    virtual Point center() const =0; // Чисто виртуальная
    virtual void move(Point to) =0;
    virtual void draw() const = 0;   // Вывод на текущем "холсте"
    virtual void rotate(int angle) = 0;
    virtual ~Shape() {}              // Деструктор
    // ...
};
```

Естественно, этот интерфейс является абстрактным классом; что касается представления, то ничто (кроме местоположения указателя на `vtbl`) не является общим для каждого `Shape`. В этом определении мы можем написать только общие функции, управляющие векторами указателей на фигуры:

```
// Поворот элементов v на угол angle градусов:
void rotate_all(vector<Shape*>& v, int angle)
{
    for (auto p : v)
        p->rotate(angle);
}
```

Чтобы определить конкретную фигуру, мы должны указать, что это `Shape`, и определить ее конкретные свойства (включая виртуальные функции):

```
class Circle : public Shape
{
public:
    Circle(Point p, int rad);    // Конструктор

    Point center() const override
    {
        return x;
    }
    void move(Point to) override
    {
    x = to;
    }

    void draw() const override;
    void rotate(int) override {} // Простой алгоритм
```

```
private:
    Point x;                    // Центр
    int r;                      // Радиус
};
```

Пока что пример `Shape` и `Circle` не дает ничего нового по сравнению с примером `Container` и `Vector_container`, но мы можем продолжить:

```
// Используем окружность как базовый класс для лица:
class Smiley : public Circle
{
public:
    Smiley(Point p, int rad) : Circle{p,r}, mouth{nullptr} { }
    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes)
            delete p;
    }

    void move(Point to) override;
    void draw() const   override;
    void rotate(int)    override;

    void add_eye(Shape* s)
    {
        eyes.push_back(s);
    }

    void set_mouth(Shape* s);
    virtual void wink(int i); // Подмигивание i-м глазом
    // ...
private:
    vector<Shape*> eyes;      // Обычно два глаза
    Shape* mouth;
};
```

Функция-член `push_back()` класса `vector` копирует свой аргумент в `vector` (здесь — `eyes`) в качестве последнего элемента, увеличивая размер вектора на единицу.

Теперь мы можем определить `Smiley::draw()` с использованием вызовов члена `draw()` базового класса и членов:

```
void Smiley::draw() const
{
    Circle::draw();
    for (auto p : eyes)
        p->draw();
    mouth->draw();
}
```

Обратите внимание на то, как Smiley хранит глаза в vector стандартной библиотеки и удаляет их в своем деструкторе. Деструктор Shape является виртуальным, а деструктор Smiley перекрывает его. Виртуальный деструктор необходим для абстрактного класса, потому что объект производного класса обычно обрабатывается через интерфейс, предоставляемый его абстрактным базовым классом. В частности, он может быть удален с помощью указателя на базовый класс, и в этом случае механизм вызова виртуальной функции гарантирует, что будет вызван правильный деструктор. Этот деструктор затем неявно вызывает деструкторы его базовых классов и членов.

В этом упрощенном примере задача программиста — поместить глаза и рот надлежащим образом в круг, представляющий лицо.

Мы можем добавлять элементы данных, операции или и то и другое, поскольку мы определяем новый класс путем наследования. Это дает большую гибкость — вместе с соответствующими возможностями для путаницы и плохого дизайна.

4.5.1. Преимущества иерархий

Иерархия классов предоставляет две разновидности преимуществ.

- *Наследование интерфейса.* Объект производного класса можно использовать везде, где требуется объект базового класса. То есть базовый класс действует в качестве интерфейса для производного класса. Примерами являются классы Container и Shape. Такие классы часто являются абстрактными.
- *Наследование реализации.* Базовый класс предоставляет функции или данные, что упрощает реализацию производных классов. Примерами являются использование классом Smiley конструктора Circle и функции-члена Circle::draw(). Такие базовые классы часто имеют члены данных и конструкторы.

Конкретные классы — особенно с небольшими представлениями — очень похожи на встроенные типы: мы определяем их как локальные переменные, обращаемся к ним с помощью имен, копируем их и т.д. Классы в иерархии классов различаются — обычно для них выделяется память с использованием оператора new, и доступ к ним выполняется через указатели или ссылки. Например, рассмотрим функцию, которая считывает данные, описывающие фигуры, из входного потока и создает соответствующие объекты Shape:

```
enum class Kind { circle, triangle, smiley };

// Чтение описаний фигур из входного потока is:
Shape* read_shape(istream& is)
{
```

```
    // ... Чтение заголовка фигуры из is и поиск его Kind k ...
    switch (k)
    {
    case Kind::circle:
        // Чтение данных окружности {Point,int} в p и r
        return new Circle{p,r};
    case Kind::triangle:
        // Чтение вершин треугольника {Point,Point,Point}
        // в p1, p2 и p3
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // Чтение данных смайлика {Point,int,Shape,Shape,Shape}
        // в p, r, e1, e2 и m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

Программа может использовать функцию чтения фигуры следующим образом:

```
void user()
{
    std::vector<Shape*> v;

    while (cin)
        v.push_back(read_shape(cin));

    draw_all(v);      // Вызов draw() каждого элемента
    rotate_all(v,45); // Вызов rotate(45) каждого элемента

    for(auto p : v)   // Не забудьте удалить элементы
        delete p;
}
```

Очевидно, что пример упрощен (особенно в отношении обработки ошибок), но он наглядно иллюстрирует, что `user()` не имеет абсолютно никакого представления о том, с какими видами фигур он работает. Код `user()` может быть скомпилирован один раз и позже использоваться для новых видов фигур, добавленных в программу. Обратите внимание, что нет никаких указателей на фигуры вне `user()`, поэтому за освобождение объектов отвечает сама функция. Это делается с помощью оператора `delete`, полагаясь при этом на виртуальный деструктор `Shape`. Поскольку деструктор является виртуальным, `delete` вызывает деструктор для наиболее позднего производного класса. Это имеет решающее значение, поскольку производный класс может захватывать всевозможные ресурсы (такие, как дескрипторы файлов, блокировки и выходные потоки), которые должны быть освобождены. В нашем

случае Smiley удаляет свои объекты eyes и mouth. Как только это сделано, вызывается деструктор Circle. Объекты строятся конструкторами "снизу вверх" (сначала — базовый), а уничтожаются деструкторами "сверху вниз" (сначала — производные).

4.5.2. Навигация по иерархии

Функция read_shape() возвращает Shape*, так что мы можем рассматривать все фигуры единообразно. Однако что нам делать, если мы захотим использовать функцию-член, имеющуюся только в определенном производном классе, как, например, wink() в классе Smiley? Мы можем запросить, "является ли данная фигура Smiley?", воспользовавшись оператором dynamic_cast:

```
Shape* ps{read_shape(cin)};
if (Smiley*p = dynamic_cast<Smiley*>(ps)) // ps указывает на Smiley?
{
    // Да, это Smiley; используем его
}
else
{
    // Нет, это не Smiley, делаем что-то иное
}
```

Если во время выполнения объект, на который указывает аргумент dynamic_cast (в данном случае — ps) не имеет ожидаемого типа (в данном случае — Smiley) или типа класса, производного от ожидаемого, то dynamic_cast возвращает nullptr.

Мы используем приведение dynamic_cast к типу указателя, когда аргумент представляет собой корректный указатель. Затем мы проверяем, является ли результат равным nullptr. Эту проверку часто удобно помещать в инициализацию переменной в условии.

Мы можем использовать dynamic_cast и для приведения к ссылочному типу. Если объект не относится к ожидаемому типу, dynamic_cast в этом случае генерирует исключение bad_cast:

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)}; // Может генерировать
                                        // исключение std::bad_cast
```

Код будет более чистым, если dynamic_cast используется с ограничениями. Если мы сможем избежать использования информации о типе, то сможем написать более простой и эффективный код. Но иногда информация о типе теряется и должна быть восстановлена. Обычно это происходит, когда мы передаем объект какой-либо системе, которая принимает интерфейс, определенный базовым классом. Когда эта система позже возвращает объект обратно,

нам, возможно, потребуется восстановить его исходный тип. Операции, похожие на dynamic_cast, известны как операции "является разновидностью" или "является экземпляром".

4.5.3. Избежание утечки ресурсов

Опытные программисты должны были заметить, что я оставил открытыми три возможности для ошибок.

- Программист, реализующий Smiley, может забыть выполнить delete для указателя на mouth.
- Пользователь read_shape() может забыть удалить возвращаемый указатель.
- Владелец контейнера указателей на объекты Shape может забыть выполнить удаление объектов, на которые они указывают.

В этом смысле указатели на объекты, размещенные в свободной памяти, оказываются опасными: "обычный старый указатель" не должен использоваться для представления владения. Например:

```
void user(int x)
{
    Shape* p = new Circle{Point{0,0},10};
    // ...
    if (x<0) throw Bad_x{}; // Потенциальная утечка
    if (x==0) return;       // Потенциальная утечка
    // ...
    delete p;
}
```

Если только x не является положительным числом, мы получаем утечку. Присваивать результат выполнения оператора new "голому указателю" означает напрашиваться на неприятности.

Одним простым решением такого рода проблем является использование интеллектуального указателя стандартной библиотеки unique_ptr (§13.2.1) вместо "голого указателя" там, где требуется удаление:

```
class Smiley : public Circle
{
    // ...
private:
    vector<unique_ptr<Shape>> eyes; // Обычно два глаза
    unique_ptr<Shape> mouth;
};
```

Это пример простой, общей и эффективной методики управления ресурсами (§5.3).

В качестве приятного побочного эффекта этого изменения нам больше не нужно определять деструктор для `Smiley`. Компилятор будет неявно генерировать деструктор, который выполнит требуемое уничтожение `unique_ptr` (§5.3) в векторе. Код, использующий `unique_ptr`, будет таким же эффективным, как и код с обычными указателями.

Теперь рассмотрим пользователей `read_shape()`:

```
// Читаем описания фигур из входного потока is:
unique_ptr<Shape> read_shape(istream& is)
{
    // Читаем заголовок фигуры из is и находим ее Kind k
    switch (k)
    {
    case Kind::circle:
        // Читаем данные Circle {Point,int} в p и r
        return unique_ptr<Shape>{new Circle{p,r}}; // §13.2.1
        // ...
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);       // Вызов draw() для каждого элемента
    rotate_all(v,45); // Вызов rotate(45) для каждого элемента
}                      // Все объекты Shape неявно уничтожаются
```

Теперь каждым объектом владеет `unique_ptr`, который удаляет свой объект, когда он больше не нужен, т.е. когда его `unique_ptr` выходит из области видимости.

Чтобы версия `user()` с использованием `unique_ptr` была работоспособна, нам нужны версии функций `draw_all()` и `rotate_all()`, которые принимали бы в качестве аргументов `vector<unique_ptr<Shape>>`. Написание множества таких `_all()`-функций может оказаться очень утомительным занятием; в §6.3.2 показано альтернативное решение.

4.6. Советы

[1] Выражайте идеи непосредственно в коде; §4.1; [CG:P.1].

[2] Конкретный тип является простейшей разновидностью класса. Где это возможно, предпочитайте конкретные типы более сложным классам и простым структурам данных; §4.2; [CG:C.10].

[3] Для представления простых концепций используйте конкретные классы; §4.2.

[4] Для компонентов, критичных с точки зрения производительности, предпочитайте конкретные классы иерархиям классов; §4.2.

[5] Для управления инициализацией объектов определяйте конструкторы; §4.2.1, §5.1.1; [CG:C.40] [CG:C.41].

[6] Делайте функцию членом только тогда, когда необходим непосредственный доступ к представлению класса; §4.2.1; [CG:C.4].

[7] Определяйте операторы, в первую очередь, для имитации обычного применения; §4.2.1; [CG:C.160].

[8] Используйте свободные функции для симметричных операторов; §4.2.1; [CG:C.161].

[9] Объявляйте функции-члены, которые не модифицируют объект, как `const`; §4.2.1.

[10] Если конструктор захватывает ресурс, необходим деструктор, освобождающий этот ресурс[2]; §4.2.2; [CG:C.20].

[11] Избегайте операций с "голыми" `new` и `delete`; §4.2.2; [CG:R.11].

[12] Для управления ресурсами используйте дескрипторы ресурсов и идиому RAII; §4.2.2; [CG:R.1].

[13] Если класс представляет собой контейнер, снабдите его конструктором на основе списка инициализации; §4.2.3; [CG:C.103].

[14] Когда требуется полное разделение интерфейса и реализации, используйте в качестве интерфейсов абстрактные классы; §4.3; [CG:C.122].

[15] Обращайтесь к полиморфным объектам с помощью указателей и ссылок; §4.3.

[16] Абстрактный класс обычно не нуждается в конструкторе; §4.3; [CG:C.126].

[17] Используйте иерархии классов для представления концепций с иерархической структурой; §4.5.

[18] Класс с виртуальной функцией должен иметь виртуальный деструктор; §4.5; [CG:C.127].

[19] Используйте ключевое слово `override` для явного указания перекрытия в больших иерархиях классов; §4.5.1; [CG:C.128].

[20] При проектировании иерархии классов различайте наследование реализации и наследование интерфейса; §4.5.1; [CG:C.129].

[21] Используйте `dynamic_cast` там, где навигация по иерархии классов неизбежна; §4.5.2; [CG:C.146].

---

[2] Так как ресурс может захватываться не только в конструкторе, деструктор должен освобождать *все* захваченные за время жизни объекта ресурсы. — *Примеч. ред.*

[22] Используйте `dynamic_cast` для ссылочного типа там, где невозможность найти необходимый класс рассматривается как ошибка; §4.5.2; [CG:C.147].

[23] Используйте `dynamic_cast` для типа указателя там, где невозможность найти необходимый класс рассматривается как корректный вариант; §4.5.2; [CG:C.148].

[24] Используйте `unique_ptr` или `shared_ptr`, чтобы не забывать удалять объекты, созданные с помощью оператора `new`; §4.5.3; [CG:C.149].

5

Основные операции

Когда кто-то говорит "Я хочу язык программирования,
в котором достаточно просто сказать,
что я хочу сделать", дайте ему леденец.

— Алан Перлис

5.1. Введение

Некоторые операции, такие как инициализация, присваивание, копирование и перемещение, являются фундаментальными в том смысле, что правила языка делают о них определенные предположения. Другие операции, такие как == и <<, имеют обычный стандартный смысл, который опасно игнорировать.

5.1.1. Основные операции

Построение объектов играет ключевую роль во многих проектах. Широкий спектр их применения отражается в диапазоне и гибкости языковых возможностей для поддержки инициализации.

Конструкторы, деструкторы, операции копирования и перемещения для типа логически нераздельны. Мы должны определять их как согласованный набор или в противном случае страдать от логических проблем или проблем с производительностью. Если класс `X` имеет деструктор, который выполняет нетривиальную задачу, такую как освобождение памяти или блокировки, классу, вероятно, потребуется полный набор функций:

```
class X
{
public:
    X(Sometype);            // Обычный конструктор: создание объекта
    X();                    // Конструктор по умолчанию
    X(const X&);            // Копирующий конструктор
    X(X&&);                 // Перемещающий конструктор
    X& operator=(const X&); // Копирующее присваивание:
                            // очистка целевого объекта и копирование
    X& operator=(X&&);      // Перемещающее присваивание:
                            // очистка целевого объекта и перемещение
    ~X();                   // Деструктор: очистка объекта
    // ...
};
```

Имеется пять ситуаций, когда объект может быть скопирован или перемещен: когда он выступает в роли

- источника присваивания;
- инициализатора объекта;
- аргумента функции;
- возвращаемого значения функции;
- исключения.

Присваивание использует оператор копирующего или перемещающего присваивания. В принципе, в других случаях используется копирующий или перемещающий конструктор. Однако вызов копирующего или перемещающего конструктора часто отбрасывается при оптимизации путем конструирования объекта, используемого для инициализации справа, непосредственно в целевом объекте. Например:

```
X make(Sometype);
X x = make(value);
```

В этой ситуации компилятор обычно создает X из make() непосредственно в x, тем самым отменяя (аннулируя) копирование (copy elision).

В дополнение к инициализации именованных объектов и объектов в свободной памяти конструкторы используются для инициализации временных объектов и для реализации явного преобразования типов.

За исключением "обычного конструктора", эти специальные функции-члены генерируются компилятором по мере необходимости. Если вы хотите явно затребовать генерацию реализаций по умолчанию, можете сделать это следующим образом:

```
class Y
{
public:
    Y(Sometype);
    Y(const Y&) = default; // Я хочу, чтобы компилятор сгенерировал
                           // копирующий конструктор по умолчанию
    Y(Y&&)      = default; // и перемещающий конструктор по умолчанию
    // ...
};
```

Если вы явным образом указали генерацию некоторых специальных функций по умолчанию, то прочие определения по умолчанию генерироваться не будут.

Если у класса есть член-указатель, обычно рекомендуется явно записывать операции копирования и перемещения. Причина в том, что указатель может указывать на нечто, что класс должен удалить, и в этом случае выполняемое по умолчанию почленное копирование будет неправильным. Или напротив — член может указывать на то, что класс *не* должен удалять. В любом случае читателю кода хотелось бы это точно знать. Пример можно найти в §5.2.1.

Хорошее эмпирическое правило (иногда называемое *правилом нуля*) состоит в том, чтобы определить либо все основные операции, либо ни одну из них (используя генерацию по умолчанию для всех них). Например:

```
struct Z
{
    Vector v;
    string s;
};
Zz1;        // Инициализация z1.v и z1.s по умолчанию
Z z2 = z1; // Копирование z1.v и z1.s по умолчанию
```

В этом случае компилятор при необходимости будет генерировать почленные конструктор, копирующий конструктор, перемещающий конструктор и деструктор по умолчанию, и все они будут иметь правильную семантику.

В дополнение к =default имеется конструкция =delete, указывающая, что данная операция не должна генерироваться. Базовый класс в иерархии

классов является классическим примером того, где не следует разрешать почленное копирование. Например:

```
class Shape
{
public:
    Shape(const Shape&) =delete; // Копирование запрещено
    Shape& operator=(const Shape&) =delete;
    // ...
};
void copy(Shape& s1, const Shape& s2)
{
    s1 = s2; // Ошибка: копирование Shape не разрешено
}
```

Конструкция `=delete` приводит к тому, что попытка использования удаленной функции является ошибкой времени компиляции; `=delete` может использоваться для подавления любой функции, а не только важных функций-членов.

5.1.2. Преобразования типов

Конструктор, получающий единственный аргумент, определяет операцию преобразования из типа аргумента. Например, `complex` (§4.2.1) предоставляет конструктор от `double`:

```
complex z1 = 3.14; // z1 становится равным {3.14,0.0}
complex z2 = z1*2; // z2 становится равным z1*{2.0,0} == {6.28,0.0}
```

Такое неявное преобразование иногда оказывается идеальным, но не всегда. Например, `Vector` (§4.2.2) предоставляет конструктор от `int`:

```
Vector v1 = 7; // OK: v1 имеет 7 элементов
```

Такая запись обычно рассматривается как неудачная, и `vector` стандартной библиотеки не допускает такого "преобразования" `int` в `vector`.

Способ избежать этой проблемы заключается в том, чтобы разрешить только явное "преобразование", т.е. мы можем определить конструктор следующим образом:

```
class Vector
{
public:
    explicit Vector(int s); // Неявного преобразования
    // ...                  // int в Vector нет
};
```

Это дает нам

```
Vector v1(7);  // OK: v1 содержит 7 элементов
Vector v2 = 7; // Ошибка: неявного преобразования int в Vector нет
```

Когда дело доходит до преобразований, на `Vector` похоже больше типов, чем на `complex`, поэтому, если нет веской причины поступать иначе, используйте `explicit` для конструкторов, которые принимают один аргумент.

5.1.3. Инициализаторы членов

При определении члена данных класса ему может быть предоставлен инициализатор, именуемый *инициализатором члена по умолчанию*. Рассмотрим версию класса `complex` (§4.2.1):

```
class complex
{
    double re = 0;  // Представление: два double
    double im = 0;  // со значениями по умолчанию 0.0
public:
    // Конструктор complex из двух скаляров {r,i}:
    complex(double r, double i):re{r},im{i}{}

    // Конструктор complex из одного скаляра {r,0}:
    complex(double r):re{r} {}

    complex() {} // complex по умолчанию: {0,0}
    // ...
}
```

Когда конструктор не предоставляет значение, используется значение по умолчанию. Это упрощает код и помогает избежать случайного оставления члена без инициализации.

5.2. Копирование и перемещение

По умолчанию объекты могут быть скопированы. Это справедливо для объектов как пользовательских, так и встроенных типов. По умолчанию смысл копирования состоит в почленном копировании: копируется по отдельности каждый член. Например, используем `complex` из §4.2.1:

```
void test(complex z1)
{
    complex z2 {z1}; // Копирующая инициализация
    complex z3;
    z3 = z2;         // Копирующее присваивание
    // ...
}
```

Теперь `z1`, `z2` и `z3` имеют одно и то же значение, потому что как присваивание, так и инициализация копируют оба члена.

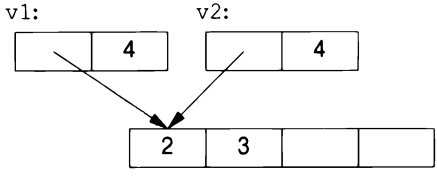
Разрабатывая класс, мы всегда должны рассматривать, можно ли, и как именно, копировать объект. Для простых конкретных типов почленное копирование часто оказывается корректной семантикой копирования. Для некоторых сложных конкретных типов, таких как `Vector`, почленное копирование является неправильной семантикой копирования; для абстрактных типов почленное копирование почти никогда не является правильной семантикой.

5.2.1. Копирование контейнеров

Когда класс является *дескриптором ресурса*, т.е. отвечает за объект, доступ к которому осуществляется через указатель, почленное копирование по умолчанию обычно оказывается катастрофой. Почленная копия будет нарушать инвариант дескриптора ресурса (§3.5.2). Например, копирование по умолчанию создало бы копию `Vector`, ссылающуюся на те же элементы, что и оригинал:

```
void bad_copy(Vector v1)
{
    Vector v2 = v1; // Копирование представления v1 в v2
    v1[0] = 2;      // v2[0] теперь тоже равно 2!
    v2[1] = 3;      // v1[1] теперь тоже равно 3!
}
```

В предположении, что `v1` имеет четыре элемента, графически результат может быть представлен следующим образом.



К счастью, тот факт, что `Vector` имеет деструктор, является важной подсказкой о том, что семантика копирования по умолчанию (почленно) неверна и что компилятор должен хотя бы предупредить об этом. Нам нужно определить лучшую семантику копирования.

Копирование объекта класса определяется двумя членами: *копирующим конструктором* и *копирующим присваиванием*:

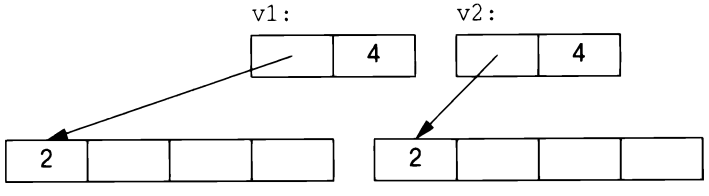
```
class Vector
{
private:
    double* elem; // elem указывает на массив из sz double
    int sz;
public:
    // Конструктор: установка инварианта, захват ресурсов:
    Vector(int s);
```

```
    // Деструктор: освобождение ресурсов:
    ~Vector(){delete[] elem;}

    Vector(const Vector& a);            // Копирующий конструктор
    Vector& operator=(const Vector& a); // Копирующее присваивание

    double& operator[](int i);
    const double& operator[](int i) const;
    int size() const;
};
```

Корректное определение копирующего конструктора для `Vector` выделяет пространство для необходимого количества элементов, а затем копирует в него элементы, так что после копирования каждый вектор имеет собственную копию элементов:

```
Vector::Vector(const Vector& a) // Копирующий конструктор
:elem {new double[a.sz]},       // Выделение памяти для элементов
sz {a.sz}
{
    for (int i=0; i!=sz; ++i)   // Копирование элементов
        elem[i] = a.elem[i];
}
```

Теперь результат примера `v2=v1` может быть представлен следующим образом.



Конечно, в дополнение к копирующему конструктору нам требуется копирующее присваивание:

```
Vector& Vector::operator=(const Vector&a)// Копирующее присваивание
{
    double* p = new double[a.sz];
    for (int i=0; i!=a.sz; ++i)
        p[i] = a.elem[i];
    delete[] elem;                      // Удаление старых элементов
    elem = p;
    sz = a.sz;
    return *this;
}
```

Имя `this` в функциях-членах является предопределенным указателем на объект, для которого вызывается эта функция-член.

5.2.2. Перемещение контейнеров

Мы можем управлять копированием, определяя копирующий конструктор и копирующее присваивание, но само копирование может быть дорогостоящим для больших контейнеров. Мы избегаем затрат на копирование, когда передаем объекты функции с использованием ссылок, но не можем вернуть ссылку на локальный объект в качестве результата (локальный объект будет уничтожен к тому моменту, когда у вызывающей функции появится возможность взглянуть по ссылке). Рассмотрим следующий код:

```
Vector operator+(const Vector& a, const Vector& b)
{
    if (a.size()!=b.size())
        throw Vector_size_mismatch{};
    Vector res(a.size());
    for (int i=0; i!=a.size(); ++i)
        res[i]=a[i]+b[i];
    return res;
}
```

Возврат из оператора + включает копирование результата из локальной переменной `res` в некоторое место, где к нему сможет обратиться вызывающая функция. Мы можем использовать этот оператор + следующим образом:

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
    Vector r;
    // ...
    r = x+y+z;
    // ...
}
```

Это приведет к копированию `Vector` по крайней мере дважды (по одному для каждого использования оператора +). Если вектор большой — скажем, 10 000 `double`, — это может смущать. Больше всего смущает то, что `res` в `operator+()` после копирования никогда не используется вновь. В действительности нам не нужно копирование; мы просто хотим получить результат из функции. По сути, мы хотим *переместить* вектор, а не *скопировать* его. К счастью, мы можем указать свое намерение:

```
class Vector
{
    // ...
    Vector(const Vector& a);             // Копирующий конструктор
    Vector& operator=(const Vector& a); // Копирующее присваивание
    Vector(Vector&& a);                  // Перемещающий конструктор
    Vector& operator=(Vector&& a);       // Перемещающее присваивание
};
```

При наличии такого определения, чтобы реализовать передачу возвращаемого значения из функции, компилятор выберет перемещающий конструктор. Это означает, что `r=x+y+z` не будет содержать копирования векторов; вместо этого векторы просто перемещаются.

Как обычно, определение перемещающего конструктора `Vector` тривиально:

```
Vector::Vector(Vector&& a)
:elem{a.elem},          // "забираем" элементы из a
sz{a.sz}
{
    a.elem = nullptr; // Теперь в a нет элементов
    a.sz = 0;
}
```

`&&` означает "ссылка на r-значение" и является ссылкой, с которой мы можем связать r-значение. Слово "r-значение" (rvalue) является дополнением к l-значению (lvalue), которое грубо означает "нечто, что может появиться в левой части присваивания". Таким образом, r-значение в первом приближении является значением, которому нельзя выполнить присваивание, например целое число, возвращаемое вызовом функции. Таким образом, r-ссылка является ссылкой на нечто, чему *никто иной* ничего не может присвоить, поэтому мы можем безопасно "украсть" ее значение. Примером может служить локальная переменная `res` в `operator+()` в примере с `Vector`.

Перемещающий конструктор *не* принимает константный аргумент: в конечном счете перемещающий конструктор должен удалить значение из своего аргумента. Аналогично определяется и перемещающее присваивание.

Операция перемещения применяется, когда r-ссылка используется в качестве инициализатора или как правая сторона присваивания.

После перемещения перемещаемый объект должен находиться в состоянии, которое позволяет выполнить деструктор. Как правило, мы также разрешаем присваивание перемещенному объекту. Это предполагают и алгоритмы стандартной библиотеки (глава 12, "Алгоритмы"). Наш `Vector` поступает именно таким образом.

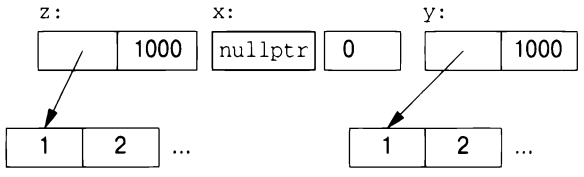
Когда программист знает, что значение больше не будет использоваться, но компилятор недостаточно умен, чтобы это понять, программист может указать ему на этот факт:

```
Vector f()
{
    Vector x(1000);
    Vector y(2000);
    Vector z(3000);
    // Получаем копию (x может использоваться в f() позже):
    z = x;
```

```
    // Выполняем перемещение (перемещающее присваивание):
    y = std::move(x);
    // ... Здесь x лучше не использовать ...
    return z; // Используется перемещение
}
```

Функция стандартной библиотеки `move()` в действительности ничего не делает; она просто возвращает ссылку на свой аргумент, для которой возможно перемещение — *r-ссылку*; это всего лишь разновидность приведения (§4.2.3).

Непосредственно перед `return` мы имеем следующее.



Когда мы возвращаемся из `f()`, переменная `z` уничтожается, после того как ее элементы перемещены из `f()` оператором `return`. Однако деструктор `y` выполняет `delete[]` для его элементов.

Компилятор обязан (согласно стандарту C++) устранить большинство копирований, связанных с инициализацией, поэтому перемещающие конструкторы вызываются не так часто, как вы могли себе представить. Такое *исключение копирования* (copy elision) убирает даже незначительные накладные расходы. С другой стороны, неявно исключить копирование или перемещение из присваиваний, как правило, невозможно; поэтому перемещающее присваивание может иметь решающее значение для производительности.

5.3. Управление ресурсами

Определяя конструкторы, операции копирования, перемещения и деструктор, программист может обеспечить полный контроль над временем жизни содержащегося в объекте ресурса (такого, как элементы контейнера). Кроме того, перемещающий конструктор позволяет объекту легко и дешево перемещаться из одной области видимости в другую. Таким образом, объекты, которые мы не можем или не хотим копировать из области видимости, могут быть просто и дешево перемещены из нее. Рассмотрим класс `thread` стандартной библиотеки, используемый для представления параллельных вычислений (§15.2), и вектор из миллиона `double`. Мы не можем копировать первый объект и не хотим копировать последний.

```
std::vector<thread> my_threads;
Vector init(int n)
{
    // Параллельное выполнение heartbeat (в отдельном потоке):
```

```
    thread t{heartbeat};

    // Перемещение t в my_threads (§13.2.2):
    my_threads.push_back(std::move(t));

    // ... Другие инициализации ...
    Vector vec(n);

    for (int i=0; i!=vec.size(); ++i)
        vec[i] = 777;
    return vec; // Перемещение vec из init()
}
auto v = init(1'000'000); // Запуск heartbeat и инициализация v
```

Во многих случаях дескрипторы ресурсов, такие как `Vector` и `thread`, являются превосходными альтернативами непосредственному использованию встроенных указателей. Фактически интеллектуальные указатели стандартной библиотеки, такие как `unique_ptr`, тоже являются дескрипторами ресурсов (§13.2.1).

Я использовал `vector` стандартной библиотеки для хранения потоков `thread`, потому что мы не можем параметризовать наш простой `Vector` типом элементов (до §6.2).

Во многом так же, как `new` и `delete` исчезают из кода приложения, мы можем заставить исчезнуть в дескрипторах ресурсов указатели. В обоих случаях результат представляет собой более простой и удобный в обслуживании код без дополнительных накладных расходов. В частности, мы можем добиться *строгой безопасности ресурсов*, т.е. устранить утечки ресурсов для общего понятия ресурса. Примерами являются объекты `vector`, хранящие память, `thread`, хранящие системные потоки, и `fstream`, хранящие файлы.

Во многих языках управление ресурсами в основном делегируется сборщику мусора. C++ также предлагает интерфейс сборки мусора, так что вы можете подключить сборщик мусора. Однако лично я считаю сборку мусора последним вариантом — после исчерпания более ясных, более общих и лучше локализованных альтернатив управления ресурсами. Мой идеал заключается в том, чтобы не создавать мусор, тем самым устраняя необходимость в сборщике мусора: не мусори!

Сборка мусора фундаментально представляет собой глобальную схему управления памятью. Умные реализации могут компенсировать эту глобальность, но по мере того, как системы становятся все более и более распределенными (вспомните о кешах, многоядерности и кластерах), локальность становится важнее, чем когда-либо ранее.

Кроме того, память не является единственным ресурсом. Ресурс — это все, что должно быть захвачено и (явно или неявно) освобождено после ис-

пользования. Примерами ресурсов являются память, блокировки, сокеты, дескрипторы файлов и потоков. Нет никакого сюрприза в том, что ресурс, который является не просто памятью, называется *ресурсом, не являющимся памятью* (non-memory resource). Хорошая система управления ресурсами обрабатывает все разновидности ресурсов. Утечек следует избегать в любой системе, работающей длительное время, но чрезмерное удержание ресурсов может быть почти таким же плохим, как и утечка. Например, если система удерживает память, блокировки, файлы и тому подобное в два раза дольше, чем требуется, — потенциально такой системе требуется вдвое большее количество ресурсов.

Прежде чем прибегать к сборке мусора, систематически используйте дескрипторы ресурсов: пусть каждый ресурс имеет владельца в некоторой области видимости и по умолчанию освобождается в конце области видимости его владельца. В C++ это называется идиомой *RAII* (Resource Acquisition Is Initialization — захват ресурса есть инициализация) и интегрируется с обработкой ошибок в форме исключений. Ресурсы могут перемещаться из одной области видимости в другую с помощью семантики перемещения или "интеллектуальных указателей", а совместное владение может быть представлено "совместно используемым указателем" (§13.2.1).

В стандартной библиотеке C++ идиома RAII очень распространена для разных видов ресурсов, например для памяти (`string`, `vector`, `map`, `unordered_map` и т.д.), файлов (`ifstream`, `ofstream` и т.д.), потоков выполнения (`thread`), блокировок (`lock_guard`, `unique_lock` и т.д.) и объектов общего назначения (через `unique_ptr` и `shared_ptr`). Результатом является неявное управление ресурсами, невидимое при обычном использовании и приводящее к малой продолжительности захвата ресурсов.

5.4. Обычные операции

Некоторые операции при их определении для типа имеют обычный смысл. Этот обычный смысл часто предполагается программистами и библиотеками (в частности, стандартной библиотекой), поэтому разумно соответствовать ему при разработке новых типов, для которых эти операции имеют смысл.

- Сравнения: ==, !=, <, <=, > и >= (§5.4.1).
- Операции с контейнерами: `size()`, `begin()` и `end()` (§5.4.2).
- Операции ввода-вывода: >> и << (§5.4.3).
- Пользовательские литералы (§5.4.4).
- `swap()` (§5.4.5).
- Хеш-функции: `hash<>` (§5.4.6).

5.4.1. Сравнения

Смысл сравнения на равенство (== и !=) тесно связан с копированием. После копирования копии должны рассматриваться как равные (эквивалентные) объекты:

```
X a = нечто;
X b = a;
assert(a==b); // Если здесь a!=b, "нечто" очень странное (§3.5.4)
```

Одновременно с == определите также != и убедитесь, что a!=b означает !(a==b).

Аналогично, если вы определяете оператор <, то определите также операторы <=, >, >= и убедитесь в выполнении обычных тождеств:

- a<=b означает (a<b)||(a==b) и !(b<a).
- a>b означает b<a.
- a>=b означает (a>b)||(a==b) и !(a<b).

Чтобы оба операнда бинарного оператора трактовались одинаково, его лучше всего определить как свободную функцию в пространстве имен соответствующего класса. Например:

```
namespace NX
{
    class X
    {
        // ...
    };
    bool operator==(const X&, const X&);
    // ...
};
```

5.4.2. Операции с контейнерами

Если только у вас нет по-настоящему веской причины поступать иначе, проектируйте контейнеры в стиле контейнеров стандартной библиотеки (глава 11, "Контейнеры"). В частности, сделайте контейнер безопасным с точки зрения ресурсов, реализуя его как дескриптор с соответствующими существенными операциями (§5.1.1, §5.2).

Контейнеры стандартной библиотеки знают количество хранящихся в них элементов, и мы можем получить его с помощью вызова size(). Например:

```
for(size_t i=0; i<c.size(); ++i)  // size_t – имя возвращаемого типа
    c[i] = 0;                     // size() в стандартной библиотеке
```

Однако вместо обхода контейнеров с использованием индексов от 0 до size() стандартные алгоритмы (глава 12, "Алгоритмы") полагаются на понятие *последовательностей*, ограниченных парами *итераторов*:

```
for (auto p = c.begin(); p != c.end(); ++p)
    *p = 0;
```

Здесь c.begin() является итератором, указывающим на первый элемент c, а c.end() указывает на элемент за последним элементом c. Подобно указателям, итераторы поддерживают операторы ++ для перехода к следующему элементу и * — для доступа к значению указываемого элемента. Такая *модель итератора* (§12.3) позволяет обеспечить большую обобщенность и эффективность. Итераторы используются для передачи последовательностей алгоритмам стандартной библиотеки. Например:

```
sort(v.begin(),v.end());
```

С подробностями и другими операциями с контейнерами можно ознакомиться в главах 11, "Контейнеры", и 12, "Алгоритмы".

Еще одно средство неявного обхода всех элементов контейнера — цикл for по диапазону:

```
for (auto& x : c)
    x = 0;
```

Здесь c.begin() и c.end() используются неявно; этот цикл грубо эквивалентен более явному циклу.

5.4.3. Операции ввода-вывода

Для пары целочисленных операндов оператор << означает сдвиг влево, а >> — сдвиг вправо. Однако для потоков iostream это операторы вывода и ввода соответственно (§1.8, глава 10, "Ввод и вывод"). С подробностями и другими операциями ввода-вывода можно ознакомиться в главе 10, "Ввод и вывод".

5.4.4. Пользовательские литералы

Одна из целей классов заключалась в том, чтобы позволить программисту проектировать и реализовывать типы, точно имитирующие встроенные типы. Конструкторы обеспечивают инициализацию, которая эквивалентна (или даже превосходит) гибкости и эффективности встроенной инициализации типа, но для встроенных типов мы имеем также соответствующие литералы.

- 123 представляет собой int.
- 0xFF00u представляет собой unsigned int.

- `123.456` представляет собой `double`.
- `"Surprise!"` представляет собой `const char[10]`.

Может быть полезным предоставить подобные литералы для пользовательских типов. Это делается путем определения смысла подходящего суффикса литерала, поэтому мы можем получить следующее.

- `"Surprise!"s` представляет собой `std::string`.
- `123s` представляет собой секунды.
- `12.7i` является мнимым числом, так что `12.7i+47` представляет собой комплексное число (в данном случае — `{47,12.7}`).

В частности, в стандартной библиотеке с использованием соответствующих заголовочных файлов и пространств имен мы получаем доступ к следующим суффиксам.

Суффиксы стандартной библиотеки для литералов		
`<chrono>`	`std::literals::chrono_literals`	`h, min, s, ms, us, ns`
`<string>`	`std::literals::string_literals`	`s`
`<string_view>`	`std::literals::string_literals`	`sv`
`<complex>`	`std::literals::complex_literals`	`i, il, if`

Неудивительно, что литералы с такими определяемыми пользователем суффиксами называются *пользовательскими литералами* (user-defined literal — UDL). Они определяются с использованием *литеральных операторов*. Литеральный оператор преобразует литерал типа своего аргумента в его возвращаемый тип. Например, суффикс `i` для мнимых чисел может быть реализован следующим образом:

```
// Мнимые числа:
constexpr complex<double> operator""i(long double arg)
{
    return {0,arg};
}
```

Здесь

- `operator""` указывает, что мы определяем литеральный оператор;
- `i` после индикатора литерального оператора `""` представляет собой суффикс, для которого оператор определяет его значение;
- тип аргумента, `long double`, указывает, что суффикс (`i`) определен для литералов с плавающей точкой;

- возвращаемый тип, `complex<double>`, указывает тип возвращаемого значения.

С учетом сказанного мы можем написать

```
complex<double> z = 2.7182818+6.283185i;
```

5.4.5. swap()

Многие алгоритмы, в частности `sort()`, используют функцию `swap()`, которая выполняет обмен значений двух объектов. Такие алгоритмы обычно предполагают, что функция `swap()` очень быстрая и не генерирует исключений. Стандартная библиотека предоставляет `std::swap(a,b)`, реализованную как три операции перемещения: `(tmp = a, a = b, b = tmp)`. Если вы создаете тип, дорогостоящий для копирования, но такой, которой может быть корректно обменен (например, функцией сортировки), обеспечьте для него операции перемещения или `swap()` (или обе). Обратите внимание, что контейнеры стандартной библиотеки (глава 11, "Контейнеры") и `string` (§9.2.1) имеют операции быстрого перемещения.

5.4.6. hash<>

Класс стандартной библиотеки `unordered_map<K,V>` представляет собой хеш-таблицу с `K` в качестве типа ключа и `V` в качестве типа значения (§11.5). Для использования типа `X` в качестве ключа требуется определение `hash<X>`. Стандартная библиотека делает это для распространенных типов, таких как `std::string`.

5.5. Советы

[1] Управляйте созданием, копированием, перемещением и уничтожением объектов; §5.1.1; [CG:R.1].

[2] Проектируйте конструкторы, присваивания и деструктор как согласованный набор операций; §5.1.1; [CG:C.22].

[3] Определяйте либо все важные операции, либо ни одной; §5.1.1; [CG:C.21].

[4] Если генерируемые по умолчанию конструктор, присваивание и деструктор вам подходят, позвольте компилятору их генерировать (не пишите их самостоятельно); §5.1.1; [CG:C.20].

[5] Если класс имеет член-указатель, вероятно, ему требуются пользовательские (или удаленные) деструктор, копирование и перемещение; §5.1.1; [CG:C.32] [CG:C.33].

[6] Если у класса есть пользовательский деструктор, ему, вероятно, требуются пользовательские или удаленные операции копирования и перемещения; §5.2.1.

[7] По умолчанию объявляйте конструктор с единственным аргументом как `explicit`; §5.1.1; [CG:C.46].

[8] Если член класса имеет подходящее значение по умолчанию, предоставьте соответствующий инициализатор члена-данных; §5.1.3; [CG:C.48].

[9] Переопределите или запретите копирование, если копирование по умолчанию не годится для вашего типа; §5.2.1, §4.6.5; [CG:C.61].

[10] Возвращайте контейнеры по значению (полагаясь на эффективное перемещение); §5.2.2; [CG:F.20].

[11] Для больших операндов используйте в качестве типов аргументов константные ссылки; §5.2.2; [CG:F.16].

[12] Обеспечьте строгую безопасность ресурсов — никогда не допускайте утечки того, что можно рассматривать как ресурс; §5.3; [CG:R.1].

[13] Если класс представляет собой дескриптор ресурса, ему требуются пользовательские конструктор, деструктор и операции копирования; §5.3; [CG:R.1].

[14] Перегружайте операции, имитируя их обычное использование; §5.4; [CG:C.160].

[15] Следуйте принципам проектирования контейнеров стандартной библиотеки; §5.4.2; [CG:C.100].

6

Шаблоны

Место для вашей цитаты.

— Б. Страуструп

6.1. Введение

Тому, кому нужен вектор, вряд ли всегда будет нужен именно вектор `double`. Вектор — это общая концепция, не зависящая от понятия числа с плавающей точкой. Следовательно, тип элемента вектора должен быть представлен независимо от самого вектора. *Шаблон* — это класс или функция, которую мы параметризуем с помощью набора типов или значений. Мы используем шаблоны для представления идей, которые лучше всего воспринимаются как нечто общее, из которого мы можем генерировать конкретные типы и функции, указывая аргументы, такие как тип `double` элемента `vector`.

6.2. Параметризованные типы

Мы можем обобщить наш вектор элементов типа `double` до вектора элементов произвольного типа, сделав его шаблоном и заменив конкретный тип `double` параметром типа. Например:

```
template<typename T>
class Vector
{
private:
    T* elem;  // elem указывает на массив из sz элементов типа T
    int sz;
public:
    // Конструктор; установка инварианта, захват ресурсов:
    explicit Vector(int s);
    // Деструктор: освобождение захваченных ресурсов:
    ~Vector() { delete[] elem; }

    // ... Операции копирования и перемещения ...

    T& operator[](int i);              // Для неконстантного Vector
    const T& operator[](int i) const;  // Для const Vector (§4.2.1)

    int size() const { return sz; }
};
```

Префикс `template<typename T>` делает `T` параметром объявления, префиксом которого он является. Это C++-версия математического выражения ∀T — "для всех T" (или, точнее, "для всех типов `T`"). Если вы хотите выразить математическое выражение "для всех T, таких, что P(T)", вам нужны концепты (§6.2.1, §7.2). Применение ключевого слова `class` для введения параметра типа эквивалентно использованию `typename`, и в старом коде часто встречается префикс `template<class T>`.

Функции-члены могут быть определены аналогично:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0)
        throw Negative_size{};
    elem = new T[s];
    sz = s;
}
template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

С учетом вышесказанного можно определить объекты Vector следующим образом:

```
Vector<char> vc(200);       // Вектор из 200 символов
Vector<string> vs(17);      // Вектор из 17 строк
Vector<list<int>> vli(45);  // Вектор из 45 списков целых чисел
```

Пара символов >> в Vector<list<int>> завершает вложенные аргументы шаблона; это не ошибочно размещенный оператор ввода.

Использовать Vector можно следующим образом:

```
void write(const Vector<string>& vs) // Vector некоторых строк
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Чтобы обеспечить поддержку цикла for по диапазону для нашего Vector, мы должны определить подходящие функции begin() и end():

```
template<typename T>
T* begin(Vector<T>& x)
{
    // Указатель на первый элемент или nullptr:
    return x.size() ? &x[0] : nullptr;
}

template<typename T>
T* end(Vector<T>& x)
{
    // Указатель за последний элемент:
    return x.size() ? &x[0]+x.size() : nullptr;
}
```

Теперь можно записать:

```
void f2(Vector<string>& vs) // Vector некоторых строк
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

Аналогично можно определить списки, векторы, отображения (т.е. ассоциативные массивы), неупорядоченные отображения (т.е. хеш-таблицы) и тому подобное как шаблоны (глава 11, "Контейнеры").

Шаблоны представляют собой механизм времени компиляции, поэтому их использование не требует дополнительных затрат времени по сравнению с написанным вручную кодом. Фактически код, созданный для Vector <double>, идентичен коду, сгенерированному для версии Vector из главы 4, "Классы". Кроме того, код, сгенерированный для vector<double> из стан-

дартной библиотеки, скорее всего, будет лучшего качества (потому что в его реализацию вложено больше усилий).

Шаблон плюс набор аргументов шаблона называется *инстанцированием* или *специализацией*. В конце процесса компиляции, во *время инстанцирования*, генерируется код для каждого инстанцирования, используемого в программе (§7.5). Для сгенерированного кода выполняются проверки типов, так что этот код оказывается таким же безопасным с точки зрения типов, как и написанный вручную. К сожалению, такая проверка часто выполняется в конце процесса компиляции, во время инстанцирования.

6.2.1. Ограниченные аргументы шаблона (C++20)

Чаще всего шаблон имеет смысл только для аргументов шаблона, которые отвечают определенным критериям. Например, `Vector` обычно предлагает операцию копирования и, если это так, должен требовать, чтобы его элементы были копируемы. То есть мы должны требовать, чтобы аргумент шаблона `Vector` был не просто `typename`, а `Element`, где "`Element`" определяет требования к типу, которому могут принадлежать элементы:

```
template<Element T>
class Vector
{
private:
    T* elem;  // elem указывает на массив из sz элементов типа T
    int sz;
    // ...
};
```

Этот префикс `template<Element T>` представляет собой C++-версию математического выражения "для всех `T`, таких, что `Element(T)`"; т.е. `Element` является предикатом, который проверяет, обладает ли `T` всеми свойствами, которые требует `Vector`. Такой предикат называется *концептом* (concept) (§7.2). Шаблонный аргумент, для которого указан концепт, называется *ограниченным аргументом*, а шаблон, для которого аргумент ограничен, — *ограниченным шаблоном*.

Попытка инстанцирования шаблона с типом, не соответствующим его требованиям, является ошибкой времени компиляции. Например

```
Vector<int> v1;    // OK: можно копировать int
Vector<thread> v2; // Ошибка: нельзя копировать
                   // стандартный thread (§15.2)
```

Поскольку официально до стандарта C++20 язык программирования C++ концепты не поддерживает, более старый код использует неограниченные аргументы шаблона и указывает необходимые требования в документации.

6.2.2. Аргументы-значения шаблонов

В дополнение к аргументам-типам шаблоны могут получать аргументы-значения. Например:

```
template<typename T, int N>
struct Buffer
{
    using value_type = T;
    constexpr int size() { return N; }
    T[N];
    // ...
};
```

Псевдоним (`value_type`) и `constexpr`-функция предоставляют пользователям доступ (только для чтения) к аргументам шаблона.

Аргументы-значения полезны во многих контекстах. Например, `Buffer` позволяет нам создавать буфера произвольного размера без использования динамической памяти:

```
// Глобальный буфер символов (статически выделенная память):
Buffer<char,1024> glob;
void fct()
{
    Buffer<int,10> buf; // Локальный буфер целых чисел (в стеке)
    // ...
}
```

Аргумент-значение шаблона должен быть константным выражением.

6.2.3. Вывод аргументов шаблона

Рассмотрим применение шаблона стандартной библиотеки `pair`:

```
pair<int,double> p = {1,5.2};
```

Многим необходимость указывать аргументы-типы шаблона кажется утомительной, поэтому стандартная библиотека предлагает функцию `make_pair()`, которая выводит аргументы возвращаемого ею шаблона `pair` из аргументов функции:

```
auto p = make_pair(1,5.2); // p является pair<int,double>
```

Это приводит к очевидному вопросу "Почему нельзя просто вывести параметры шаблона из аргументов конструктора?" В C++17 это возможно:

```
pair p = {1,5.2};          // p является pair<int,double>
```

Это проблема не только `pair`; функции `make_` очень распространены. Рассмотрим простой пример:

```
template<typename T>
class Vector
{
public:
    Vector(int);
    // Конструктор из списка инициализации:
    Vector(initializer_list<T>);
    // ...
};

// Вывод типа элементов v1 из типа элементов инициализатора:
Vector v1 {1,2,3};

// Вывод типа элементов v2 из типа элементов v1:
Vector v2 = v1;

// p указывает на Vector<int>:
auto p = new Vector {1,2,3};

// Тип элемента должен быть указан явно (он нигде не упоминается):
Vector<int> v3(1);
```

Эта возможность упрощает запись и может устранить раздражение, вызываемое ошибочным вводом избыточных аргументов-типов шаблона. Однако это не панацея. Вывод типов может приводить к сюрпризам (как в функциях `make_`, так и в конструкторах). Рассмотрим следующий фрагмент кода:

```
// Vector<string>:
Vector<string> vs1 {"Hello","World"};

// Вывод Vector<const char*> (Сюрприз?):
Vector vs {"Hello", "World"};

// Вывод Vector<string>:
Vector vs2 {"Hello"s, "World"s};

// Ошибка: негомогенный список инициализаторов:
Vector vs3 {"Hello"s, "World"};
```

Тип строкового литерала в стиле C — `const char*` (§1.7.1). Если это не то, что вы хотели, добавьте суффикс `s`, чтобы сделать его корректной строкой `string` (§9.2). Если элементы списка инициализации имеют различные типы, компилятор не в состоянии вывести единственный тип элементов, и мы получаем сообщение об ошибке.

Когда аргумент шаблона не может быть выведен из аргументов конструктора, мы можем помочь компилятору, предоставляя *правила вывода* (deduction guide). Рассмотрим следующий фрагмент:

```
template<typename T>
class Vector2
{
public:
    using value_type = T;
    // ...

    // Конструктор из списка инициализации:
    Vector2(initializ er_list<T>);

    // Конструктор из диапазона [b:e):
    template<typename Iter>
    Vector2(Iter b, Iter e);

    // ...
};

Vector2 v1 {1,2,3,4,5}; // Тип элементов — int
Vector2 v2(v1.begin(),v1.begin()+2);
```

Очевидно, что v2 должен быть `Vector2<int>`, но без помощи компилятор не может это вывести. В коде указано только, что существует конструктор из пары значений одного и того же типа. Без языковой поддержки концептов (§7.2) компилятор ничего не может предполагать об этом типе. Чтобы разрешить вывод, мы можем добавить правило вывода после объявления `Vector2`:

```
template<typename Iter>
Vector2(Iter,Iter) -> Vector2<typename Iter::value_type>;
```

То есть если компилятор встречает `Vector2`, инициализированный парой итераторов, то он должен вывести `Vector2::value_type` как тип значения итератора.

Применение правил вывода может сопровождаться очень тонкими эффектами, поэтому лучше всего разрабатывать шаблоны классов таким образом, чтобы правила вывода были не нужны. Однако в настоящий момент стандартная библиотека полна классов, которые (пока) не используют концепты (§7.2) и содержат неоднозначности, поэтому в ней используется довольно много правил вывода.

6.3. Параметризованные операции

Шаблоны имеют гораздо больше возможностей, чем простая параметризация контейнера типом элемента. В частности, они широко используются для параметризации типов и алгоритмов стандартной библиотеки (§11.6, §12.6).

Существует три средства выражения операции, параметризованной типами или значениями:

- шаблон функции;
- функциональный объект: объект, который может хранить данные и быть вызванным наподобие функции;
- лямбда-выражение: сокращенная запись функционального объекта.

6.3.1. Шаблоны функций

Мы можем написать функцию, которая вычисляет сумму значений элементов любой последовательности, которую может обойти цикл `for` для диапазона (например, контейнера) следующим образом:

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v)
{
    for (auto x : s)
        v+=x;
    return v;
}
```

Аргумент шаблона `Value` и аргумент функции `v` предназначены для того, чтобы вызывающая функция могла указать тип и начальное значение аккумулятора (переменной, в которой нужно накапливать сумму):

```
void user(Vector<int>& vi, list<double>& ld,
          vector<complex<double>>& vc)
{
    // Сумма вектора элементов типа int (суммирует int):
    int x = sum(vi,0);

    // Сумма вектора элементов типа int (суммирует double):
    double d = sum(vi,0.0);

    // Сумма списка элементов типа double:
    double dd = sum(ld,0.0);

    // Сумма вектора элементов типа complex<double>:
    auto z = sum(vc,complex{0.0,0.0});
}
```

Цель суммирования значений типа `int` с аккумулятором типа `double` может состоять в том, чтобы успешно справиться с числом, бо́льшим, чем наибольшее допустимое значение `int`. Обратите внимание, как из аргументов функции выводятся аргументы типов шаблона для `sum<Sequence,Value>`. К счастью, нам не нужно явно указывать эти типы.

Эта функция `sum()` представляет собой упрощенную версию алгоритма `accumulate()` стандартной библиотеки (§14.3).

Шаблон функции может быть функцией-членом, но не виртуальной функцией — компилятор не в состоянии знать все инстанцирования такого шаблона в программе, а потому не в состоянии корректно генерировать `vtbl` в данной ситуации (§4.4).

6.3.2. Функциональные объекты

Одной особенно полезной разновидностью шаблона является *функциональный объект* (иногда называемый *функтором*), используемый для определения объектов, которые могут быть вызваны с использованием синтаксиса вызова функций. Например:

```
template<typename T>
class Less_than
{
    const T val; // Значение, с которым выполняется сравнение
public:
    Less_than(const T& v) :val{v} { }
    bool operator()(const T& x) const      // Оператор вызова
    {
        return x<val;
    }
};
```

Функция `operator()` реализует оператор "вызова функции" или просто "вызова" — `()`. Можно определить именованные переменные типа `Less_than` для некоторых аргументов типа:

```
Less_than lti{42};                 // lti(i) сравнивает i и 42 (i<42)
Less_than lts{"Backus"s};          // lts(s) сравнивает s<"Backus"
Less_than<string> lts2{"Naur"}; // "Naur" — строка в стиле C, поэтому
                                   // нам нужен <string> для
                                   // правильного оператора <
```

Такой объект можно вызвать так же, как функцию:

```
void fct(int n, const string & s)
{
    bool b1 = lti(n); // true, если n<42
    bool b2 = lts(s); // true, если s<"Backus"
    // ...
}
```

Такие функциональные объекты широко используются в качестве аргументов алгоритмов. Например, можно подсчитать количество вхождений встречающихся значений, для которых предикат возвращает значение `true`:

```
template<typename C, typename P>
// requires Sequence<C> && Callable<P,Value_type<P>>
int count(const C& c, P pred)
```

```
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

Предикат — это нечто, что может быть вызвано для возврата значения `true` или `false`. Например:

```
void f(const Vector<int>& vec, const list<string>& lst,
       int x, const string& s)
{
    cout << "Количество значений, меньших " << x << ": "
         << count(vec,Less_than{x}) << '\n';
    cout << "Количество значений, меньших " << s << ": "
         << count(lst,Less_than{s}) << '\n';
}
```

Здесь `Less_than{x}` создает объект типа `Less_than<int>`, оператор вызова которого выполняет сравнение со значением `int`, переданным в `x`; `Less_than{s}` создает объект, который выполняет сравнение со строкой `s`. Красота этих функциональных объектов заключается в том, что они хранят значение, с которым выполняют сравнение. Нам не нужно писать отдельную функцию для каждого значения (и каждого типа), и не нужно вводить неприятные глобальные переменные для хранения значений. Кроме того, для такого простого функционального объекта, как `Less_than`, компилятор легко выполняет встраивание, а потому вызов `Less_than` оказывается намного эффективнее, чем косвенный вызов функции. Возможность хранения данных и эффективность делают функциональные объекты особенно полезными в качестве аргументов алгоритмов.

Функциональные объекты, используемые для указания смысла ключевых операций общего алгоритма (например, `Less_than` для `count()`), часто называются *объектами стратегии* (policy objects).

6.3.3. Лямбда-выражения

В §6.3.2 мы определили `Less_than` отдельно от его использования. Это может быть неудобно, а потому в языке имеется возможность записи неявного создания функциональных объектов:

```
void f(const Vector<int>& vec, const list<string>& lst,
       int x, const string& s)
{
    cout << "Количество значений, меньших " << x
         << ": " << count(vec,[&](int a){ return a<x; })
         << '\n';
```

```
    cout << "Количество значений, меньших " << s
         << ": " << count(lst,[&](const string& a){ return a<s; })
         << '\n';
}
```

Запись `[&](int a){ return a<x; }` называется *лямбда-выражением*. Оно генерирует функциональный объект точно так же, как и `Less_than<int>{x}`. Конструкция `[&]` представляет собой *список захвата*, указывающий, что локальные имена, используемые в теле лямбда-выражения (такие, как `x`), будут доступны по ссылке. Для того чтобы "захватить" только `x`, мы можем сделать следующее: `[&x]`. Если мы хотим передать генерируемому объекту копию `x`, то для этого должны использовать запись `[=x]`. Запись `[]` указывает, что не захватываются никакие локальные имена; запись для захвата всех локальных имен по ссылке — `[&]`, а запись для захвата всех локальных имен по значению — `[=]`.

Использование лямбда-выражений может быть удобным и кратким, но одновременно и неясным. Для нетривиальных действий (скажем, более чем для простого выражения) я предпочитаю именовать операцию, чтобы более четко изложить ее цель и сделать ее доступной для использования в нескольких местах программы.

В §4.5.3 мы отметили раздражение из-за необходимости писать множество функций для выполнения операций над элементами векторов указателей и `unique_ptr`, таких как `draw_all()` и `rotate_all()`. Функциональные объекты (в частности, лямбда-выражения) могут помочь нам, позволяя отделить обход контейнера от указания того, что должно быть выполнено с каждым элементом.

Во-первых, нам нужна функция, применяющая операцию к каждому объекту, на который указывают элементы контейнера указателей:

```
template<typename C, typename Oper>
void for_all(C& c, Oper op) // Считаем, что C – контейнер указателей
// requires Sequence<C> && Callable<Oper,Value_type<C>> (см. §7.2.1)
{
    for (auto& x : c)
        op(x); // Передача в op() ссылки на каждый элемент
}
```

Теперь мы можем написать версию `user()` из §4.5 без создания множества функций `_all`:

```
void user2()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
```

```
    // draw_all():
    for_all(v,[](unique_ptr<Shape>& ps){ ps->draw(); });

    // rotate_all(45):
    for_all(v,[](unique_ptr<Shape>& ps){ ps->rotate(45); });
}
```

Я передаю в лямбда-выражение `unique_ptr<Shape>&`, так что функция `for_all()` не должна заботиться о том, как именно хранятся объекты. В частности, вызовы `for_all()` не влияют на время жизни переданных `Shape`, а тела лямбда-выражений используют аргумент так же, как если бы он был простым указателем.

Как и функция, лямбда-выражение может быть обобщенным. Например:

```
template<class S>
void rotate_and_draw(vector<S>& v, int r)
{
    for_all(v,[](auto& s){ s->rotate(r); s->draw(); });
}
```

Здесь, как и в объявлениях переменных, `auto` означает, что в качестве инициализатора принимается любой тип (аргумент рассматривается как инициализирующий формальный параметр в вызове). Это делает лямбда-выражение с параметром `auto` шаблоном, обобщенным лямбда-выражением. По причинам, затерявшимся где-то в комитетах по стандартизации, такое использование `auto` в настоящее время не допускается для аргументов функций.

Мы можем вызывать такую обобщенную функцию `rotate_and_draw()` с любым контейнером объектов, для которых допустимы `draw()` и `rotate()`. Например:

```
void user4()
{
    vector<unique_ptr<Shape>> v1;
    vector<Shape*> v2;
    // ...
    rotate_and_draw(v1,45);
    rotate_and_draw(v2,90);
}
```

Используя лямбда-выражение, можно превратить любую инструкцию в выражение. В основном это используется для предоставления операций для вычисления значения в качестве аргумента, но эта способность является более общей. Рассмотрим сложную инициализацию:

```
// Варианты инициализации:
enum class Init_mode { zero, seq, cpy, patrn };
```

```
// Грязный код:
// int n, Init_mode m, vector<int>& arg и итераторы p и q
// определены где-то в другом месте
vector<int> v;

switch (m)
{
case zero:
    v = vector<int>(n); // n элементов инициализированы нулями
    break;
case cpy:
    v = arg;
    break;
};

// ...

if (m == seq)
    v.assign(p,q);      // Копируем из последовательности [p:q)

// ...
```

Это стилизованный пример, но, к сожалению, не атипичный. Нам нужно выбрать один из множества вариантов инициализации структуры данных (здесь v) и нам нужно делать разные вычисления для разных вариантов. Такой код часто беспорядочен и является источником ошибок:

- переменная может использоваться до того, как получит предусмотренное значение;
- "код инициализации" может смешиваться с другим кодом, затрудняя понимание;
- при смешивании "кода инициализации" с другим кодом гораздо проще забыть какой-то из case;
- это не инициализация, а присваивание.

Этот код можно превратить в лямбда-выражение, используемое в качестве инициализатора:

```
// int n, Init_mode m, vector<int>& arg и итераторы p и q
// определены где-то в другом месте
vector<int> v = [&]
{
    switch (m) {
    case zero:
        // n элементов инициализированы нулями:
        return vector<int>(n);
    case seq:
        // Копирование из последовательности [p:q):
        return vector<int>{p,q};
```

```
    case cpy:
        return arg;
    }
}();
// ...
```

Я все еще "забыл" `case`, но теперь это легко заметить.

6.4. Шаблонные механизмы

Для определения хороших шаблонов нужны некоторые поддерживающие возможности языка программирования.

- Значения, зависящие от типа: *шаблоны переменных* (§6.4.1).
- Псевдонимы типов и шаблонов: *шаблоны псевдонимов* (§6.4.2).
- Механизм выбора времени компиляции: `if constexpr` (§6.4.3).
- Механизм времени компиляции для опроса свойств типов и выражений: выражения `requires` (§7.2.3).

Кроме того, в проектировании и использовании шаблонов свою роль зачастую играют `constexpr`-функции (§1.6) и утверждения `static_assert` (§3.5.5).

Эти базовые механизмы являются, в первую очередь, инструментами для построения общих, основополагающих абстракций.

6.4.1. Шаблоны переменных

Когда мы используем тип, нам часто нужны константы и значения этого типа. Это, конечно, имеет место и когда мы используем шаблон класса: когда мы определяем `C<T>`, нам часто нужны константы и переменные типа `T` и других типов, зависящих от `T`. Вот пример из динамического моделирования жидкости [19]:

```
template <class T>
constexpr T viscosity = 0.4;

template <class T>
constexpr space_vector<T> external_acceleration = { T{}, T{-9.8}, T{} };

auto vis2 = 2*viscosity<double>;
auto acc = external_acceleration<float>;
```

Здесь `space_vector` представляет собой трехмерный вектор.

Естественно, в качестве инициализаторов можно использовать произвольные выражения подходящего типа:

```
// is_assignable - свойство типа (§13.9.1):
template<typename T, typename T2>
constexpr bool Assignable = is_assignable<T&,T2>::value;

template<typename T>
void testing()
{
    static_assert(Assignable<T&,double>,"нельзя присваивать double");
    static_assert(Assignable<T&,string>,"нельзя присваивать string");
}
```

После нескольких значительных изменений эта идея стала основой определения концептов (§7.2).

6.4.2. Псевдонимы

Неожиданно часто бывает полезно ввести синоним типа или шаблона. Например, стандартный заголовок <cstddef> содержит определение псевдонима size_t, возможно, такой:

```
using size_t = unsigned int;
```

Фактический тип с именем size_t зависит от реализации, поэтому в другой реализации size_t может быть, например, unsigned long. Наличие псевдонима size_t позволяет программисту писать переносимый код.

Очень часто для параметризованного типа предоставляются псевдонимы для типов, связанных с его аргументами шаблона. Например:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};
```

Фактически каждый контейнер стандартной библиотеки предоставляет для имени типа значений своих элементов псевдоним value_type (глава 11, "Контейнеры"). Это позволяет писать код, который будет работать для любого контейнера, следующего этому соглашению. Например:

```
template<typename C>
using Value_type = typename C::value_type; // Тип элементов C

template<typename Container>
void algo(Container& c)
{
    Vector<Value_type<Container>> vec;     // Храним здесь результаты
    // ...
}
```

Механизм псевдонимов может использоваться для определения нового шаблона путем связывания некоторых или всех аргументов шаблона. Например:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string,Value>;
String_map<int> m;      // m представляет собой Map<string,int>
```

6.4.3. `if` времени компиляции

Рассмотрим запись операции, которая может использовать один из двух вариантов операции — `slow_and_safe(T)` или `simple_and_fast(T)`. Такие проблемы часто встречаются в коде, где важны обобщенность и производительность. Традиционное решение состоит в том, чтобы написать пару перегруженных функций и выбирать наиболее подходящую на основе свойства (§13.9.1), такого как, например, свойство стандартной библиотеки `is_pod`. При использовании иерархии классов базовый класс может обеспечить общую операцию `slow_and_safe`, а производный класс — перекрыть ее реализацией `simple_and_fast`.

В C++17 можно просто использовать `if` времени компиляции:

```
template<typename T>
void update(T& target)
{
    // ...
    if constexpr(is_pod<T>::value)
        simple_and_fast(target); // Для "простых старых данных"
    else
        slow_and_safe(target);
    // ...
}
```

`is_pod<T>` — это свойство типа (§13.9.1), которое говорит нам о том, может ли этот тип быть тривиально копирован.

Инстанцируется только выбранная ветвь `if constexpr`. Это решение предлагает оптимальную производительность и локальность оптимизации.

Что важно, `if constexpr` не является механизмом манипуляции текстом и не может нарушать обычные правила грамматики, типов и областей видимости. Например:

```
template<typename T>
void bad(T arg)
{
```

```
    if constexpr(Something<T>::value)
        try {                          // Синтаксическая ошибка

    g(arg);

    if constexpr(Something<T>::value)
        } catch(...) { /* ... */ }  // Синтаксическая ошибка
}
```

Разрешение таких манипуляций текстом может серьезно подорвать удобочитаемость кода и создать проблемы для инструментов, основанных на современных методах представления программ (таких, как "абстрактные синтаксические деревья").

6.5. Советы

[1] Используйте шаблоны для выражения алгоритмов, применимых для многих типов аргументов; §6.1; [CG:T.2].

[2] Используйте шаблоны для выражения контейнеров; §6.2; [CG:T.3].

[3] Используйте шаблоны для повышения уровня абстракции кода; §6.2; [CG:T.1].

[4] Шаблоны безопасны с точки зрения типов, но соответствующие проверки выполняются слишком поздно; §6.2.

[5] Позвольте конструкторам или шаблонам функций выводить аргументы типов шаблонов классов; §6.2.3.

[6] Используйте функциональные объекты в качестве аргументов алгоритмов; §6.3.2; [CG:T.40].

[7] Используйте лямбда-выражения, если вам требуются простые функциональные объекты в единственном месте; §6.3.2.

[8] Виртуальная функция-член не может быть шаблонной функцией-членом; §6.3.1.

[9] Используйте псевдонимы шаблонов для упрощения записи и сокрытия деталей реализации; §6.4.2.

[10] Чтобы использовать шаблон, убедитесь, что его определение (а не только объявление) находится в области видимости; §7.5.

[11] Шаблоны предоставляют возможность "неявной типизации"; §7.5.

[12] Отдельной компиляции шаблонов нет: включайте с помощью директивы `#include` определения шаблонов в каждую единицу трансляции, которая их использует.

7

Концепты и обобщенное программирование

Программировать следует начинать
с интересных алгоритмов.

— Алекс Степанов

7.1. Введение

Для чего нужны шаблоны? Другими словами, какие методы программирования эффективны при использовании шаблонов? Шаблоны предлагают следующее.

- Возможность передавать типы (а также значения и шаблоны) в качестве аргументов без потери информации. Это подразумевает отличные возможности для встраивания, что обеспечивает большие преимущества для современных компиляторов.

- Возможность соединения информации из разных контекстов во время инстанцирования, что также подразумевает большие возможности оптимизации.
- Возможность передавать константные значения в качестве аргументов, подразумевающая возможность вычислений во время компиляции.

Другими словами, шаблоны предоставляют мощный механизм для вычислений времени компиляции и манипуляции типами, что может привести к очень компактному и эффективному коду. Помните, что типы (классы) могут содержать как код (§6.3.2), так и значения (§6.2.2).

Первое и наиболее частое применение шаблонов — это поддержка *обобщенного программирования*, т.е. программирования, ориентированного на проектирование, реализацию и использование общих алгоритмов. Здесь "общий" означает, что алгоритм может быть разработан как принимающий широкое разнообразие типов, лишь бы они удовлетворяли требованиям алгоритма к его аргументам. Вместе с концептами шаблоны являются основной поддержкой обобщенного программирования в C++. Шаблоны обеспечивают параметрический полиморфизм (времени компиляции).

7.2. Концепты (C++20)

Рассмотрим функцию `sum()` из §6.3.1:

```
template<typename Seq, typename Num>
Num sum(Seq s, Num v)
{
    for (const auto& x : s)
        v+=x;
    return v;
}
```

Она может быть вызвана для любой структуры данных, которая поддерживает функции `begin()` и `end()`, так, чтобы работал цикл `for` для диапазона. Такие структуры включают, например, `vector`, `list` и `map` стандартной библиотеки. Кроме того, тип элемента структуры данных ограничен только его использованием: он должен быть типом, который мы можем сложить с аргументом `Value`. Примерами являются `int`, `double` или `Matrix` (при разумном определении матрицы `Matrix`). Можно сказать, что алгоритм `sum()` является обобщенным в двух измерениях: типа структуры данных, используемой для хранения элементов ("последовательность") и типа элементов.

Итак, `sum()` требует, чтобы ее первый аргумент шаблона представлял собой некоторую последовательность, а второй аргумент шаблона — некоторое число. Мы называем такие требования *концептами*.

Языковая поддержка концептов еще не вошла в стандарт ISO C++, но в настоящее время является технической спецификацией ISO [11]. В настоящее время уже имеются используемые реализации, поэтому я рискну рекомендовать концепты в этой книге — пусть даже детали могут измениться и может пройти несколько лет до того, как каждый сможет использовать концепты в своем производственном коде.

7.2.1. Применение концептов

Большинство аргументов шаблонов должны удовлетворять определенным требованиям для правильной компиляции шаблона и правильного функционирования сгенерированного кода. Таким образом, большинство шаблонов должны быть ограниченными шаблонами (§6.2.1). Ключевое слово `typename` является наименее ограничивающим, требующим только того, чтобы аргумент был типом. Обычно мы можем сделать большее. Рассмотрим функцию `sum()` еще раз:

```
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v)
{
    for (const auto& x : s)
        v+=x;
    return v;
}
```

Так код становится намного яснее. Как только мы определим, что означают понятия `Sequence` и `Number`, компилятор сможет отклонить неверные вызовы, просмотрев только интерфейс `sum()`, а не его реализацию. Это улучшит отчеты об ошибках.

Однако спецификация интерфейса `sum()` неполная: я "забыл" сказать, что мы должны иметь возможность суммировать элементы `Sequence` с `Number`. Мы можем это сделать:

```
template<Sequence Seq, Number Num>
    requires Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

`Value_type` у последовательности представляет собой тип элементов в последовательности. `Arithmetic<X,Y>` — это концепт, указывающий, что мы можем выполнять арифметические вычисления с типами `X` и `Y`. Это спасает нас от случайных попыток вычисления `sum()` для `vector<string>` или `vector<int*>`, принимая при этом аргументы `vector<int>` и `vector <complex<double>>`.

В этом примере нам понадобился только оператор +=, но для простоты и гибкости мы не должны слишком строго ограничивать наш шаблонный аргу-

мент. В частности, мы можем когда-нибудь захотеть выразить `sum()` в терминах + и =, а не +=, и тогда мы будем рады, что использовали более общий концепт (в данном случае — `Arithmetic`), а не узкое требование "наличия +=".

Частичные спецификации, как в первой версии `sum()` с использованием концептов, могут быть очень полезными. Если спецификация не будет полной, некоторые ошибки не будут найдены до момента инстанцирования. Тем не менее частичные спецификации могут помочь выразить намерения и необходимы для плавного постепенного развития, когда мы изначально не знаем все необходимые требования. При наличии зрелых библиотек концептов исходные спецификации будут близки к совершенству.

Неудивительно, что конструкция `requires Arithmetic<Value_type<Seq>,Num>` называется конструкцией `requirements` (конструкцией требований). Запись `template<Sequence Seq>` просто является сокращением для явного использования `requires Sequence<Seq>`. Если вам нравится многословность, можете воспользоваться следующей эквивалентной записью:

```
template<typename Seq, typename Num>
    requires Sequence<Seq> && Number<Num>
                           && Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

С другой стороны, можно также воспользоваться эквивалентностью двух записей, чтобы написать

```
template<Sequence Seq, Arithmetic<Value_type<Seq>> Num>
Num sum(Seq s, Num n);
```

В тех случаях, когда мы еще не можем использовать концепты, приходится ограничиваться соглашениями об именовании и комментариями, такими как

```
template<typename Sequence, typename Number>
    // requires Arithmetic<Value_type<Sequence>,Number>
Numer sum(Sequence s, Number n);
```

Какую бы запись мы ни выбрали, важно спроектировать шаблон с семантически значимыми ограничениями его аргументов (§7.2.4).

7.2.2. Перегрузка на основе концептов

Как только мы правильно определили шаблоны с их интерфейсами, мы можем выполнять перегрузку на основе их свойств, как делали это для функций. Рассмотрим немного упрощенную функцию стандартной библиотеки `advance()`, которая передвигает итератор (§12.3):

```
template<Forward_iterator Iter>
void advance(Iter p, int n) // Перемещение p на n элементов вперед
{
```

```
    while(n--)
        ++p;                    // Однонаправленный итератор имеет ++,
}                               // но не + или +=

template<Random_access_iterator Iter>
void advance(Iter p, int n) // Перемещение p на n элементов вперед
{
    p+=n;                       // Итератор с произвольным доступом
}                               // имеет операцию +=
```

Компилятор выберет шаблон с наиболее строгими требованиями, которым удовлетворяют аргументы. `list` поддерживает однонаправленные итераторы, но не итераторы с произвольным доступом, которые поддерживает `vector`, поэтому мы получаем

```
void user(vector<int>::iterator vip, list<string>::iterator lsp)
{
    advance(vip,10);    // Использует быструю версию advance()
    advance(lsp,10);    // Использует медленную версию advance()
}
```

Подобно другим перегрузкам, это механизм времени компиляции, не влекущий накладные расходы времени выполнения. Если компилятор не находит наилучший вариант, он сообщает об ошибке неоднозначности. Правила перегрузки на основе концептов намного проще, чем правила общей перегрузки (§1.3). Рассмотрим сначала единственный аргумент для нескольких альтернативных функций.

- Если аргумент не соответствует концепту, данная альтернатива не может быть выбрана.
- Если аргумент соответствует концепту только в одной альтернативе, то выбирается именно эта альтернатива.
- Если аргументы двух альтернатив одинаково хорошо соответствуют концепту, мы сталкиваемся с неоднозначностью.
- Если аргументы двух альтернатив соответствуют концепту и один из них более строгий, чем другой (соответствует всем тем же требованиям, что и другой, и еще некоторым), то выбирается именно эта альтернатива.

Для выбранной альтернативы должны выполняться следующие условия:

- соответствие всех аргументов; при этом
- соответствие всех аргументов как минимум столь же хорошее, как и для других альтернатив; при этом
- лучшее соответствие как минимум одного аргумента.

7.2.3. Корректный код

Вопрос о том, предлагает ли набор аргументов шаблона то, что шаблон требует от своих параметров, в конечном итоге сводится к тому, являются ли некоторые выражения корректными.

Используя выражение `requires`, мы можем проверить, корректен ли набор выражений. Например:

```
template<Forward_iterator Iter>
void advance(Iter p, int n)    // Перемещение p на n элементов вперед
{
    while(n--)
        ++p;                   // Однонаправленный итератор имеет ++,
}                              // но не + или +=

template<Forward_iterator Iter, int n>
requires requires(Iter p, int i)
{
    p[i]; p+i;                 // Iter имеет операции индексирования
}                              // и сложения

void advance(Iter p, int n)    // Перемещение p на n элементов вперед
{
    p+=n;                      // Итератор с произвольным доступом
}                              // имеет операцию +=
```

Нет, `requires requires` — не опечатка. Первый `requires` начинает конструкцию требований, а второй начинает `requires`-выражение

```
requires(Iter p, int i) { p[i]; p+i; }
```

`requires`-выражение представляет собой предикат, который возвращает `true`, если инструкции в нем являются корректным кодом, и `false` — в противном случае.

Я рассматриваю `requires`-выражения как ассемблерный код обобщенного программирования. Как и обычный ассемблерный код, `requires`-выражения чрезвычайно гибкие и не налагают никакой дисциплины программирования. В той или иной форме они находятся "на дне" более интересного обобщенного кода, так же как ассемблерный код находится "на дне" наиболее интересного обычного кода. Подобно ассемблеру, `requires`-выражения не должны быть видны в "обычном коде". Если вы видите в своем коде `requires requires`, вероятно, ваш код на слишком низком уровне.

Использование `requires requires` в `advance()` преднамеренно неэлегантное и хакерское. Обратите внимание, что я "забыл" указать операцию `+=` и типы результатов операций. Вы предупреждены! Предпочитайте именованные концепты, имя которых указывает их семантический смысл.

Отдавайте предпочтение правильно именованным понятиям с хорошо определенной семантикой (§7.2.4) и используйте `requires`-выражения в их определениях.

7.2.4. Определение концептов

В конечном итоге мы ожидаем, что сможем найти полезные концепты, такие как `Sequence` и `Arithmetic`, в библиотеках (включая стандартную библиотеку). Техническая спецификация [37] в настоящее время уже предлагает набор для ограничений алгоритмов стандартной библиотеки (§12.7). Однако простые концепты определить несложно.

Концепт — это предикат времени компиляции, указывающий, каким образом можно использовать один или несколько типов. Рассмотрим сначала один из простейших примеров:

```
template<typename T>
concept Equality_comparable =
    requires (T a, T b) {
        { a == b } -> bool; // Сравнение Ts с помощью ==
        { a != b } -> bool; // Сравнение Ts с помощью !=
    };
```

`Equality_comparable` — это концепт, который мы используем для обеспечения того, что значения можно сравнивать на равенство и неравенство. Мы просто говорим, что для двух значений данного типа они должны быть сравниваемы с использованием операторов `==` и `!=`, и результат этих операций должен быть преобразуем в `bool`. Например:

```
static_assert(Equality_comparable<int>); // Успешно

struct S { int a; };
// Сбой из-за отсутствия у структур == и != по умолчанию:
static_assert(Equality_comparable<S>);
```

Определение концепта `Equality_comparable` в точности эквивалентно описанию на естественном языке, и не более того. Значение `concept` всегда имеет тип `bool`.

Определение `Equality_comparable` для негомогенных сравнений практически столь же простое:

```
template<typename T, typename T2 = T>
concept Equality_comparable =
requires (T a, T2 b) {
    { a == b } -> bool; // Сравнение T и T2 с помощью ==
    { a != b } -> bool; // Сравнение T и T2 с помощью !=
    { b == a } -> bool; // Сравнение T2 и T с помощью ==
    { b != a } -> bool; // Сравнение T2 и T с помощью !=
};
```

Фрагмент `typename T2=T` гласит, что если мы не определяем второй аргумент шаблона, то `T2` будет тем же самым, что и `T`; `T` представляет собой *аргумент шаблона по умолчанию*.

Мы можем протестировать `Equality_comparable` следующим образом:

```
static_assert(Equality_comparable<int,double>); // Успешно
static_assert(Equality_comparable<int>);        // Успешно (T2 = int)
static_assert(Equality_comparable<int,string>); // Сбой
```

В качестве более сложного примера рассмотрим последовательность:

```
template<typename S>
concept Sequence = requires(S a) {
    typename Value_type<S>;    // S должно иметь тип значения.
    typename Iterator_type<S>; // S должно иметь тип итератора.

    // begin(a) должна возвращать итератор:
    { begin(a) } -> Iterator_type<S>;
    // end(a) должна возвращать итератор:
    { end(a) } -> Iterator_type<S>;

    requires Same_type<Value_type<S>,Value_type<Iterator_type<S>>>;
    requires Input_iterator<Iterator_type<S>>;
};
```

Чтобы тип `S` являлся `Sequence`, он должен предоставить `Value_type` (тип его элементов) и `Iterator_type` (тип его итераторов, см. §12.1). Он также должен гарантировать, что существуют функции `begin()` и `end()`, которые возвращают итераторы, так как это идиоматично для контейнеров стандартной библиотеки (§11.3). Наконец, `Iterator_type` в действительности должен быть `input_iterator` с элементами того же типа, что и элементы `S`.

Наиболее сложными понятиями для определения являются те, которые представляют фундаментальные языковые концепции. Следовательно, лучше использовать набор из установленной библиотеки (§12.7).

7.3. Обобщенное программирование

Разновидность *обобщенного программирования*, поддерживаемая C++, сосредоточивается вокруг идеи абстрагирования от конкретных, эффективных алгоритмов для получения обобщенных алгоритмов, которые могут быть объединены с различными представлениями данных для создания широкого спектра полезного программного обеспечения [39]. Абстракции, представляющие фундаментальные операции и структуры данных, называются *концептами*; они выглядят как требования, предъявляемые к параметрам шаблона.

7.3.1. Использование концептов

Хорошие, полезные концепты фундаментальны и открывают больше, чем планировалось. Примерами являются целое число и число с плавающей точкой (определенные даже в классическом C), последовательность, и более общие математические концепции, такие как поле и векторное пространство. Они представляют собой фундаментальные концепции области применения. Именно поэтому их называют "концептами". Идентификация и формализация концептов до степени, необходимой для эффективного обобщенного программирования, может оказаться проблемой.

Давайте в качестве примера рассмотрим концепт `Regular` (§12.7). Тип является регулярным, когда он ведет себя подобно `int` или `vector`. Объект регулярного типа

- может быть создан с помощью конструктора по умолчанию;
- может быть скопирован (с использованием обычной семантики копирования, дающей два независимых объекта, эквивалентных при сравнении) с использованием конструктора или присваивания;
- может сравниваться с использованием операторов == и !=;
- не страдает от технических проблем при применении чрезмерно умных трюков программирования.

Еще одним примером регулярного типа является `string`. Подобно `int`, `string` также является `StrictTotallyOrdered` (§12.7), т.е. две строки можно сравнивать с помощью операторов <, <=, > и >= с соответствующей семантикой.

Концепт — это не просто синтаксическое понятие, оно фундаментально семантическое. Например, не определяйте оператор + для деления; это не соответствует требованиям для любых разумных чисел. К сожалению, у нас пока что нет языковой поддержки для выражения семантики, поэтому, чтобы получить семантически значимые понятия, мы вынуждены полагаться на знания экспертов и здравый смысл. Не определяйте семантически бессмысленные понятия, такие как `Addable` или `Subtractable` (суммируемые и вычитаемые). Вместо этого опирайтесь на знания предметной области для определения понятий, которые соответствуют фундаментальным понятиям этой предметной области.

7.3.2. Абстракции с использованием шаблонов

Хорошие абстракции тщательно выращиваются из конкретных примеров. Не рекомендуется пытаться "абстрагироваться", стремясь предусмотреть все мыслимые потребности и методы; это направление к неэлегантному и разду-

тому коду. Вместо этого начните с одного — а лучше с нескольких — конкретных примеров реального использования и попытайтесь устранить несущественные детали. Рассмотрим следующий код:

```
double sum(const vector<int>& v)
{
    double res = 0;
    for(auto x : v)
        res += x;
    return res;
}
```

Очевидно, что это один из множества способов вычисления суммы последовательности чисел.

Рассмотрим, что же делает этот код менее обобщенным, чем он должен быть.

- Почему только `int`?
- Почему только `vector`?
- Почему накапливание суммы выполняется в `double`?
- Почему начинаем с `0`?
- Почему суммируем?

Отвечая на первые четыре вопроса превращением конкретных типов в аргументы шаблона, мы получаем простейшую форму алгоритма `accumulate` стандартной библиотеки:

```
template<typename Iter, typename Val>
Val accumulate(Iter first, Iter last, Val res)
{
    for (auto p = first; p!=last; ++p)
        res += *p;
    return res;
}
```

Теперь мы имеем следующее.

- Структура данных, которая должна быть обойдена, абстрагирована с помощью пары итераторов, представляющих последовательность (§12.1).
- Тип аккумулятора сделан параметром.
- Начальное значение теперь представляет собой входные данные; тип аккумулятора представляет собой тип этого начального значения.

Быстрое исследование (или еще лучше — измерение) показывает, что код, сгенерированный для вызовов с различными структурами данных, идентичен

коду, который вы получаете из исходного примера с ручным кодированием. Например:

```
void use(const vector<int>& vec, const list<double>& lst)
{
    // Суммирование в double:
    auto sum = accumulate(begin(vec),end(vec),0.0);
    auto sum2 = accumulate(begin(lst),end(lst),sum);
    //
}
```

Процесс обобщения конкретного фрагмента кода (или нескольких фрагментов) при сохранении производительности называется *подъемом* (lifting). Наилучший способ разработки шаблона зачастую представляет собой такую последовательность действий:

- сначала разрабатывается конкретная версия;
- затем выполняются отладка, тестирование и измерения;
- и наконец конкретные типы заменяются аргументами шаблона.

Обычно повторение `begin()` и `end()` утомляет, так что можно немного упростить пользовательский интерфейс:

```
// Range – нечто с функциями-членами begin() и end():
template<Range R, Number Val>
Val accumulate(R r, Val res = 0)
{
    for (auto p = begin(r); p!=end(r); ++p)
        res += *p;
    return res;
}
```

Для полной обобщенности мы можем абстрагировать и операцию +=; см. §14.3.

7.4. Вариативные шаблоны

Шаблон может быть определен как принимающий произвольное количество аргументов произвольных типов. Такой шаблон называется *вариативным шаблоном* (variadic template). Рассмотрим простую функцию для вывода значений любого типа, для которых определен оператор <<:

```
void user()
{
    // first: 1 2.2 hello:
    print("first: ", 1, 2.2, "hello\n"s);
    // second: 0.2 c yuck! 0 1 2:
    print("\nsecond: ", 0.2, 'c', "yuck!"s, 0, 1, 2, '\n');
}
```

Традиционно реализация вариативного шаблона заключалась в том, чтобы отделить первый аргумент от остальных, а затем рекурсивно вызвать вариативный шаблон для остальных аргументов:

```
void print()
{
    // Что делать при отсутствии аргументов? Ничего!
}

template<typename T, typename ... Tail>
void print(T head, Tail... tail)
{
    // Что делается с каждым аргументом, например
    cout << head << ' ';
    print(tail...);
}
```

`typename...` указывает, что `Tail` представляет собой последовательность типов. `Tail...` указывает, что `tail` является последовательностью значений типов, перечисленных в `Tail`. Параметр, объявленный с помощью троеточия `...`, называется *пакетом параметров*. Здесь `tail` — это аргумент функции, который представляет собой пакет параметров, элементы которого имеют типы, найденные в аргументе шаблона, который представляет собой пакет параметров `Tail`. Таким образом, `print()` может принимать любое количество аргументов любых типов.

Вызов `print()` разделяет аргументы на голову (первый элемент) и хвост (остальные элементы). Выполняется вывод головного элемента, после чего `print()` вызывается для хвоста. В конечном итоге `tail` становится пустым, поэтому нам нужна версия `print()` без аргументов, способная справиться с этой ситуацией и завершить рекурсию. Можно обойтись и без `print()` без аргументов, используя для его исключения `if` времени компиляции:

```
template<typename T, typename ... Tail>
void print(T head, Tail... tail)
{
    cout << head << ' ';
    if constexpr(sizeof...(tail)> 0)
        print(tail...);
}
```

Я использовал `if` времени компиляции (§6.4.3), а не `if` времени выполнения, чтобы избежать генерации последней, никогда не вызываемой функции `print()` без аргументов.

Сила вариативных шаблонов заключается в том, что они могут принимать любые аргументы, которые вы захотите им передать. Слабые же их стороны включают следующее.

- Рекурсивная реализация может оказаться сложной задачей.
- Рекурсивные реализации могут быть неожиданно дорогими во время компиляции.
- Проверка типа интерфейса, возможно, представляет собой сложную шаблонную программу.

Из-за гибкости вариативные шаблоны широко используются в стандартной библиотеке, иногда даже чрезмерно широко.

7.4.1. Выражения свертки

Чтобы упростить реализацию простых вариативных шаблонов, C++17 предлагает ограниченную форму итерации по элементам пакета параметров. Например:

```
template<Number... T>
int sum(T... v)
{
    return (v + ... + 0); // Прибавление всех элементов v к 0
}
```

Здесь `sum()` может принимать любое количество элементов любого типа. В предположении, что `sum()` действительно суммирует свои аргументы, мы получаем

```
int x = sum(1, 2, 3, 4, 5); // x равно 15
// y равно 114 (2.4 обрезается до 2, а значение символа 'a':
int y = sum('a', 2.4, x); равно 97)
```

Тело `sum` использует выражение свертки:

```
return (v + ... + 0); // Прибавление всех элементов v к 0
```

Здесь `(v+...+0)` означает прибавление всех элементов `v` к начальному значению `0`. Первым прибавляется крайний справа аргумент (имеющий наивысший индекс): `(v[0]+(v[1]+(v[2]+(v[3]+(v[4]+0)))))`. То есть суммирование начинается справа, где находится `0`. Такое действие называется *правой сверткой* (right fold). В качестве альтернативы мы можем использовать *левую свертку* (left fold):

```
template<Number... T>
int sum2(T... v)
{
    return (0 + ... + v); // Прибавление всех элементов v к 0
}
```

Теперь первым прибавляется аргумент, крайний слева (с наименьшим индексом): `((((0+v[0])+v[1])+v[2])+v[3])+v[4])`. То есть суммирование начинается слева, где находится `0`.

Свертка (fold) представляет собой очень мощную абстракцию, очевидным образом связанную с алгоритмом `accumulate()` стандартной библиотеки, со множеством имен в разных языках и сообществах. В C++ выражения свертки в настоящее время ограничены упрощением реализации вариативных шаблонов. Свертка не обязана выполнять только числовые вычисления. Рассмотрим знаменитый пример:

```
template<typename ...T>
void print(T&&... args)
{
    (std::cout << ... << args) << '\n'; // Вывод всех аргументов
}
// (((((std::cout<<"Hello!"s)<<' ')<<"World ")<<2017)<<'\n');
print("Hello!"s,' ',"World ",2017);
```

Многие сценарии просто включают набор значений, которые могут быть преобразованы в общий тип. В таких случаях простое копирование аргументов в вектор или желаемый тип часто упрощает дальнейшее использование:

```
template<typename Res, typename... Ts>
vector<Res> to_vector(Ts&&... ts)
{
    vector<Res> res;
    (res.push_back(ts) ...); // Начальное значение не требуется
    return res;
}
```

Мы можем использовать `to_vector` следующим образом:

```
auto x = to_vector<double>(1,2,4.5,'a');

template<typename ... Ts>
int fct(Ts&&... ts)
{
    // args[i] представляет собой i-й аргумент:
    auto args = to_vector<string>(ts...);
    // ... Использование args ...
}

int y = fct("foo", "bar", s);
```

7.4.2. Передача аргументов

Передача аргументов через интерфейс неизменными — важное использование вариативных шаблонов. Рассмотрим понятие сетевого входного канала, для которого фактический метод перемещения значений является парамет-

ром. Различные транспортные механизмы имеют разные наборы параметров конструктора:

```
template<typename Transport>
    requires concepts::InputTransport<Transport>
class InputChannel
{
public:
    // ...
    InputChannel(TransportArgs&&... transportArgs)
    : _transport(std::forward<TransportArgs>(transportArgs)...)
    {}
    // ...
    Transport _transport;
};
```

Функция стандартной библиотеки `forward()` (§13.2.2) используется для передачи аргументов неизменными из конструктора `InputChannel` в конструктор `Transport`.

Дело в том, что автор `InputChannel` может создать объект типа `Transport` без необходимости знать, какие аргументы необходимы для построения конкретного транспорта. Разработчику `InputChannel` нужно знать только общий пользовательский интерфейс для всех объектов `Transport`.

Передача очень распространена в фундаментальных библиотеках, где необходимы общность и низкие накладные расходы времени выполнения, и распространены очень общие интерфейсы.

7.5. Модель компиляции шаблонов

В предположении наличия концептов (§7.2), аргументы шаблона проверяются на соответствие его концептам. Об обнаруженных ошибках компилятор сообщает программисту, который должен решить указанные проблемы. Проверка того, что в данный момент не может быть проверено, например аргументы для неограниченных шаблонных параметров, переносится на то время, когда будет сгенерирован код для шаблона и его набора аргументов — "во время инстанцирования шаблона". В случае кода, разработанного до появления концептов, именно в этот момент и происходит проверка всех типов. При использовании концептов эта проверка выполняется только после того, как успешно завершена проверка концептов.

Неприятный побочный эффект (поздней) проверки типов времени инстанцирования заключается в том, что ошибка типа может быть обнаружена слишком поздно и привести к поразительно плохим сообщениям об ошибках, поскольку компилятор обнаруживает проблему только после объединения информации из нескольких мест программы.

Проверка типа времени инстанцирования, предоставляемая для шаблонов, проверяет использование аргументов в определении шаблона. Это обеспечивает *неявную типизацию* времени компиляции, часто именуемую *утиной типизацией* ("Если нечто выглядит, как утка, плавает, как утка, и крякает, как утка, то это, вероятно, и есть утка"). Используя более техническую терминологию, мы работаем со значениями, а наличие и смысл операции зависят исключительно от значений операндов. Это отличается от альтернативного представления о том, что объекты имеют типы, которые определяют наличие и смысл операций. Значения "живут" в объектах. Это способ работы объектов (например, переменных) в C++, и только значения, соответствующие требованиям объекта, могут быть в него помещены. То, что делается во время компиляции с использованием шаблонов, в основном не касается объектов и имеет дело только со значениями. Исключение составляют локальные переменные в `constexpr`-функции (§1.6), которые используются внутри компилятора как объекты.

Чтобы использовать неограниченный шаблон, его определение (а не только его объявление) должно быть в области видимости в точке использования. Например, стандартный заголовочный файл `<vector>` содержит определение `vector`. На практике это означает, что определения шаблонов обычно находятся в заголовочных файлах, а не в `.cpp`-файлах. Все изменяется, когда мы начинаем использовать модули (§3.3). При использовании модулей исходный текст организован одинаково как для обычных функций, так и для шаблонных. В обоих случаях определения будут защищены от проблем, связанных с включением текста.

7.6. Советы

[1] Шаблоны обеспечивают общий механизм программирования времени компиляции; §7.1.

[2] При проектировании шаблона внимательно рассмотрите концепты (требования), предъявляемые к аргументам шаблона; §7.3.2.

[3] При проектировании шаблона в качестве начальной реализации используйте конкретную версию, прошедшую отладку и измерения; §7.3.2.

[4] Используйте концепты как инструмент проектирования; §7.2.1.

[5] Указывайте концепты для всех аргументов шаблонов; §7.2; [CG:T.10].

[6] Когда это возможно, используйте стандартные концепты (например, концепты диапазонов); §7.2.4; [CG:T.11].

[7] Используйте лямбда-выражения, если вам нужен простой функциональный объект для использования в единственном месте; §6.3.2.

[8] Раздельной компиляции шаблонов нет: вы должны включать с помощью директивы `#include` определения шаблонов в каждую единицу трансляции, которая их использует.

[9] Используйте шаблоны для выражения контейнеров и диапазонов; §7.3.2; [CG:T.3].

[10] Избегайте "концептов" без значимой семантики; §7.2; [CG:T.20].

[11] Требуйте в концепте полный набор операций; §7.2; [CG:T.21].

[12] Используйте вариативные шаблоны, когда вам нужна функция, получающая переменное количество аргументов различных типов; §7.4.

[13] Не используйте вариативные шаблоны для гомогенных списков аргументов (предпочитайте использовать для этого списки инициализации); §7.4.

[14] При использовании шаблона убедитесь, что его определение (а не только объявление) находится в области видимости; §7.5.

[15] Шаблоны обеспечивают неявную типизацию времени компиляции; §7.5.

8

Обзор библиотеки

Зачем терять время на обучение,
когда невежество мгновенно?

— Гоббс

- **Введение**
- **Компоненты стандартной библиотеки**
- **Заголовочные файлы и пространство имен стандартной библиотеки**
- **Советы**

8.1. Введение

Никакая серьезная программа не может быть написана просто на языке программирования. Во-первых, имеется разработанный набор библиотек, которые составляют основу для дальнейшей работы. Писать на голом языке программу в большинстве случаев очень утомительно, тем более что практически любая задача может быть упрощена благодаря использованию хороших библиотек.

Продолжая материал, изложенный в главах 1–7, в главах 9–15 представлен краткий обзор основных возможностей стандартной библиотеки. Я очень бегло представляю полезные типы стандартной библиотеки, такие как `string`, `ostream`, `variant`, `vector`, `map`, `path`, `unique_ptr`, `thread`, `regex` и `complex`, а также наиболее распространенные способы их применения.

Как и в главах 1–7, читателю настоятельно рекомендуется не переживать из-за неполного понимания всех деталей. Цель главы — дать общее представление о наиболее полезных библиотечных средствах.

Спецификация стандартной библиотеки составляет более двух третей стандарта ISO C++. Исследуйте ее возможности и предпочитайте их, а не "самопальные" альтернативы. При проектировании библиотеки затрачено немало умственных усилий профессионалов, еще больше их вложено в ее реализацию; многие усилия продолжают направляться на ее обслуживание и расширение.

Возможности стандартной библиотеки, описанные в этой книге, являются частью каждой полной реализации C++. В дополнение к компонентам стандартной библиотеки большинство реализаций предлагают варианты графического интерфейса пользователя (graphical user interface — GUI), веб-интерфейсы, интерфейсы баз данных и т.д. Аналогично большинство сред разработки приложений предоставляют "фундаментальные библиотеки" для корпоративных или промышленных "стандартных" сред разработки и/или выполнения. Здесь я не описываю такие системы и библиотеки. Моя цель состоит в том, чтобы обеспечить самодостаточное описание C++, определенное стандартом, и сохранить переносимость примеров. Естественно, программисту предлагается изучить более широкие возможности, доступные в большинстве конкретных систем.

8.2. Компоненты стандартной библиотеки

Возможности, предоставляемые стандартной библиотекой, можно классифицировать следующим образом.

- Поддержка языка программирования времени выполнения (например, выделение памяти или информация о типах времени выполнения).
- Стандартная библиотека C (с небольшими изменениями для минимизации нарушений системы типов).
- Строки (с поддержкой международных наборов символов, локализацией и представлениями подстрок, доступными только для чтения); см. §9.2.
- Поддержка регулярных выражений; см. §9.4.
- Потоки ввода-вывода представляют собой расширяемый каркас для ввода и вывода, к которому пользователи могут добавлять собственные типы, потоки, стратегии буферизации, локали и наборы символов (глава 10, "Ввод и вывод"). Существует также библиотека для переносимой работы с файловыми системами (§10.10).
- Каркас контейнеров (таких, как `vector` и `map`) и алгоритмов (таких, как `find()`, `sort()` и `merge()`); см. главы 11, "Контейнеры", и 12, "Алгоритмы". Этот каркас, обычно именуемый STL [39], расширяем, поэтому пользователи могут добавлять собственные контейнеры и алгоритмы.
- Поддержка числовых вычислений (таких, как стандартные математические функции, комплексные числа, векторы с арифметическими операциями и генераторы случайных чисел); см. §4.2.1 и главу 14, "Числовые вычисления".

- Поддержка параллельного программирования, включая потоки выполнения `thread` и блокировки; см. главу 15, "Параллельные вычисления". Поддержка параллелизма является основанием для расширений, так что пользователи могут добавлять поддержку новых моделей параллелизма в виде библиотек.
- Параллельные версии большинства алгоритмов STL и некоторых числовых алгоритмов (например, `sort()` или `reduce()`); см. §12.9 и §14.3.1.
- Утилиты для поддержки шаблонного метапрограммирования (например, свойства типов; §13.9), обобщенного программирования в стиле STL (например, `pair`; §13.4.3), общее программирование (например, `variant` и `optional`; §13.5.1, §13.5.2) и `clock` (§13.7).
- Поддержка эффективного и безопасного управления ресурсами, а также интерфейс для необязательных сборщиков мусора (§5.3).
- "Интеллектуальные указатели" для управления ресурсами (например, `unique_ptr` и `shared_ptr`; §13.2.1).
- Контейнеры специального назначения, такие как `array` (§13.4.1), `bitset` (§13.4.2) и `tuple` (§13.4.3).
- Суффиксы для популярных единиц измерения, например `ms` для миллисекунд и `i` для мнимых чисел (§5.4.4).

Основными критериями включения класса в библиотеку были следующие:

- он может быть полезен почти каждому программисту на C ++ (как новичкам, так и экспертам);
- он может быть предоставлен в общем виде, который не добавляет значительные накладные расходы по сравнению с более простой версией той же функциональной возможности;
- простое использование должно быть легким в освоении (относительно сложности, присущей задаче).

По сути, стандартная библиотека C++ предоставляет наиболее распространенные фундаментальные структуры данных вместе с фундаментальными алгоритмами, используемыми с ними.

8.3. Заголовочные файлы и пространство имен стандартной библиотеки

Каждое средство стандартной библиотеки предоставляется через некоторый стандартный заголовочный файл. Например:

```
#include<string>
#include<list>
```

Эти строки делают доступными стандартные классы `string` и `list`.

Стандартная библиотека определена в пространстве имен (§3.4), именуемом `std`. Чтобы использовать средства стандартной библиотеки, используется префикс `std::`.

```
std::string sheep {"Четыре ноги хорошо, две - плохо!"};
std::list<std::string> slogans {"Война - это мир",
                                "Свобода - это рабство",
                                "Незнание - сила"};
```

Для простоты я редко использую в примерах префикс `std::` явно. Я также не всегда явно включаю нужные заголовочные файлы. Чтобы скомпилировать и запустить фрагменты программ из этой книги, вы должны включить соответствующие заголовочные файлы и сделать доступными имена, которые они объявляют. Например:

```
#include<string>      // Делаем доступными стандартные строки
using namespace std; // Делаем имена из std без префикса
// OK: здесь string представляет собой std::string
string s {"C++ - язык программирования общего назначения"};
```

Как правило, внесение всех имен из конкретного пространства имен в глобальное — признак плохого вкуса. Однако в этой книге я использую только стандартную библиотеку, и программисты должны знать, что она предлагает.

Ниже приведен список заголовочных файлов стандартной библиотеки; все их объявления находятся в пространстве имен `std`.

Избранные заголовочные файлы стандартной библиотеки		
`<algorithm>`	`copy()`, `find()`, `sort()`	Глава 12
`<array>`	`array`	§13.4.1
`<chrono>`	`duration`, `time_point`	§13.7
`<cmath>`	`sqrt()`, `pow()`	§14.2
`<complex>`	`complex`, `sqrt()`, `pow()`	§14.4
`<filesystem>`	`path`	§10.10
`<forward_list>`	`forward_list`	§11.6
`<fstream>`	`fstream`, `ifstream`, `ofstream`	§10.7
`<future>`	`future`, `promise`	§15.7
`<ios>`	`hex`, `dec`, `scientific`, `fixed`, `defaultfloat`	§10.6
`<iostream>`	`istream`, `ostream`, `cin`, `cout`	Глава 10

Окончание табл.

Избранные заголовочные файлы стандартной библиотеки		
<map>	map, multimap	§11.5
<memory>	unique_ptr, shared_ptr, allocator	§13.2.1
<random>	default_random_engine, normal_distribution	§14.5
<regex>	regex, smatch	§9.4
<string>	string, basic_string	§9.2
<set>	set, multiset	§11.6
<sstream>	istringstream, ostringstream	§10.8
<stdexcept>	length_error, out_of_range, runtime_error	§3.5.1
<thread>	thread	§15.2
<unordered_map>	unordered_map, unordered_multimap	§11.5
<utility>	move(), swap(), pair	Глава 13
<variant>	variant	§13.5.1
<vector>	vector	§11.2

Этот список далеко неполон.

Предоставляются также заголовочные файлы из стандартной библиотеки C, такие как <stdlib.h>. Для каждого такого заголовка есть также версия с именем с префиксом c, и удаленным .h. Эта версия, такая как <cstdlib>, помещает свои объявления в пространство имен std.

8.4. Советы

[1] Не изобретайте колесо — пользуйтесь библиотеками; §8.1; [CG:SL.1.]

[2] Если у вас есть возможность выбора, предпочитайте стандартную библиотеку другим библиотекам; §8.1; [CG:SL.2].

[3] Не думайте, что стандартная библиотека идеально подходит для любой задачи; §8.1.

[4] Не забывайте включать заголовочные файлы для используемых вами библиотечных средств; §8.3.

[5] Помните, что средства стандартной библиотеки находятся в пространстве имен std; §8.3; [CG:SL.3].

9

Строки и регулярные выражения

Предпочитайте стандартное отличному.

— Странк и Уайт[1]

9.1. Введение

Работа с текстом является основной частью большинства программ. Стандартная библиотека C++ предлагает тип `string`, который спасает большинство пользователей от применения С-образных манипуляций массивами символов с помощью указателей. Тип `string_view` позволяет работать с последовательностями символов, которые могут храниться в ином месте (например, в `std::string` или `char[]`). Кроме того, предоставляется возможность сопоставления текста регулярным выражениям, позволяющая искать соответствующие образцы в тексте. Регулярные выражения представлены в форме, подобной той, которая распространена в большинстве современных языков. И `string`, и `regex` могут использовать различные типы символов (например, Unicode).

[1] Цитата из книги Уильяма Странка (мл.) (W. Strunk Jr.) *Элементы стиля* (1920) с дополнениями Элвина Уайта (E. White) (1959). — *Примеч. пер.*

9.2. Строки

Стандартная библиотека в дополнение к строковым литералам (§1.2.1) предоставляет тип `string`, отвечающий концепту `Regular` (§7.2, §12.7), для владения и управления последовательностями символов различных типов. Тип `string` предоставляет множество полезных строковых операций, таких как конкатенация. Например:

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}
auto addr = compose("dmr","bell-labs.com");
```

Здесь `addr` инициализируется последовательностью символов `dmr@bell-labs.com`. "Сложение" строк типа `string` означает конкатенацию. Вы можете "складывать" со строкой `string` другую строку `string`, строковый литерал, строку в стиле C или символ. Стандартная строка `string` имеет перемещающий конструктор, так что возврат даже длинных строк по значению выполняется эффективно (§5.2.2).

Во многих приложениях наиболее распространенной формой конкатенации является добавление чего-то к концу строки. Это действие непосредственно поддерживается операцией +=. Например:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n'; // Добавление символа новой строки
    s2 += '\n';     // Добавление символа новой строки
}
```

Эти два способа добавления в конец строки семантически эквивалентны, но я предпочитаю последний, потому что он более ясно указывает, что он делает, более краток и, возможно, более эффективен.

Класс `string` — изменяемый. В дополнение к = и += поддерживаются также операции индекса (с использованием `[]`) и получения подстроки. Например:

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10); // s = "Stroustrup"
    name.replace(0,5,"nicholas"); // name == "nicholas Stroustrup"
    name[0] = toupper(name[0]);   // name == "Nicholas Stroustrup"
}
```

Операция `substr()` возвращает строку, которая является копией подстроки, указанной аргументами. Первый аргумент — это индекс в строке (позиция), а второй — длина требуемой подстроки. Поскольку индексация начинается с `0`, `s` получает значение `Stroustrup`.

Операция `replace()` заменяет подстроку значением. В этом случае подстрока, начинающаяся с `0` длиной `5`, представляет собой `Niels`; она заменяется этой операцией на `nicholas`. Наконец, я заменяю начальный символ его эквивалентом в верхнем регистре. Таким образом, окончательное значение `name` — `Nicholas Stroustrup`. Обратите внимание, что заменяющая подстрока не обязана быть того же размера, что и заменяемая.

Среди множества полезных операций `string` — присваивание (=), индексирование (`[]` или `at()`, как и для класса `vector`; §11.2.2), сравнение (== и !=) и лексикографическое упорядочение (<, <=, > и >=), итерирование (с использованием итераторов, как для класса `vector`; §12.2), ввод (§10.3) и использование времени компиляции потока (§10.8).

Естественно, строки `string` могут сравниваться одна с другой, со строками в стиле C (§1.7.1) и со строковыми литералами. Например:

```
string incantation;
void respond(const string& answer)
{
    if (answer == incantation) {
        // Выполнение магических действий[2]
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}
```

Если вам нужна строка в стиле C (массив `char` с завершающим нулевым символом), `string` предоставляет доступ только для чтения к содержащимся в нем символам. Например:

```
void print(const string& s)
{
    // s.c_str() возвращает указатель на символы s:
    printf("Для любителей printf: %s\n",s.c_str());
    cout << "Для любителей потоков: " << s << '\n';
}
```

Строковый литерал по определению представляет собой `const char*`. Чтобы получить литерал типа `std::string`, используйте суффикс `s`. Например:

[2] Incantation — в переводе "заклинание". — *Примеч. пер.*

```
auto s = "Cat"s; // std::string
auto p = "Dog";  // Строка в стиле C: const char*
```

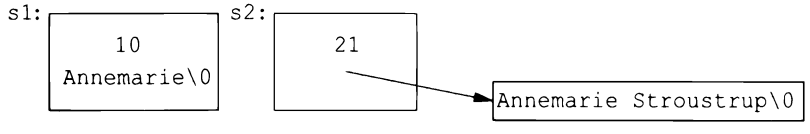
Для использования суффикса `s` вам нужно использовать пространство имен `std::literals::string_literals` (§5.4.4).

9.2.1. Реализация `string`

Реализация строкового класса является популярным и полезным упражнением. Однако для общего использования наши тщательно продуманные первые попытки редко соответствуют стандартной строке `string` в удобстве или производительности. В наши дни строка обычно реализуется с использованием *оптимизации коротких строк*, т.е. короткие строковые значения сохраняются в самом объекте `string`, и только более длинные строки размещаются в свободной памяти. Рассмотрим следующий пример:

```
string s1 {"Annemarie"};            // Короткая строка
string s2 {"Annemarie Stroustrup"}; // Длинная строка
```

Схема размещения в памяти может выглядеть следующим образом:



Когда значение строки изменяется от короткой до длинной строки (и наоборот), ее представление настраивается соответствующим образом. Сколько символов может содержать "короткая" строка? Это зависит от конкретной реализации, но "около 14 символов" — это неплохое предположение.

Фактическая производительность `string` может зависеть от среды времени выполнения. В частности, в многопоточных реализациях распределение памяти может быть относительно дорогостоящим. Кроме того, когда используется большое количество строк различной длины, может произойти фрагментация памяти. Это основные причины, по которым оптимизация коротких строк стала вездесущей.

Для работы с разными наборами символов `string` на самом деле представляет собой псевдоним общего шаблона `basic_string` с символьным типом `char`:

```
template<typename Char>
class basic_string {
    // ... строка из Char ...
};

using string = basic_string<char>
```

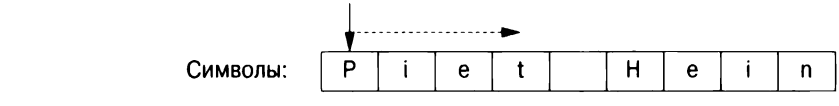
Пользователь может определить строку для произвольного типа символов. Например, в предположении, что существует тип японских символов `Jchar`, можно написать

```
using Jstring = basic_string<Jchar>;
```

После этого со строками японских символов `Jstring` можно выполнять все те же операции, что и с обычными строками.

9.3. Представления строк

Наиболее частое использование последовательности символов — это передача ее в некоторую функцию для чтения. Это может быть сделано путем передачи строки `string` по значению, по ссылке или с помощью строки в стиле C. Во многих системах существуют дополнительные альтернативы, такие как строковые типы, не предлагаемые стандартом. Во всех этих случаях при попытке передачи подстроки возникают дополнительные сложности. Чтобы решить эту проблему, стандартная библиотека предлагает класс `string_view`; который, по сути, содержит пару (указатель, длина), описывающую последовательность символов.



`string_view` предоставляет доступ к непрерывной последовательности символов. Символы могут быть сохранены многими возможными способами, в том числе в объекте `string` и в строке в стиле C. `string_view` похож на указатель или ссылку, не владеющую символами, на которые указывает. В этом он напоминает пару итераторов STL (§12.3).

Рассмотрим простую функцию конкатенации двух строк:

```
string cat(string_view sv1, string_view sv2)
{
    string res(sv1.length()+sv2.length());
    char* p = &res[0];
    for (char c : sv1)              // Первый способ копирования
        *p++ = c;
    copy(sv2.begin(),sv2.end(),p); // Второй способ
    return res;
}
```

Эта функция может быть использована следующим образом:

```
string king = "Harold";
auto s1 = cat(king,"William");          // string и const char*
auto s2 = cat(king,king);               // string и string
auto s3 = cat("Edward","Stephen"sv);    // const char* и string_view
auto s4 = cat("Canute"sv,king);
auto s5 = cat({&king[0],2},"Henry"sv);  // HaHenry
auto s6 = cat({&king[0],2},{&king[2],4});// Harold
```

Данная функция `cat()` имеет три преимущества перед функцией `compose()`, которая получает аргументы `const string&` (§9.2):

- она может применяться для последовательностей символов, управляемых разными способами;
- для строк в стиле C не создаются временные объекты типа `string`;
- в функцию можно легко передавать подстроки.

Обратите внимание на суффикс `sv` ("string view"). Чтобы его применять, нужно использовать соответствующее пространство имен:

```
using namespace std::literals::string_view_literals; // §5.4.4
```

Зачем об этом беспокоиться? Дело в том, что, когда мы передаем `"Edward"`, нам нужно построить `string_view` из `const char*`, что требует подсчета символов. Для `"Stephen"sv` эта длина вычисляется во время компиляции.

При возврате `string_view` помните, что этот тип очень похож на указатель и должен на что-то указывать:

```
string_view bad()
{
    string s = "Once upon a time";
    // Плохо: возврат указателя на локальную переменную:
    return {&s[5],4};
}
```

Мы возвращаем указатель на символы строки `string`, которая будет уничтожена до того, как мы сможем ее использовать.

Одно существенное ограничение `string_view` заключается в том, что это представление предназначено только для чтения символов. Например, вы не можете использовать `string_view` для передачи символов функции, которая изменяет свой аргумент на нижний регистр. Для этого вы можете воспользоваться `gsl::span` или `gsl::string_span` (§13.3).

Поведение при обращении к символам за рамками диапазона `string_view` не определено. Если вы хотите гарантированную проверку выхода за диапазон, используйте функцию-член `at()`, которая генерирует исключение `out_of_range` при попытке доступа за пределами диапазона, используйте `gsl::string_span` (§13.3) или "просто будьте внимательны".

9.4. Регулярные выражения

Регулярные выражения являются мощным инструментом для обработки текста. Они обеспечивают способ простого и подробного описания шаблонов в тексте (например, почтовый индекс США, такой как TX 77845, или дату в стиле ISO, например 2009-06-07) и эффективного поиска таких шаблонов. В заголовочном файле <regex> стандартная библиотека обеспечивает поддержку регулярных выражений в виде класса std::regex и функций поддержки. Чтобы попробовать библиотеку regex "на зуб", давайте определим и выведем шаблон регулярного выражения[3]:

```
// Шаблон почтового индекса США: XXddddd-dddd и его варианты:
regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};
```

Программисты практически на любом языке, которые использовали регулярные выражения, сочтут строку \w{2}\s*\d{5}(-\d{4})? знакомой. Она задает шаблон, начинающийся с двух букв \w{2}, за которыми следует необязательный пробел \s*, за которым следуют пять цифр \d{5} с необязательным последующим тире и четырьмя цифрами -\d{4}. Если вы не знакомы с регулярными выражениями, то сейчас самое время познакомиться с ними, например, в книгах [55], [33] и [22].

Чтобы записать шаблон, я использую необработанный строковый литерал, начинающийся с R"(и заканчивающийся)". Он позволяет использовать обратную косую черту и кавычки в строке непосредственно, без специальных последовательностей символов. Необработанные строки особенно хорошо подходят для регулярных выражений, потому что они, как правило, содержат много обратных косых черт. Если бы я использовал обычную строку, то определение шаблона было бы следующим:

```
regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"};
```

В заголовочном файле <regex> стандартная библиотека предоставляет поддержку регулярных выражений:

- regex_match(): соответствует ли регулярное выражение строке (известного размера) (§9.4.2);
- regex_search(): поиск строки, которая соответствует регулярному выражению, в (произвольно длинном) потоке данных (§9.4.1).

---

[3] В данном случае следует отличать шаблон регулярного выражения (pattern) от шаблона C++ (template). Поскольку из контекста понятно, о чем именно идет речь в том или ином случае; уточнения, какой именно шаблон имеется в виду, в основном опущены. — *Примеч. пер.*

- `regex_replace()`: поиск строки, которая соответствует регулярному выражению, в (произвольно длинном) потоке данных и ее замена;
- `regex_iterator`: итерация по всем соответствиям шаблону (§9.4.3);
- `regex_token_iterator`: итерация по несоответствиям.

9.4.1. Поиск

Простейший способ применения шаблона — поиск соответствия ему в потоке данных:

```
int lineno = 0;
for(string line; getline(cin,line); )     // Чтение в буфер line
{
    ++lineno;
    smatch matches;                       // Соответствующие строки
    if (regex_search(line ,matches,pat)) // Поиск pat в line
        cout << lineno << ": " << matches[0] << '\n';
}
```

`regex_search(line,matches,pat)` выполняет поиск в `line` всего, что соответствует регулярному выражению, хранящемуся в `pat`, и если находятся какие-либо совпадения, то они сохраняются в `matches`. Если совпадений не найдено, `regex_search(line,matches,pat)` возвращает `false`. Переменная `matches` имеет тип `smatch`. Буква `s` означает `string`, а `smatch` представляет собой `vector` найденных соответствий типа `string`. Первый элемент, здесь — `matches[0]`, представляет собой полное совпадение. Результат `regex_search()` представляет собой набор совпадений, обычно представленных в виде `smatch`:

```
void use()
{
    ifstream in("file.txt");  // Входной файл
    if (!in)                  // Проверка открытия файла
        cerr << "no file\n";

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; // Шаблон индекса

    int lineno = 0;
    for(string line; getline(in,line); )
    {
        ++lineno;
        smatch matches;                      // Найденные строки
        if (regex_search(line, matches, pat))
        {
            // Полное соответствие:
            cout << lineno << ": " << matches[0] << '\n';
            // Если есть подшаблон и для него имеется соответствие:
            if (1 < matches.size() && matches[1].matched)
```

```
                cout << "\t: " << matches[1] << '\n';    // Подшаблон
        }
    }
}
```

Эта функция читает файл в поисках почтовых индексов США, таких как `TX77845` и `DC 20500-0001`. Тип `smatch` — контейнер с результатами поиска. Здесь `matches[0]` — это весь шаблон, а `matches[1]` — необязательный четырехзначный подшаблон.

Символ новой строки `\n` может быть частью шаблона, поэтому можно искать многострочные шаблоны. Очевидно, если мы хотим это сделать, нам не следует читать строки по одной.

Синтаксис и семантика регулярных выражений разработаны таким образом, что регулярные выражения могут быть скомпилированы в конечные автоматы для эффективного выполнения [14]. Тип `regex` выполняет эту компиляцию во время выполнения.

9.4.2. Запись регулярных выражений

Библиотека регулярных выражений `regex` может распознавать несколько вариантов обозначений для регулярных выражений. Здесь я использую стандартную запись, вариант стандарта ECMA, используемый в ECMAScript (более известен как JavaScript).

Синтаксис регулярных выражений основан на символах со специальным значением.

Специальные символы регулярных выражений			
`.`	Любой единственный символ	`\`	Следующий символ имеет особое значение
`[`	Начало класса символов	`*`	Нуль или большее количество (суффикс)
`]`	Конец класса символов	`+`	Один или большее количество (суффикс)
`{`	Начало счетчика	`?`	Необязателен (нуль или один) (суффикс)
`}`	Конец счетчика	`|`	Альтернатива (или)
`(`	Начало группировки	`^`	Начало строки; отрицание
`)`	Конец группировки	`$`	Конец строки

Например, мы можем указать строку, начинающуюся с нуля или более символов `A`, за которыми следует одна или несколько букв `B`, а затем — необязательный символ `C`:

```
^A*B+C?$
```

Вот примеры строк, соответствующих этому регулярному выражению:

```
AAAAAAAAAAAAABBBBBBBBBC
BC
B
```

А вот примеры строк, не соответствующих этому регулярному выражению:

```
AAAAA      // Нет B
  AAAABC   // В начале строки пробелы
AABBCC     // Слишком много C
```

Часть шаблона рассматривается как подшаблон (который может быть извлечен отдельно из `smatch`), если он находится в круглых скобках. Например:

```
\d+-\d+       // Подшаблона нет
\d+(-\d+)     // Один подшаблон
(\d+)(-\d+)   // Два подшаблона
```

Шаблон может быть сделан необязательным или повторяющимся (по умолчанию он повторяется ровно один раз) с помощью суффикса.

	Повторения
`{n}`	Ровно `n` раз
`{n,}`	`n` или большее количество раз
`{n,m}`	Не менее `n` и не более `m` раз
`*`	Нуль или большее количество раз, т.е. `{0,}`
`+`	Один или большее количество раз, т.е. `{1,}`
`?`	Необязателен (нуль или один раз), т.е. `{0,1}`

Например:

```
A{3}B{2,4}C*
```

Вот примеры строк, соответствующих этому регулярному выражению:

```
AAABBC
AAABBB
```

А вот примеры строк, не соответствующих этому регулярному выражению:

```
AABBC          // Слишком мало A
AAABC          // Слишком мало B
AAABBBBBCCC    // Слишком много B
```

Суффикс `?` после любого указателя повторений (`?`, `*`, `+` и `{}`) делает шаблон "ленивым" или "нежадным", т.е. в процессе поиска шаблона он будет

искать кратчайшее соответствие, а не длиннейшее. По умолчанию всегда выполняется поиск наидлиннейшего соответствия; этот принцип известен как *правило наибольшего соответствия* (Max Munch rule). Рассмотрим строку

```
ababab
```

Шаблон `(ab)+` находит всю строку `ababab`, но шаблон `(ab)+?` находит только первое `ab`.

Наиболее распространенные классификации символов имеют имена.

	Классы символов
`alnum`	Любая буква или цифра
`alpha`	Любая буква
`blank`	Любой пробельный символ, кроме разделителя строк
`cntrl`	Любой управляющий символ
`d`	Любая десятичная цифра
`digit`	Любая десятичная цифра
`graph`	Любой графический символ
`lower`	Любой символ в нижнем регистре
`print`	Любой выводимый символ
`punct`	Любой символ пунктуации
`s`	Любой пробельный символ
`space`	Любой пробельный символ
`upper`	Любой символ в верхнем регистре
`w`	Любой символ слова (буква, цифра или подчеркивание)
`xdigit`	Любая шестнадцатеричная цифра

В регулярных выражениях имя класса символов должно находиться в специальных скобках — `[::]`. Например, `[:digit:]` соответствует десятичной цифре. Кроме того, они должны использоваться в паре скобок `[]`, определяющей класс символов.

Для некоторых классов символов поддерживается сокращенная запись.

	Аббревиатуры для классов символов	
`\d`	Десятичная цифра	`[[:digit:]]`
`\s`	Пробельный символ (пробел, табуляция и т.п.)	`[[:space:]]`
`\w`	Буква (`a-z`), цифра (`0-9`) или подчеркивание (`_`)	`[_[:alnum:]]`

Окончание табл.

	Аббревиатуры для классов символов	
\D	Не \d	[^[:digit:]]
\S	Не \s	[^[:space:]]
\W	Не \w	[^_[:alnum:]]

Кроме того, языки с поддержкой регулярных выражений часто предоставляют следующее.

Нестандартные (но распространенные) сокращения для классов символов		
\l	Символы в нижнем регистре	[[:lower:]]
\u	Символы в верхнем регистре	[[:upper:]]
\L	Не \l	[^[:lower:]]
\U	Не \u	[^[:upper:]]

Для обеспечения переносимости используйте имена классов символов, а не эти аббревиатуры.

В качестве примера рассмотрим написание шаблона, который описывает идентификаторы C++: символ подчеркивания или буква, за которой следует, возможно, пустая последовательность букв, цифр или символов подчеркивания. Чтобы проиллюстрировать ряд тонкостей, я включаю в пример несколько ложных попыток:

```
// Ошибка: символы множества ":alpha", за которыми следует...:
[:alpha:][:alnum:]*

// Ошибка: не принимает подчеркивание ('_' не относится к alpha):
[[:alpha:]][[:alnum:]]*

// Ошибка: подчеркивание не входит и в alnum:
([[:alpha:]]|_)[[:alnum:]]*

// OK, но топорно:
([[:alpha:]]|_)([[:alnum:]]|_)*

// OK: включает подчеркивание в классы символов:
[[:alpha:]_][[:alnum:]_]*

// Также OK:
[_[:alpha:]][_[:alnum:]]*

// \w - эквивалент для [_[:alnum:]]:
[_[:alpha:]]\w*
```

Наконец, вот функция, которая использует простейшую версию `regex_match()` (§9.4.1) для проверки, является ли строка идентификатором:

```
bool is_identifier(const string& s)
{
    regex pat {"[_[:alpha:]]\\w*"}; // Подчеркивание или буква,
            // за которой следует нуль или большее количество
            // подчеркиваний, букв или цифр
    return regex_match(s,pat);
}
```

Обратите внимание на удвоение обратной косой черты при включении обратной косой черты в обычный строковый литерал. Используйте необработанные строковые литералы, чтобы облегчить работу со специальными символами. Например:

```
bool is_identifier(const string& s)
{
    regex pat {R"([_[:alpha:]]\w*)"};
    return regex_match(s,pat);
}
```

Вот несколько примеров шаблонов:

```
Ax*           // A, Ax, Axxxx
Ax+           // Ax, Axxx                        Не A
\d-?\d        // 1-2, 12                         Не 1--2
\w{2}-\d{4,5} // Ab-1234, XX-54321, 22-5432      \w включает цифры
(\d*:)?(\d+)  // 12:3, 1:23, 123, :123           Не 123:
(bs|BS)       // bs, BS                          Не bS
[aeiouy]      // a, o, u - английские гласные    Не x
[^aeiouy]     // x, k - английские согласные     Не e
[a^eiouy]     // a, ^, o, u - английские гласные или ^
```

Группа (подшаблон), потенциально представимая с помощью `sub_match`, отделяется круглыми скобками. Если вам нужны круглые скобки, не определяющие подшаблон, используйте `(?:` вместо простого `(`. Например:

```
(\s|:|,)*(\d*)  // Необязательные пробельные символы, двоеточия
     // и/или запятые, за которыми следуют необязательные цифры
```

В предположении, что нас не интересуют символы до числа (предположительно разделители), можно написать.

```
(?:\s|:|,)*(\d*) // Необязательные пробельные символы, двоеточия
      // и/или запятые, за которыми следуют необязательные цифры
```

Это предохранит механизм регулярных выражений от необходимости сохранять первые символы: вариант `(?:` содержит один только подшаблон.

	Примеры группировки регулярных выражений
`\d*\s\w+`	Групп нет (подшаблоны)
`(\d*)\s(\w+)`	Две группы
`(\d*)(\s(\w+))+`	Две группы (группы не вкладываются)
`(\s*\w*)+`	Одна группа; один или несколько подшаблонов; в качестве `sub_match` сохраняется только последний подшаблон
`<(.*?)>(.*?)</\1>`	Три группы; `\1` означает "такая же, как и группа 1"

Последний шаблон в таблице полезен для синтаксического анализа XML. Он находит маркеры начала/конца дескриптора. Обратите внимание, что я использовал нежадное (ленивое) соответствие, `.*?`, для подшаблона между начальным и конечным дескрипторами. Если бы я использовал просто `.*`, это вызвало бы проблемы:

```
Always look on the <b>bright</b> side of <b>life</b>.
```

Жадное соответствие первого подшаблона привело бы к тому, что первая открывающая угловая скобка < была бы сопоставлена с последней закрывающей >. Это было бы корректным поведением, но вряд ли тем, которое хотел программист.

Более подробное изложение регулярных выражений приведено в [22].

9.4.3. Итераторы

Мы можем определить `regex_iterator` для итерирования последовательности символов, обнаруживающих совпадение с шаблоном. Например, можно использовать `sregex_iterator` (`regex_iterator<string>`) для вывода всех разделенных пробельными символами слов в строке `string`:

```
void test()
{
    string input = "aa as; asd ++e^asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input.begin(),input.end(),pat);
         p!=sregex_iterator{}; ++p)
        cout << (*p)[1] << '\n';
}
```

Это дает нам следующий вывод:

```
as
asd
asdfg
```

Первое слово — `aa` — пропущено, так как ему не предшествуют пробельные символы. Если упростить шаблон до `R"((\w+))"`, то получится

```
aa
as
asd
e
asdf
asdfg
```

Итератор `regex_iterator` является двунаправленным итератором, так что мы не можем непосредственно итерировать входной поток `istream` (который предоставляет только входной итератор). Мы также не можем записывать с помощью `regex_iterator`, а `regex_iterator` по умолчанию (`regex_iterator{}`) является единственно возможным концом последовательности.

9.5. Советы

[1] Используйте `std::string` для владения последовательностями символов; §9.2; [CG:SL.str.1].

[2] Предпочитайте операции над `string` функциям для работы со строками в стиле C; §9.1.

[3] Используйте `string` для объявления переменных и членов, но не базового класса; §9.2.

[4] Возвращайте `string` по значению (положившись на семантику перемещения); §9.2, §9.2.1.

[5] Непосредственно или косвенно используйте для чтения подстрок `substr()` и `replace()` для их записи; §9.2.

[6] `string` при необходимости может расти и сокращаться; §9.2.

[7] При необходимости проверки выхода за границы диапазона используйте `at()`, а не итераторы или индексы `[]`; §9.2.

[8] Используйте итераторы или индексы `[]`, а не `at()`, если хотите оптимизировать скорость; §9.2.

[9] Ввод в `string` не приводит к переполнению; §9.2, §10.3.

[10] Используйте `c_str()` для получения представления `string` в виде строки в стиле C, (только) когда она вам требуется; §9.2.

[11] Используйте `stringstream` или обобщенную функцию извлечения значения (наподобие `to<X>`) для преобразования строк в числа; §10.8.

[12] Класс `basic_string` может быть использован для создания строк из символов любого типа; §9.2.1.

[13] Используйте суффикс `s` для строковых литералов, которые должны означать `string` стандартной библиотеки; §9.3 [CG:SL.str.12].

[14] Используйте `string_view` в качестве аргумента функций, которые должны читать последовательности символов, хранящиеся разными способами; §9.3 [CG:SL.str.2].

[15] Используйте `gsl::string_span` в качестве аргумента функций, которые должны записывать последовательности символов, хранящиеся разными способами; §9.3. [CG:SL.str.2], [CG:SL.str.11].

[16] Рассматривайте `string_view` как разновидность указателя, дополненного размером; он не владеет символами, на которые указывает; §9.3.

[17] Используйте суффикс `sv` для строковых литералов, которые должны представлять `string_view` стандартной библиотеки; §9.3.

[18] Используйте `regex` для большинства обычных применений регулярных выражений; §9.4.

[19] Для выражения всех, кроме самых простых, шаблонов регулярных выражений предпочитайте применять необработанные строковые литералы; §9.4.

[20] Используйте `regex_match()` для проверки соответствия входных данных полностью; §9.4, §9.4.2.

[21] Используйте `regex_search()` для поиска шаблона во входном потоке; §9.4.1.

[22] Запись регулярных выражений может быть скорректирована для соответствия различным стандартам; §9.4.2.

[23] По умолчанию регулярные выражения используют запись, принятую в ECMAScript; §9.4.2.

[24] Будьте умеренны, иначе вы рискуете превратить свой код в совершенно нечитаемую программу на языке регулярных выражений; §9.4.2.

[25] Обратите внимание, что `\i` позволяет выразить подшаблон через предыдущий подшаблон; §9.4.2.

[26] Используйте `?`, чтобы сделать шаблон “ленивым”; §9.4.2.

[27] Используйте `regex_iterator` для итерирования последовательности символов, обнаруживающих совпадение с шаблоном; §9.4.3.

10

Ввод и вывод

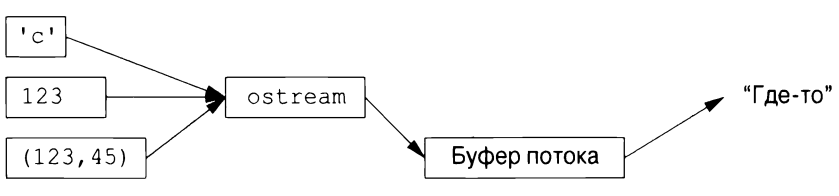
Что вы видите — это все, что вы получите.
— Брайан У. Керниган

- **Введение**
- **Вывод**
- **Ввод**
- **Состояние ввода-вывода**
- **Ввод-вывод пользовательских типов**
- **Форматирование**
- **Файловые потоки**
- **Строковые потоки**
- **Ввод-вывод в стиле C**
- **Файловая система**
- **Советы**

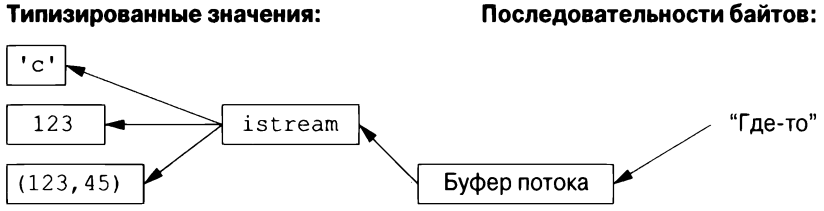
10.1. Введение

Библиотека потоков ввода-вывода обеспечивает форматированный и неформатированный буферизованный ввод-вывод текстовых и числовых значений.

`ostream` преобразует типизированные объекты в поток символов (байтов).



`istream` преобразует поток символов (байтов) в типизированные объекты.



Операции над потоками `istream` и `ostream` описаны в §10.2 и §10.3. Операции безопасны по отношению к типам и расширяемы для обработки пользовательских типов (§10.5).

Другие формы взаимодействия с пользователем, такие как графический ввод-вывод, обрабатываются через библиотеки, которые не являются частью стандарта ISO и поэтому здесь не описаны.

Рассматриваемые потоки могут использоваться для бинарного ввода-вывода и для различных типов символов, использовать локализации и передовые стратегии буферизации, но все эти темы выходят за рамки данной книги.

Потоки могут использоваться для ввода и вывода из `std::string` (§10.3), для форматирования в буферах `string` (§10.8) и для файлового ввода-вывода (§10.10).

Все классы потоков ввода-вывода имеют деструкторы, которые освобождают все ресурсы, которыми владеют потоки (например, буфера и дескрипторы файлов), т.е. они являются примерами применения идиомы "Захват ресурса есть инициализация" (RAII, §5.3).

10.2. Вывод

В `<ostream>` библиотека потока ввода-вывода определяет вывод для каждого встроенного типа. Кроме того, определить вывод пользовательского типа очень легко (§10.5). Оператор `<<` используется как оператор вывода для объектов типа `ostream`; `cout` представляет собой стандартный выходной поток, а `cerr` — стандартный поток сообщений об ошибках. По умолчанию значения, записанные в `cout`, преобразуются в последовательность символов. Например, чтобы вывести десятичное число `10`, можно написать

```
void f()
{
    cout << 10;
}
```

Этот код помещает символ `1`, за которым следует символ `0`, в стандартный выходной поток.

Точно так же можно записать

```
void g()
{
    int x {10};
    cout << x;
}
```

Вывод различных типов может быть скомбинирован обычным способом:

```
void h(int i)
{
    cout << "Значение i равно ";
    cout << i;
    cout << '\n';
}
```

При вызове `h(10)` мы получим следующий вывод:

```
Значение i равно 10
```

Программисты быстро устают повторять имя выходного потока при выводе нескольких связанных элементов. К счастью, для дальнейшего вывода может использоваться сам результат вывода. Например:

```
void h2(int i)
{
    cout << "Значение i равно " << i << '\n';
}
```

Вывод функции `h2()` в точности такой же, как и функции `h()`.

Символьная константа представляет собой символ, заключенный в одинарные кавычки. Обратите внимание, что символ выводится именно как символ, а не как числовое значение. Например:

```
void k()
{
    int b = 'b';   // Внимание: char неявно преобразуется в int
    char c = 'c';
    cout << 'a' << b << c;
}
```

Целочисленное значение символа `'b'` равно `98` (в кодировке ASCII, используемой в реализации C++, с которой я работаю), поэтому в данном случае мы получаем вывод `a98c`.

10.3. Ввод

В заголовочном файле `<istream>` стандартная библиотека предлагает потоки `istream` для ввода данных. Как и `ostream`, `istream` работают с символьными строковыми представлениями встроенных типов и могут легко быть расширены для работы с пользовательскими типами.

Оператор >> используется в качестве оператора ввода; cin представляет собой стандартный входной поток. Тип правого операнда оператора >> определяет, какие входные данные приемлемы и каков целевой объект ввода. Например:

```
void f()
{
    int i;
    cin >> i; // Чтение целочисленного значения в i
    double d;
    cin >> d; // Чтение числа с плавающей точкой двойной точности в d
}
```

Этот код читает целое число наподобие 1234 из стандартного ввода в целочисленную переменную i и число с плавающей точкой, такое как 12.34e5, в переменную d типа double.

Подобно операциям вывода, операции ввода могут быть объединены в цепочки, так что приведенный выше код можно записать следующим образом:

```
void f()
{
    int i;
    double d;
    cin >> i >> d; // Чтение в переменные i и d
}
```

В обоих случаях чтение целого числа прерывается любым символом, который не является цифрой. По умолчанию оператор >> пропускает начальные пробельные символы, поэтому приемлемой полной последовательностью ввода будет следующая:

```
1234
12.34e5
```

Часто нам требуется считать последовательность символов. Это удобно сделать путем чтения в переменную типа string. Например:

```
void hello()
{
    cout << "Введите ваше имя\n";
    string str;
    cin >> str;
    cout << "Hello, " << str << "!\n";
}
```

Если вы введете имя Bjarne, то получите вывод

```
Hello, Bjarne!
```

По умолчанию пробельные символы, такие как пробел или символ новой строки, завершают чтение, так что если вы введете имя и фамилию, например `Bjarne Stroustrup`, то вывод останется прежним:

```
Hello, Bjarne!
```

Прочесть строку полностью можно с помощью функции `getline()`. Например:

```
void hello_line()
{
    cout << "Введите ваше имя\n";
    string str;
    getline(cin,str);
    cout << "Hello, " << str << "!\n";
}
```

Теперь, если вы введете `Bjarne Stroustrup`, то получите ожидаемый вывод:

```
Hello, Bjarne Stroustrup!
```

Символ новой строки, завершающий вводимую строку, отбрасывается, и `cin` готов к вводу очередной строки.

Использование форматированных операций ввода-вывода, как правило, менее подвержено ошибкам, более эффективно и требует меньшего кода, чем работа с символами один за другим. В частности, потоки `istream` заботятся об управлении памятью и проверке диапазона. Мы можем выполнять форматирование и при работе с памятью с помощью строковых потоков `stringstream` (§10.8).

Стандартные строки обладают прекрасным умением расширяться, чтобы хранить все то, что вы в них записываете; при работе с ними вам не требуется предварительно знать максимальный размер строки. Так что, если вы введете пару мегабайтов точек с запятой, программа вернет вам эти страницы с точками с запятой.

10.4. Состояние ввода-вывода

У потока `iostream` есть состояние, которое можно проверить, чтобы определить, удачно ли выполнена операция. Наиболее часто используется чтение последовательности значений:

```
vector<int> read_ints(istream& is)
{
    vector<int> res;
    for (int i; is>>i; )
        res.push_back(i);
    return res;
}
```

Здесь чтение выполняется до тех пор, пока не встретится нечто, что не является целым числом. Это то, что обычно является концом ввода. Здесь происходит следующее — операция `is>>i` возвращает ссылку на `is`, а проверка `iostream` дает `true`, если поток готов к другой операции.

В общем случае состояние ввода-вывода хранит всю информацию, необходимую для чтения или записи, такую как информация о форматировании (§10.6), состояние ошибки (например, достигнут ли конец ввода) и какая буферизация используется. В частности, пользователь может установить состояние отражающим, что произошла некоторая ошибка (§10.5), и очистить состояние, если ошибка не была серьезной. Например, мы могли бы представить себе версию `read_ints()`, которая считывает строку, завершающую ввод:

```
vector<int> read_ints(istream& is, const string& terminator)
{
    vector<int> res;
    for (int i; is >> i; )
        res.push_back(i);

    if (is.eof())    // Все в порядке: конец файла[1]
        return res;

    if (is.fail())   // Ошибка чтения int. Это строка завершения?
    {
        is.clear();  // Сброс состояния в good()
        is.unget(); // Помещаем считанную не цифру назад в поток
        string s;
        if (cin>>s && s==terminator)
            return res;
        // Устанавливаем состояние cin в fail():
        cin.setstate(ios_base::failbit);
    }
    return res;
}

auto v = read_ints(cin,"stop");
```

10.5. Ввод-вывод пользовательских типов

В дополнение к вводу-выводу встроенных типов и стандартных строк `string` библиотека `iostream` позволяет программистам определять ввод-вывод для собственных типов. Например, рассмотрим простой тип `Entry`,

[1] Обратите внимание, что признак конца файла устанавливается **после** неудачной попытки чтения за концом файла, поэтому часто использующаяся начинающими программистами конструкция `while(!is.eof()){ чтение }` приводит к неверным результатам. — *Примеч. пер.*

который мы могли бы использовать для представления записей в телефонной книге:

```
struct Entry {
    string name;
    int number;
};
```

Мы можем легко определить простой оператор вывода для записи `Entry` с использованием формата *{"имя",номер}*:

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

Пользовательский оператор вывода получает поток вывода (по ссылке) в качестве первого аргумента и возвращает его как результат.

Соответствующий оператор ввода более сложный, потому что он должен проверять корректность форматирования и обрабатывать ошибки:

```
istream& operator>>(istream& is, Entry& e)
// Читает пару { "name", number }. Требует {, "" и }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') // Начинается с {"
    {
        // Значение string по умолчанию - пустая строка "":
        string name;
        while (is.get(c) && c!='"') // Все до " является частью name
            name+=c;

        if (is>>c && c==',')
        {
            int number = 0;
            if (is>>number>>c && c=='}') // Читаем число и }
            {
                e = {name ,number};      // Присваивание записи e
                return is;
            }
        }
    }
    is.setstate(ios_base::failbit);      // Фиксация ошибки в потоке
    return is;
}
```

Операция ввода возвращает ссылку на ее поток `istream`, которая может быть использована для проверки успеха операции. Например, при использовании в качестве условия `is>>c` означает "Удалось ли считать символ из `is` в `c`?"

По умолчанию `is>>c` пропускает пробельные символы, чего не делает `is.get(c)`, поэтому оператор ввода для `Entry` игнорирует (опускает) пробелы вне строки имени, но не внутри самого имени. Например:

```
{ "John Marwood Cleese", 123456 }
{"Michael Edward Palin", 987654}
```

Мы можем читать такие пары значений из входного потока в `Entry`, например, следующим образом:

```
for (Entry ee; cin>>ee; ) // Чтение из cin в ee
    cout << ee << '\n';   // Запись ee в cout
```

Вывод будет следующим:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

В §9.4 приведено описание более систематического метода распознавания шаблонов в потоках символов (с использованием соответствия регулярным выражениям).

10.6. Форматирование

Библиотека `iostream` предоставляет большой набор операций для управления форматом ввода и вывода. Простейшие элементы управления форматированием называются *манипуляторами* и описаны в заголовочных файлах `<ios>`, `<istream>`, `<ostream>` и `<iomanip>` (для манипуляторов, которые принимают аргументы). Например, мы можем выводить целые числа в виде десятичных (по умолчанию), восьмеричных или шестнадцатеричных чисел:

```
// Выводит 1234,4d2,2322:
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';
```

Можно явно указать формат вывода чисел с плавающей точкой:

```
constexpr double d = 123.456;
cout << d << "; "                  // Вывод по умолчанию для d
     << scientific   << d << "; "  // Стиль 1.123e2 для d
     << hexfloat     << d << "; "  // Шестнадцатеричная запись для d
     << fixed        << d << "; "  // Стиль 123.456 для d
     << defaultfloat << d << '\n'; // Формат по умолчанию для d
```

Этот код дает следующий вывод:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

Точность — это целое число, определяющее количество цифр, используемых для отображения числа с плавающей точкой.

- *Общий* формат (`defaultfloat`) позволяет реализации выбрать формат, который представляет значение в стиле, который лучше всего сохраняет значение в доступном количестве знакомест. Точность определяет максимальное количество цифр.
- *Научный* формат (`scientific`) представляет значение с одной цифрой до десятичной точки и экспонентой. Точность определяет максимальное количество цифр после десятичной точки.
- *Фиксированный* формат (`fixed`) представляет значение как целочисленную часть, за которой следуют десятичная точка и дробная часть. Точность определяет максимальное количество цифр после десятичной точки.

Значения с плавающей точкой округляются, а не просто усекаются; функция `precision()` не влияет на целочисленный вывод. Например:

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
cout << 1234.56789 << '\n';
```

Это дает следующий вывод:

```
1234.5679 1234.5679 123456
1235 1235 123456
1235
```

Эти манипуляторы с плавающей точкой являются "клейкими", т.е. их влияние сохраняется и для последующих операций с плавающей точкой.

10.7. Файловые потоки

В заголовочном файле `<fstream>` стандартная библиотека предоставляет потоки для чтения из файла и записи в него:

- `ifstream` для чтения из файла;
- `ofstream` для записи в файл;
- `fstream` для чтения из файла и записи в него.

Например:

```
ofstream ofs {"target"}; // "o" означает "output" – вывод
if (!ofs)
    error("Не могу открыть 'target' для записи");
```

Проверка того факта, что файловый поток был успешно открыт, обычно выполняется путем проверки его состояния.

```
ifstream ifs {"source"}; // "i" означает "input" - ввод
if (!ifs)
    error("Не могу открыть 'source' для чтения");
```

В предположении, что тесты успешны, `ofs` можно использовать как обычный `ostream` (точно так же, как `cout`), а `ifs` — как обычный `istream` (точно так же, как `cin`).

Позиционирование файлов и более подробное управление способом открытия файла возможны, но выходят за рамки этой книги.

О составлении имен файлов и манипуляциях файловой системой читайте в §10.10.

10.8. Строковые потоки

В заголовочном файле <sstream> стандартная библиотека предоставляет потоки для записи в строку `string` и чтения из нее:

- `istringstream` для чтения из строки `string`;
- `ostringstream` для записи в строку `string`;
- `stringstream` для чтения из строки и записи в нее.

Например:

```
void test()
{
    ostringstream oss;
    oss << "{Температура," << scientific << 123.4567890 << "}";
    cout << oss.str() << '\n';
}
```

Результат из `ostringstream` можно прочитать с помощью функции `str()`. Одно распространенное использование `ostringstream` заключается в форматировании строки перед передачей ее графическому интерфейсу. Аналогично строка, полученная от графического интерфейса пользователя, может быть прочитана с использованием форматированных операций ввода (§10.3) путем ее помещения в поток `istringstream`.

Строковый поток `stringstream` может использоваться как для чтения, так и для записи. Например, мы можем определить операцию, которая может преобразовывать любой тип со строковым представлением в другой, который также может быть представлен как `string`:

```
template<typename Target =string, typename Source =string>
Target to(Source arg)          // Преобразование Source в Target
{
    stringstream interpreter;
    Target result;
    if (!(interpreter << arg)  ||      // Запись arg в поток
```

```
        !(interpreter>>result) ||       // Чтение result из потока
        !(interpreter>>std::ws).eof()) // Что-то осталось в потоке?
        throw runtime_error{"Ошибка to<>()"};
    return result;
}
```

Аргумент шаблона функции должен быть явно указан только в том случае, если он не может быть выведен или если для него нет значения по умолчанию (§7.2.4), поэтому мы можем написать

```
auto x1 = to<string,double>(1.2); // Явно (и многословно)
auto x2 = to<string>(1.2);        // Source выводится как double
auto x3 = to<>(1.2);              // Target по умолчанию string;
                                  // Source выводится как double
auto x4 = to(1.2);                // Угловые скобки <> излишни;
                                  // Target по умолчанию string;
                                  // Source выводится как double
```

Если все аргументы шаблона функции берутся по умолчанию, угловые скобки <> могут быть опущены.

Я считаю это хорошим примером общности и простоты использования, которые могут быть достигнуты за счет сочетания возможностей языка и средств стандартной библиотеки.

10.9. Ввод-вывод в стиле C

Стандартная библиотека C++ поддерживает также ввод-вывод стандартной библиотеки C, включая функции `printf()` и `scanf()`. Многие применения этой библиотеки являются небезопасными с точки зрения типов и защищенности от взлома, поэтому я не рекомендую ее использовать. В частности, с ее помощью может быть трудно организовать безопасный и удобный ввод. Она не поддерживает пользовательские типы. Если вы *не* используете ввод-вывод в стиле C и беспокоитесь о производительности ввода-вывода, вызовите

```
ios_base::sync_with_stdio(false); // Устранение накладных расходов
```

Без этого вызова потоки `iostream` *могут* быть более медленными из-за обеспечения совместимости со вводом-выводом в стиле C.

10.10. Файловая система

Большинство систем имеют понятие *файловой системы*, обеспечивающей доступ к информации, перманентно хранящейся в виде *файлов*. К сожалению, свойства файловых систем и способы манипулирования ими сильно различаются. Чтобы справиться с этим, библиотека файловой системы `<filesystem>` предлагает единый интерфейс для большинства возможно-

стей большинства файловых систем. Используя <filesystem>, мы можем переносимо

- выражать пути файловой системы и перемещаться по ней;
- исследовать типы файлов и связанные с ними разрешения.

Библиотека файловой системы может работать с Unicode, но подробное объяснение этой темы выходит за рамки книги. Для получения подробной информации я рекомендую сайт *cppreference* [13] и документацию Boost для файловой системы [1].

Рассмотрим пример:

```
path f = "dir/hypothetical.cpp"; // Именование файла
assert(exists(f));               // Файл f должен существовать
if (is_regular_file(f))          // f - обычный файл?
    cout << f << " - файл размером " << file_size(f) << '\n';
```

Обратите внимание, что программа управления файловой системой обычно выполняется на компьютере вместе с другими программами. Таким образом, содержимое файловой системы между двумя командами может меняться. Например, хотя мы, в первую очередь, проверили, что файл f существует, это может оказаться неверным, когда в следующей строке мы выясняем, является ли f обычным файлом.

path — довольно сложный класс, способный обрабатывать различные наборы символов и соглашения многих операционных систем. В частности, он может обрабатывать имена файлов из командной строки, переданные в main(). Например:

```
int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        cerr << "Требуются аргументы\n";
        return 1;
    }

    path p{argv[1]}; // Создание path из командной строки

    // Примечание: path можно вывести как строку:
    cout << p << " " << exists(p) << '\n';

    // ...
}
```

Корректность path не проверяется до его использования. И даже тогда его корректность зависит от соглашений системы, в которой работает программа.

Естественно, что path можно использовать для открытия файла:

```
void use(path p)
{
    ofstream f {p};
    if (!f) error("Неверное имя файла: ", p);
    f << "Hello, file!";
}
```

<filesystem> в дополнение к path предлагает типы для обхода каталогов и запросов о свойствах найденных файлов.

Типы для работы с файловой системой (список неполный)	
path	Путь в каталоге
filesystem_error	Исключение файловой системы
directory_entry	Запись каталога
directory_iterator	Для итерирования по каталогу
recursive_directory_iterator	Для итерирования по каталогу и его подкаталогам

Рассмотрим простой, но не такой уж нереальный пример:

```
void print_directory(path p)
try
{
    if (is_directory(p))
    {
        cout << p << ":\n";
        for (const directory_entry& x : directory_iterator{p})
            cout << " " << x.path() << '\n';
    }
}
catch (const filesystem_error& ex)
{
    cerr << ex.what() << '\n';
}
```

Строка может быть неявно преобразована в path, поэтому мы можем использовать print_directory следующим образом:

```
void use()
{
    print_directory(".");  // Текущий каталог
    print_directory(".."); // Родительский каталог
    print_directory("/");  // Корневой каталог Unix
    print_directory("c:"); // Том C в Windows
    for (string s; cin>>s; )
        print_directory(s);
}
```

Если бы я хотел перечислить еще и подкаталоги, я написал бы `recursive_directory_iterator{p}`. Если бы я хотел вывести записи в лексикографическом порядке, я скопировал бы пути `path` в вектор и отсортировал их перед выводом.

Класс `path` предлагает множество распространенных и полезных операций.

Операции с `path` (список неполный) `p` и `p2` представляют собой объекты `path`	
`value_type`	Тип символов, используемый кодировкой файловой системы: `char` — в POSIX, `wchar_t` — в Windows
`string_type`	`std::basic_string<value_type>`
`const_iterator`	Константный двунаправленный итератор с `value_type` для `path`
`iterator`	Псевдоним для `const_iterator`
`p=p2`	Присваивание `p2` объекту `p`
`p/=p2`	Конкатенация `p` и `p2` с использованием разделителя имен файлов (по умолчанию — `/`)
`p+=p2`	Конкатенация `p` и `p2` (без разделителя)
`p.native()`	"Родной" для системы формат `p`
`p.string()`	`p` в "родном" формате в виде строки
`p.generic_string()`	`p` в обобщенном формате в виде строки
`p.filename()`	Часть имени файла в `p`
`p.stem()`	Имя файла без расширения в `p`
`p.extension()`	Расширение имени файла в `p`
`p.begin()`	Начало последовательности элементов `p`
`p.end()`	Конец последовательности элементов `p`
`p==p2, p!=p2`	Равенство и неравенство `p` и `p2`
`p<p2, p<=p2, p>p2, p>=p2`	Лексикографические сравнения
`is>>p, os<<p`	Потоковый ввод-вывод в `p` и из него
`u8path(s)`	Путь из исходного значения `s` в кодировке UTF-8

Например:

```
void test(path p)
{
    if (is_directory(p))
    {
        cout << p << ":\n";
        for (const directory_entry& x : directory_iterator(p))
        {
```

```
            const path& f = x;    // Ссылка на path записи каталога
            if (f.extension() == ".exe")
                cout << f.stem() << " - выполнимый файл Windows\n";
            else
            {
                string n = f.extension().string();
                if (n == ".cpp" || n == ".C" || n == ".cxx")
                    cout << f.stem() << " - исходный текст C++\n";
            }
        }
    }
}
```

Мы используем `path` как строку, из которой можем получить строки с различной информацией (например, расширение `f.extension().string()`).

Обратите внимание, что соглашения об именах, естественные языки и кодировки строк отличаются большой сложностью. Абстракции библиотеки для файловых систем предлагают переносимость и существенное упрощение проблем.

Операции над файловой системой (список неполный) `p`, `p1` **и** `p2` **представляют собой** `path`; `e` **является** `error_code`; `b` **— значение типа** `bool`, **указывающее успешность операции**	
`exists(p)`	Ссылается ли `p` на существующий объект файловой системы?
`copy(p1,p2)`	Копирование файлов или каталогов из `p1` в `p2`; сообщение об ошибках в виде исключений
`copy(p1,p2,e)`	Копирование файлов или каталогов; сообщение об ошибках в виде кодов ошибок
`b=copy_file(p1,p2)`	Копирование содержимого файла из `p1` в `p2`; сообщение об ошибках в виде исключений
`b=create_directory(p)`	Создание нового каталога `p`; все промежуточные каталоги `p` должны существовать
`b=create_directories(p)`	Создание нового каталога `p`; создание также всех промежуточных каталогов для `p`
`p=current_path()`	`p` — текущий рабочий каталог
`current_path(p)`	Делает `p` текущим рабочим каталогом
`s=file_size(p)`	`s` — количество байтов в `p`
`b=remove(p)`	Удаление `p`, если это файл или пустой каталог

Многие операции имеют перегрузки, которые принимают дополнительные аргументы, такие как разрешения операционных систем. Обработка таковых

выходит далеко за рамки данной книги, поэтому вам придется поискать соответствующую информацию, если она вам понадобится, самостоятельно.

Как и `copy()`, все операции имеют две версии.

- Основная версия, показанная в таблице, например `exists(p)`. Эта функция генерирует исключение `filesystem_error` в случае неудачи операции.
- Версия с дополнительным аргументом `error_code&`, например `exists(p,e)`. Проверяйте значение `e`, чтобы убедиться в успешности операции или выяснить причины ее неудачи.

Мы используем коды ошибок, когда ожидается, что при обычном использовании операции будут часто неудачны, и генерацию исключений, когда ошибка рассматривается как исключительная ситуация.

Зачастую использование функции с запросом информации — самый простой и прямой подход к изучению свойств файла. Библиотека `<filesystem>` знает о нескольких распространенных типах файлов и классифицирует остальные как "иные".

Типы файлов (`f` представляет собой `path` или `file_status`)	
`is_block_file(f)`	Является ли `f` блочным устройством?
`is_character_file(f)`	Является ли `f` символьным устройством?
`is_directory(f)`	Является ли `f` каталогом?
`is_empty(f)`	Является ли `f` пустым файлом или каталогом?
`is_fifo(f)`	Является ли `f` именованным каналом?
`is_other(f)`	Является ли `f` некоторой иной разновидностью файла?
`is_regular_file(f)`	Является ли `f` обычным файлом?
`is_socket(f)`	Является ли `f` именованным сокетом IPC?
`is_symlink(f)`	Является ли `f` символической ссылкой?
`status_known(f)`	Известно ли состояние файла, связанного с `f`?

10.11. Советы

[1] Потоки `iostream` безопасны с точки зрения типов, чувствительны к типам и расширяемы; §10.1.

[2] Используйте ввод на уровне символов, только когда вы вынуждены к нему прибегнуть; §10.3; [CG:SL.io.1].

[3] При чтении всегда учитывайте возможность неверно отформатированных данных; §10.3; [CG:SL.io.2].

[4] Избегайте `endl` (если вы не знаете, что такое `endl`, вы ничего не пропустили); [CG:SL.io.50].

[5] Определяйте << и >> для пользовательских типов со значениями, имеющими значимые текстовые представления; §10.1, §10.2, §10.3.

[6] Используйте `cout` для обычного вывода и `cerr` — для вывода информации об ошибках; §10.1.

[7] Имеются потоки `iostream` как для обычных, так и для широких символов; кроме того, можно определить `iostream` для символов любого вида; §10.1.

[8] Поддерживается бинарный ввод-вывод; §10.1.

[9] Имеются стандартные `iostream` для стандартных потоков ввода-вывода, файлов и строк `string`; §10.2, §10.3, §10.7, §10.8.

[10] Объединяйте операции << в цепочки для более краткой и ясной записи; §10.2.

[11] Объединяйте операции >> в цепочки для более краткой и ясной записи; §10.3.

[12] Ввод данных в `string` не приводит к переполнению; §10.3.

[13] Оператор >> по умолчанию пропускает ведущие пробельные символы; §10.3.

[14] Используйте состояние потока `fail` для обработки ошибок ввода-вывода с потенциальным восстановлением после ошибки; §10.4.

[15] Можно определить операторы << и >> для пользовательских типов; §10.5.

[16] Не требуется модифицировать `istream` или `ostream` для добавления новых операторов << и >>; §10.5.

[17] Для управления форматированием используйте манипуляторы; §10.6.

[18] Спецификация `precision()` применяется ко всем последующим операциям вывода значений с плавающей точкой; §10.6.

[19] Спецификации формата для значений с плавающей точкой (например, `scientific`) применяются ко всем последующим операциям вывода значений с плавающей точкой; §10.6.

[20] Включите `#include<ios>` для использования стандартных манипуляторов; §10.6.

[21] Включите `#include<iomanip>` для использования стандартных манипуляторов, принимающих аргументы; §10.6.

[22] Не пытайтесь копировать файловый поток.

[23] Не забывайте перед использованием проверять, прикреплен ли файловый поток к файлу; §10.7.

[24] Используйте `stringstream` для форматирования в памяти; §10.8.

[25] Вы можете определить преобразования между любыми двумя типами, которые имеют строковые представления; §10.8.

[26] Ввод-вывод в стиле C не является безопасным с точки зрения типов; §10.9.

[27] Если вы не используете функции в стиле `printf`, вызовите `ios_base::sync_with_stdio(false)`; §10.9; [CG:SL.io.10].

[28] Предпочитайте `<filesystem>` непосредственному использованию определенных интерфейсов операционных систем; §10.10.

11

Контейнеры

Это было ново.
Это было необычно.
Это было просто.
Это должно быть успешным!
— Горацио Нельсон

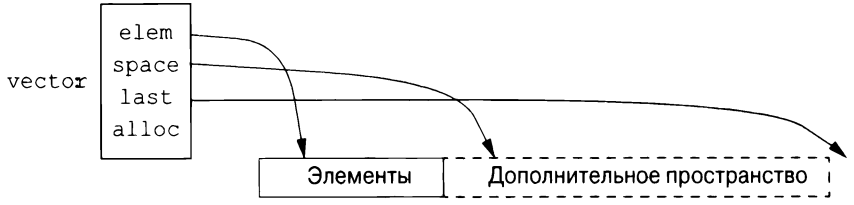
11.1. Введение

Большинство вычислений предусматривают создание коллекций значений с последующим манипулированием такими коллекциями. Простейший пример — чтение символов в строку `string` и вывод этой строки. Класс с основной целью хранения объектов обычно называется *контейнером*. Важным шагом в построении любой программы является создание подходящих контейнеров для данной задачи и их поддержка с помощью полезных фундаментальных операций.

Чтобы проиллюстрировать контейнеры стандартной библиотеки, рассмотрим простую программу для хранения имен и телефонных номеров. Это та разновидность программ, для которой "простыми и очевидными" для людей с разными базовыми знаниями кажутся совершенно разные подходы. Для хранения простой записи телефонной книги может использоваться класс `Entry` из §10.5. Здесь мы сознательно игнорируем многие сложности реального мира, такие как то, что многие номера телефонов не имеют простого представления в виде 32-битного целого числа типа `int`.

11.2. vector

Наиболее полезным контейнером стандартной библиотеки является `vector`. Вектор представляет собой последовательность элементов данного типа. Элементы хранятся в памяти последовательно. Типичная реализация `vector` (§4.2.2, §5.2) будет состоять из дескриптора, хранящего указатель на первый элемент, на элемент, следующий за последним, и на элемент, следующий за выделенной памятью (§12.1) (или эквивалентную информацию, представленную как указатель плюс смещения).



Кроме того, он содержит аллокатор (распределитель памяти, здесь — `alloc`), от которого вектор может получать память для своих элементов. Аллокатор по умолчанию для получения и освобождения памяти использует операторы `new` и `delete` (§13.6).

Мы можем инициализировать `vector` с набором значений его типа элемента:

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Доступ к элементам осуществляется через индексы. Предполагая, что мы определили << для `Entry`, можно написать:

```
void print_book(const vector<Entry>& book)
{
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

Как обычно, индексы начинаются с 0, так что `book[0]` содержит запись для `David Hume`. Функция-член вектора `size()` возвращает количество элементов в векторе.

Элементы вектора составляют диапазон, поэтому мы можем использовать цикл `for` для диапазона (§1.7):

```
void print_book(const vector<Entry>& book)
{
```

```
    for (const auto& x : book) // Об "auto" см. §1.4
        cout << x << '\n';
}
```

При определении vector мы указываем его начальный размер (начальное количество элементов):

```
vector<int> v1 = {1, 2, 3, 4}; // Размер равен 4
vector<string> v2;             // Размер равен 0
// Размер равен 23; начальное значение элементов: nullptr:
vector<Shape*> v3(23);
// Размер равен 32; начальное значение элементов: 9.9:
vector<double> v4(32,9.9);
```

Явно указанный размер заключается в обычные круглые скобки, например (23); по умолчанию элементы инициализируются значением по умолчанию для типа элемента (например, nullptr — для указателей и 0 — для чисел). Если вам не нужно значение по умолчанию, можете указать требуемое значение в качестве второго аргумента (например, 9.9 для 32 элементов v4).

Исходный размер можно изменить. Одной из наиболее полезных операций над вектором является push_back(), которая добавляет новый элемент в конец вектора, увеличивая его размер на единицу. Например, если предположить, что мы определили оператор >> для Entry, можно написать

```
void input()
{
    for (Entry e; cin>>e; )
        phone_book.push_back(e);
}
```

Здесь выполняется чтение записей Entry из стандартного ввода в phone_book до тех пор, пока не будет достигнут конец ввода (например, конец файла) или операция ввода не встретит ошибку формата.

vector стандартной библиотеки реализован таким образом, что увеличение вектора путем многократного применения push_back() оказывается эффективным. Чтобы показать, почему это так, рассмотрим разработку простого Vector из глав 4 и 6, используя представление, показанное на схеме выше:

```
template<typename T>
class Vector
{
    T* elem;  // Указатель на первый элемент
    T* space; // Указатель на первый неиспользуемый
              // (и неинициализированный) слот
    T* last;  // Указатель на последний слот
public:
    // ...
    int size();     // Количество элементов (space-elem)
```

```
    int capacity(); // Количество свободных слотов (last-elem)
    // ...
    void reserve(int newsz);     // Увеличение capacity() до newsz
    // ...
    void push_back(const T& t); // Копирование t в Vector
    void push_back(T&& t);      // Перемещение t в Vector
};
```

vector стандартной библиотеки имеет члены capacity(), reserve() и push_back(). Функция reserve() используется пользователями vector и другими членами vector для выделения памяти для дополнительных элементов. Когда требуется выделение памяти для новых элементов, имеющиеся элементы перемещаются в новое местоположение.

При наличии функций capacity() и reserve() реализация push_back() тривиальна:

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
    if (capacity()<size()+1)           // Если места для t нет,
        reserve(size()==0?8:2*size()); // удваиваем емкость вектора
    new(space) T{t};                   // t инициализирует *space
    ++space;
}
```

Теперь распределение и перемещение элементов происходят нечасто. Раньше я использовал reserve(), пытаясь повысить производительность, но это оказалось пустой тратой усилий: эвристика, используемая vector, в среднем оказалась лучше моих догадок, так что теперь я явно использую reserve() только для того, чтобы избежать перераспределения элементов, когда планирую использовать указатели на элементы.

vector может быть скопирован путем присваивания или инициализации. Например:

```
vector<Entry> book2 = phone_book;
```

Копирование и перемещение векторов реализуются конструкторами и операторами присваивания, как описано в §5.2. Присваивание вектора сопровождается копированием его элементов. Таким образом, и book2, и phone_book после инициализации book2 хранят отдельные копии каждой записи Entry в телефонной книге. Когда вектор содержит много элементов, столь невинно выглядящие присваивания и инициализации могут стать весьма дорогими. Если копирование нежелательно, следует использовать ссылки или указатели (§1.7) или операции перемещения (§5.2.2).

Вектор стандартной библиотеки очень гибкий и эффективный. Используйте его как контейнер по умолчанию, т.е. используйте его, если только у вас

нет веской причины использовать какой-либо другой контейнер. Если вы избегаете вектора из-за сомнений в его эффективности, выполните измерения. Наша интуиция в особенности ошибочна в вопросах эффективности использования контейнеров.

11.2.1. Элементы

Как и все контейнеры стандартной библиотеки, `vector` представляет собой контейнер элементов некоторого типа `T`, т.е. `vector<T>`. Почти любой тип может быть типом элемента: встроенные числовые типы (такие, как `char`, `int` или `double`), пользовательские типы (такие, как `string`, `Entry`, `list<int>` или `Matrix<double,2>`) или указатели (такие, как `const char*`, `Shape*` и `double*`). Когда вы вставляете новый элемент, его значение копируется в контейнер. Например, когда вы помещаете в контейнер целое число со значением 7, результирующий элемент действительно имеет значение 7. Этот элемент не является ссылкой или указателем на какой-либо объект, содержащий 7. Это свойство обеспечивает нас компактными контейнерами с быстрым доступом. Для программистов, которые заботятся о размерах памяти и производительности во время выполнения, это очень важно.

Если у вас есть иерархия классов (§4.5), которая основана на виртуальных функциях для получения полиморфного поведения, не храните объекты непосредственно в контейнере. Вместо этого храните указатель (или интеллектуальный указатель; §13.2.1). Например:

```
vector<Shape>  vs;              // Нет! Здесь мало места для
                                // размещения Circle или Smiley
vector<Shape*> vps;             // Лучше, но см. §4.5.3
vector<unique_ptr<Shape>> vups; // OK
```

11.2.2. Проверка выхода за границы диапазона

`vector` стандартной библиотеки не гарантирует проверки выхода за границы диапазона. Например:

```
void silly(vector<Entry>& book)
{
    int i = book[book.size()].number; // book.size() за
    // ...                            // границами диапазона
}
```

Такая инициализация, вероятно, приведет к некоторому случайному значению в `i`, а не к ошибке. Это нежелательно, и ошибки выхода за границы диапазона являются распространенной проблемой. Поэтому я часто использую следующую простую адаптацию вектора:

```
template<typename T>
class Vec : public std::vector<T>
{
public:
    using vector<T>::vector;          // Используем конструктор
                                      // из vector (под именем Vec)

    T& operator[](int i)              // С проверкой выхода
        { return vector<T>::at(i); } // за границы диапазона

    const T& operator[](int i) const // Проверка для константных
        { return vector<T>::at(i); } // объектов; §4.2.1
};
```

`Vec` наследует от `vector` все, за исключением операций индексации, которые он переопределяет с использованием проверки выхода за границу диапазона. Функция `at()` представляет собой операцию индексации `vector`, которая генерирует исключение `out_of_range`, если ее аргумент выходит за пределы диапазона вектора (§3.5.1).

Для `Vec` доступ за границами диапазона сгенерирует исключение, которое пользователь может перехватить. Например:

```
void checked(Vec<Entry>& book)
{
    try
    {
        book[book.size()] = {"Joe",999999}; // Генерирует исключение
        // ...
    }
    catch (out_of_range&)
    {
        cerr << "Выход за границы диапазона\n";
    }
}
```

Здесь будет сгенерировано, а затем перехвачено исключение (§3.5.1). Если пользователь не перехватит исключение, программа будет завершена в точно определенном порядке, а не будет продолжать выполнение или сбоить некоторым неопределенным образом. Один из способов свести к минимуму сюрпризы из-за неперехваченных исключений — использовать `main()` с `try`-блоком в качестве тела. Например:

```
int main()
try
{
    // Ваш код
}
```

```
catch (out_of_range&)
{
    cerr << "Ошибка выхода за границы диапазона\n";
}
catch (...)
{
    cerr << "Перехвачено неизвестное исключение\n";
}
```

Этот подход обеспечивает обработчик исключений по умолчанию, поэтому, если мы не сможем перехватить какое-то из исключений, в стандартный поток диагностики ошибок `cerr` (§10.2) будет выведено сообщение о неизвестном исключении.

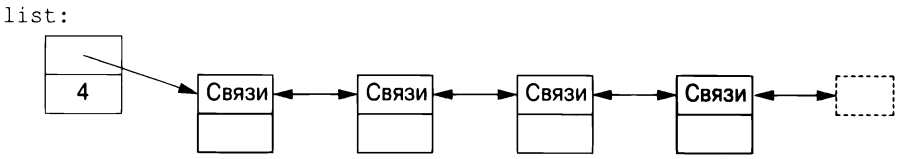
Почему стандарт не гарантирует проверку выхода за границы диапазона? Многие критически важные приложения используют `vector`, и проверка всех индексов подразумевает снижение эффективности на величину порядка 10%. Очевидно, что эта стоимость может сильно различаться в зависимости от используемого аппаратного обеспечения, оптимизаторов и приложения, которое использует индексы. Однако опыт показывает, что такие накладные расходы могут заставить людей предпочесть гораздо более опасные встроенные массивы. Привести к отказу от векторов может даже простой страх перед такими накладными расходами. Как минимум вектор легко проверяется во время отладки, а при необходимости можно легко создать версию вектора с проверкой поверх версии по умолчанию — без проверки выхода за границы диапазона. Некоторые реализации избавляют вас от необходимости определять собственный класс `Vec` (или его эквивалент), предоставляя версию вектора с проверкой выхода за границы диапазона (например, в качестве опции компилятора).

Цикл `for` для диапазона позволяет отказаться от каких-либо проверок без каких-либо затрат, обеспечивая доступ к элементам через итераторы в диапазоне `[begin(),end())`. То же самое относится к алгоритмам стандартной библиотеки: пока их аргументы, являющиеся итераторами, корректны, гарантируется отсутствие ошибок выхода за границы диапазона.

Если вы можете использовать `vector::at()` непосредственно в коде, то обходное решение `Vec` вам не понадобится. Кроме того, в некоторых стандартных библиотеках реализованы реализации `vector` с проверкой выхода за границы диапазона, которые предлагают более полную проверку, чем `Vec`.

11.3. `list`

Стандартная библиотека предлагает двусвязный список под названием `list`.



Мы используем `list` для последовательностей, в которые хотим эффективно вставлять (и удалять) элементы, не перемещая при этом другие элементы. Вставка и удаление записей телефонной книги может быть распространенной операцией, поэтому для представления простой телефонной книги подходящей структурой данных может оказаться `list`. Например:

```
list<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Используя связанный список, мы, как правило, не обращаемся к элементам с использованием индексов, как обычно делаем это для векторов. Вместо этого мы можем искать в списке элемент с заданным значением. Для этого мы воспользуемся тем фактом, что `list` представляет собой последовательность, описанную в главе 12, "Алгоритмы":

```
int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0; // Используем 0 для представления "значение не найдено"
}
```

Поиск `s` начинается с начала списка и продолжается до тех пор, пока не будет найдено значение `s` или не будет достигнут конец списка `phone_book`.

Иногда нам нужно идентифицировать элемент списка. Например, мы можем удалить элемент или вставить новый элемент перед ним. Для этого мы используем *итератор*: итератор списка идентифицирует элемент списка и может использоваться для итераций по списку (отсюда и его имя). Каждый контейнер стандартной библиотеки предоставляет функции `begin()` и `end()`, которые возвращают итераторы, указывающие на первый элемент и на элемент, следующий за последним, соответственно (глава 12, "Алгоритмы"). Используя итераторы явным образом, мы можем — менее элегантно — переписать функцию `get_number()` следующим образом:

```
int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
```

```
        if (p->name==s)
            return p->number;
    return 0; // Используем 0 для представления "значение не найдено"
}
```

На самом деле это примерно то же самое, как если бы компилятор реализовал цикл `for` по диапазону — более кратко и менее чревато ошибками. Для данного итератора `p` запись `*p` — это элемент, на который ссылается итератор; `++p` перемешает `p` для обращения к следующему элементу, а когда `p` относится к классу с членом `m`, то запись `p->m` эквивалентна записи `(*p).m`.

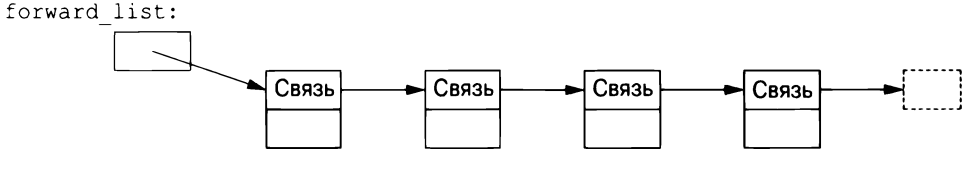
Добавление элементов в `list` и удаление их оттуда выполняется очень легко:

```
void f(const Entry& ee,
       list<Entry>::iterator p,
       list<Entry>::iterator q)
{
    //Добавление ee перед элементом, на который указывает p:
    phone_book.insert(p,ee);

    // Удаление элемента, на который указывает q:
    phone_book.erase(q);
}
```

Для списка `list` вызов `insert(p,elem)` вставляет элемент с копией значения `elem` перед элементом, на который указывает `p`. Здесь `p` может быть итератором, указывающим на элемент, следующий за концом списка. И наоборот, `erase(p)` удаляет элемент, на который указывает `p`, и уничтожает его.

Эти примеры с `list` могут быть записаны идентично с использованием `vector` и (что кажется удивительным, если вы не понимаете архитектуру машины) лучше работают с небольшим вектором, чем с небольшим списком. Когда все, что мы хотим, — это последовательность элементов, у нас есть выбор между использованием `vector` и `list`. Если у вас нет особой причины поступить иначе, используйте `vector`. Вектор лучше работает в случае обхода (например, при работе `find()` и `count()`) и при сортировке и поиске (например, с помощью `sort()` и `equal_range()`, §12.6, §13.4.3).

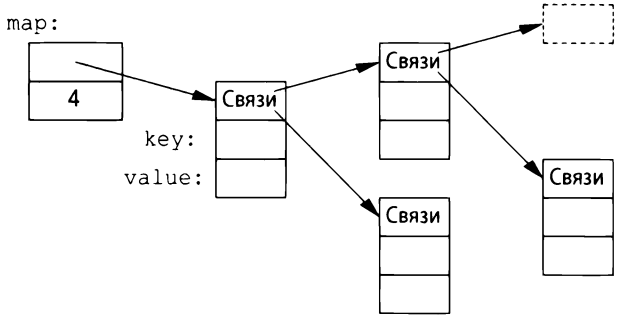
Стандартная библиотека предлагает также односвязный список, именуемый `forward_list`.



Список forward_list отличается от list тем, что допускает итерации только в одном направлении. Главная цель его существования — сэкономить память. В нем нет необходимости хранить указатель на предшественника в каждом элементе, а размер пустого списка forward_list — всего лишь один указатель. forward_list даже не хранит количество элементов. Если вам нужно количество элементов, вы можете посчитать его сами.

11.4. map

Написание кода для поиска имени в списке пар (*имя*,*число*) довольно утомительно. Кроме того, линейный поиск неэффективен везде, кроме кратчайших контейнеров. Стандартная библиотека предлагает сбалансированное двоичное дерево поиска (обычно это красно-черное дерево), именуемое map.



В других контекстах тип отображения map известен как ассоциативный массив или словарь. Он реализован как сбалансированное двоичное дерево.

Тип map стандартной библиотеки представляет собой контейнер из пар значений, оптимизированный для поиска. Мы можем использовать тот же инициализатор, что и для вектора и списка (§11.2, §11.3):

```
map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

При индексации по значению его первого типа (называемому *ключом*) map возвращает соответствующее значение второго типа (называемое *значением* или *отображаемым типом*). Например:

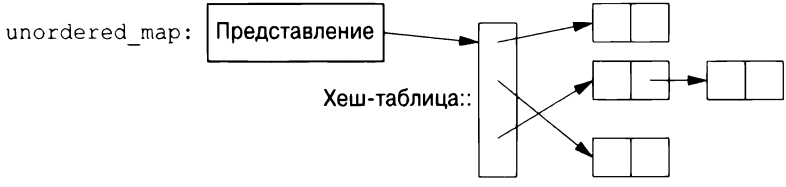
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Другими словами, индексация map — это, по сути, поиск, который мы называли get_number(). Если ключ key не найден, он вносится в map со значением по умолчанию для типа его value. Значение по умолчанию для целочисленного типа равно 0; это значение, которое я выбрал для представления неверного номера телефона.

Если бы мы хотели избежать ввода неверных номеров в нашу телефонную книгу, то могли бы использовать find() и insert() вместо [].

11.5. unordered_map

Стоимость поиска в map — O(log(n)), где n — количество элементов в отображении. Это весьма неплохо. Например, для отображения с 1 000 000 элементов мы выполняем только около 20 сравнений и косвенных обращений, чтобы найти искомый элемент. Однако во многих случаях можно добиться лучшего, используя хеширование, а не сравнение с помощью функции упорядочения, такой как оператор <. Хешированные контейнеры стандартной библиотеки называются "неупорядоченными", потому что не требуют функции упорядочения.



Например, мы можем использовать unordered_map из заголовочного файла <unordered_map> для нашей телефонной книги:

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Подобно map, в случае unordered_map также можно использовать индексацию:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

Стандартная библиотека предоставляет хеш-функцию по умолчанию для строкового типа string, а также для других встроенных типов и типов стандартной библиотеки. При необходимости вы можете предоставить собствен-

ную хеш-функцию (§5.4.6). Вероятно, наиболее распространенная потребность в "пользовательской" хеш-функции возникает, когда нам нужен неупорядоченный контейнер одного из наших собственных типов. Хеш-функция часто предоставляется в виде функционального объекта (§6.3.2). Например:

```
struct Record
{
    string name;
    int product_code;
    // ...
};

struct Rhash                        // Хеш-функция для типа Record
{
    size_t operator()(const Record& r) const
    {
        return hash<string>()(r.name)^hash<int>()(r.product_code);
    }
};

// Множество записей Record с использованием Rhash для поиска:
unordered_set<Record,Rhash> my_set;
```

Проектирование хороших хеш-функций — это искусство, требующее знаний данных, к которым они будут применяться. Часто простым и эффективным способом является создание новой хеш-функции путем объединения существующих с использованием побитового исключающего или (^).

Мы можем избежать явной передачи хеш-операции, определяя ее как специализацию `hash` стандартной библиотеки:

```
namespace std       // Создание хеш-функции для Record
{
    template<> struct hash<Record>
    {
        using argument_type = Record;
        using result_type = std::size_t;
        size_t operator()(const Record& r) const
        {
            return hash<string>()(r.name)^hash<int>()(r.product_code);
        }
    };
}
```

Обратите внимание на отличие `map` и `unordered_map`.

- `map` требует функции упорядочения (по умолчанию используется оператор <) и дает упорядоченную последовательность.
- `unordered_map` требует хеш-функцию и не поддерживает порядок элементов.

При наличии хорошей хеш-функции unordered_map при больших размерах оказывается куда быстрее, чем map. Однако в наихудшем случае поведение unordered_map с плохой хеш-функцией оказывается гораздо хуже, чем поведение map.

11.6. Обзор контейнеров

Стандартная библиотека предоставляет ряд наиболее общих и полезных типов контейнеров, позволяющих программисту выбрать контейнер, который наилучшим образом удовлетворяет потребностям приложения.

	Стандартные контейнеры
vector<T>	Вектор переменного размера (§11.2)
list<T>	Двусвязный список (§11.3)
forward_list<T>	Односвязный список
deque<T>	Двусторонняя очередь
set<T>	Множество (отображение только с ключом, без значения)
multiset<T>	Множество, в котором одно и то же значение может встречаться неоднократно
map<K,V>	Ассоциативный массив — отображение (§11.4)
multimap<K,V>	Отображение, в котором один и тот же ключ может встречаться неоднократно
unordered_map<K,V>	Отображение с использованием поиска на основе хеширования (§11.5)
unordered_multimap<K,V>	Мультиотображение с использованием поиска на основе хеширования
unordered_set<T>	Множество с использованием поиска на основе хеширования
unordered_multiset<T>	Мультимножество с использованием поиска на основе хеширования

Неупорядоченные контейнеры оптимизированы для поиска с помощью ключа (часто строки); другими словами, они реализуются с использованием хеш-таблиц.

Контейнеры определены в пространстве имен std и представлены в заголовочных файлах <vector>, <list>, <map> и т.д. (§8.3). Кроме того, стандартная библиотека предоставляет адаптеры контейнеров queue<T>, stack<T> и priority_queue<T>. Познакомьтесь с ними, если ваше приложение нуждается в соответствующих структурах данных. Стандартная би-

блиотека предоставляет также более специализированные контейнерообразные типы, такие как массив фиксированного размера `array<T,N>` (§13.4.1) и множество битов `bitset<N>` (§13.4.2).

Стандартные контейнеры и их основные операции спроектированы таким образом, чтобы быть похожими с точки зрения записи. Кроме того, смыслы операций эквивалентны для различных контейнеров. Основные операции применяются ко всем типам контейнеров, для которых они имеют смысл и могут быть эффективно реализованы.

Операции со стандартными контейнерами (список неполный)	
`value_type`	Тип элемента
`p=c.begin()`	`p` указывает на первый элемент `c`; имеется также функция `cbegin()`, возвращающая константный итератор
`p=c.end()`	`p` указывает на элемент `c`, следующий за последним; имеется также функция `cend()`, возвращающая константный итератор
`k=c.size()`	`k` равно количеству элементов в `c`
`c.empty()`	Является ли контейнер `c` пустым?
`k=c.capacity()`	`k` равно количеству элементов, которые `c` в состоянии хранить без нового выделения памяти
`c.reserve(k)`	Делает емкость равной `k`
`c.resize(k)`	Делает количество элементов равным `k`; добавленные элементы имеют значение `value_type{}`
`c[k]`	`k`-й элемент `c`; проверки выхода за границы диапазона нет
`c.at(k)`	`k`-й элемент `c`; при выходе за границы диапазона генерируется исключение `out_of_range`
`c.push_back(x)`	Добавление `x` в конец `c`; увеличение размера `c` на единицу
`c.emplace_back(a)`	Добавление `value_type{a}` в конец `c`; увеличение размера `c` на единицу
`q=c.insert(p,x)`	Добавление `x` в `c` перед `p`
`q=c.erase(p)`	Удаление из `c` элемента, на который указывает `p`
`=`	Присваивание
`==, !=`	Проверка на попарное равенство всех элементов двух контейнеров
`<, <=, >, >=`	Лексикографическое упорядочение

Такое нотационное и семантическое единообразие позволяет программистам создавать новые типы контейнеров, которые могут быть использованы точно так же, как и стандартные. Примером этого является вектор с провер-

кой выхода за границы диапазона `Vector` (§3.5.2, глава 4, "Классы"). Единообразие интерфейсов контейнеров позволяет определять алгоритмы независимо от конкретных типов контейнеров. Однако у каждого контейнера есть свои сильные и слабые стороны. Например, индексация и обход вектора `vector` выполняются легко и дешево. С другой стороны, элементы `vector` перемещаются при вставке или удалении элементов. Список `list` обладает противоположными свойствами. Обратите внимание, что для коротких последовательностей небольших элементов вектор обычно более эффективен, чем список (даже для операций `insert()` и `erase()`). Я рекомендую использовать `vector` стандартной библиотеки как тип по умолчанию для последовательностей элементов: вам нужна серьезная причина для иного выбора.

Рассматривайте односвязный список `forward_list` как контейнер, оптимизированный для пустой последовательности (§11.3). Пустой `forward_list` занимает только одно слово, тогда как пустой `vector` занимает их три. Пустые последовательности и последовательности только с одним или двумя элементами на удивление распространены и полезны.

Операция размещения, такая как `emplace_back()`, принимает аргументы для конструктора элемента и строит объект непосредственно в выделенном пространстве в контейнере, а не копирует объект в контейнер. Например, для `vector<pair<int,string>>` мы могли бы написать

```
// Создание объекта pair и его перемещение в v:
v.push_back(pair(1,"Копирование или перемещение"));

// Создание pair непосредственно в v:
v.emplace_back(1,"Построение на месте");
```

11.7. Советы

[1] Контейнер STL определяет последовательность; §11.2.

[2] Контейнеры STL представляют собой дескрипторы ресурсов; §11.2, §11.3, §11.4, §11.5.

[3] Используйте `vector` в качестве контейнера по умолчанию; §11.2, §11.6; [CG:SL.con.2].

[4] Для простого обхода контейнера используйте цикл по диапазону или пару итераторов `begin()`/`end()`; §11.2, §11.3.

[5] Используйте функцию `reserve()`, чтобы избежать недействительности указателей на элементы и итераторов; §11.2.

[6] Не следует рассчитывать на повышение производительности из-за применения `reserve()` без проведения измерений; §11.2.

[7] Используйте `push_back()` или `resize()` для контейнера вместо `realloc()` для массива; §11.2.

[8] Не используйте итераторы, указывавшие в `vector`, после изменения его размера; §11.2.

[9] Не рассчитывайте на проверку выхода за границы диапазона в операторе `[]`; §11.2.

[10] Используйте `at()`, если вам нужна гарантированная проверка выхода за границы диапазона; §11.2; [CG:SL.con.3].

[11] Используйте цикл `for` для диапазона и алгоритмы стандартной библиотеки, чтобы без затрат избежать проверки выхода за границы диапазона; §11.2.2.

[12] Элементы копируются в контейнер; §11.2.1.

[13] Для сохранения полиморфного поведения элементов храните указатели; §11.2.1.

[14] Операции вставки, такие как `insert()` и `push_back()`, в `vector` зачастую оказываются на удивление эффективными; §11.3.

[15] Используйте `forward_list` для последовательностей, которые обычно пусты; §11.6.

[16] Когда дело касается производительности — выполняйте измерения, не доверяя интуиции; §11.2.

[17] `map` обычно реализуется как красно-черное дерево; §11.4.

[18] `unordered_map` представляет собой хеш-таблицу; §11.5.

[19] Передавайте контейнеры в функции по ссылке, а возвращайте по значению; §11.2.

[20] В случае контейнеров используйте синтаксис инициализации с `()` для размеров и с `{}` — для списков элементов; §4.2.3, §11.2.

[21] Предпочитайте компактные структуры данных с последовательным размещением элементов; §11.3.

[22] Обход списка — относительно дорогостоящая процедура; §11.3.

[23] Используйте неупорядоченные контейнеры, если требуется быстрый поиск среди большого количества данных; §11.5.

[24] Используйте упорядоченные ассоциативные контейнеры (такие, как `map` или `set`), если требуется упорядоченный обход их элементов; §11.4.

[25] Используйте неупорядоченные контейнеры для элементов, типы которых не имеют естественного порядка (например, не имеют разумного оператора <); §11.4.

[26] Для получения приемлемой хеш-функции следует немного поэкспериментировать; §11.5.

[27] Зачастую хорошие хеш-функции получаются при комбинировании стандартных хеш-функций элементов с использованием оператора исключающего или (^); §11.5.

[28] Следует знать контейнеры стандартной библиотеки и предпочитать их самостоятельно создаваемым вручную структурам данных; §11.6.

12

Алгоритмы

Не преумножайте сущности
сверх необходимости.

— Уильям Оккам

- **Введение**
- **Применение итераторов**
- **Типы итераторов**
- **Итераторы потоков**
- **Предикаты**
- **Обзор алгоритмов**
- **Концепты**
- **Алгоритмы над контейнерами**
- **Параллельные алгоритмы**
- **Советы**

12.1. Введение

Структура данных, такая как список или вектор, сама по себе не слишком полезна. Чтобы ее использовать, нужны основные операции доступа, такие как добавление и удаление элементов (предусмотренные, например, для `list` и `vector`). Кроме того, мы редко просто храним объекты в контейнере. Мы сортируем их, выводим, извлекаем подмножества элементов, удаляем, ищем объекты и т.д. Поэтому стандартная библиотека в дополнение к наиболее распространенным типам контейнеров предоставляет и наиболее распространенные алгоритмы для работы с ними. Например, мы можем просто и эффективно отсортировать вектор элементов `Entry` и поместить копию каждого уникального элемента вектора в список:

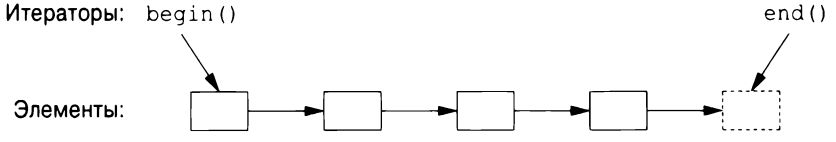
```
void f(vector<Entry>& vec, list<Entry>& lst)
{
    // Для упорядочения используется оператор <:
    sort(vec.begin(),vec.end());
```

```
    // Соседние одинаковые элементы не копируются:
    unique_copy(vec.begin(),vec.end(),lst.begin());
}
```

Чтобы этот код работал, для типа `Entry` должны быть определены операторы меньше (<) и равно (==). Например:

```
bool operator<(const Entry& x, const Entry& y) // Меньше, чем
{
    // Упорядочение записей Entry по именам:
    return x.name<y.name;
}
```

Стандартный алгоритм выражается через (полуоткрытые) последовательности элементов. *Последовательность* представлена парой итераторов, определяющих первый элемент и следующий за последним элементом:



В этом примере функция `sort()` сортирует последовательность, определенную парой итераторов `vec.begin()` и `vec.end()`, которая просто содержит все элементы вектора. Для записи (вывода) нужно указать только первый записываемый элемент. Если записывается более одного элемента, то элементы, следующие за этим начальным, будут перезаписаны. Таким образом, чтобы избежать ошибок, `lst` должен иметь как минимум столько элементов, сколько уникальных значений в `vec`.

Если бы мы хотели разместить уникальные элементы в новом контейнере, то могли бы написать

```
list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(),vec.end());

    // Добавление к res:
    unique_copy(vec.begin(),vec.end(),back_inserter(res));
    return res;
}
```

Вызов `back_inserter(res)` создает итератор для `res`, который добавляет элементы в конец контейнера, расширяя последний так, чтобы для них нашлось место. Это избавляет нас от необходимости сначала выделять фиксированное количество памяти, а затем заполнять его. Таким образом, стандартные контейнеры совместно с `back_inserter()` исключают необходи-

мость использования явного управления памятью в стиле C с использованием функции `realloc()`. У списка `list` стандартной библиотеки имеется конструктор перемещения (§5.2.2), который делает возврат `res` по значению эффективным (даже для списков из тысяч элементов).

Если вы найдете стиль кода с использованием пары итераторов, такой как `sort(vec.begin(),vec.end())`, утомительным, можете определить версии алгоритмов для контейнеров и писать просто `sort(vec)` (§12.8).

12.2. Применение итераторов

Для контейнера можно получить несколько итераторов, ссылающихся на полезные элементы; `begin()` и `end()` — лучшие тому примеры. Кроме того, многие алгоритмы возвращают итераторы. Например, стандартный алгоритм `find` ищет значение в последовательности и возвращает итератор найденного элемента:

```
bool has_c(const string& s, char c) // Есть ли в s символ c?
{
    auto p = find(s.begin(),s.end(),c);
    if (p!=s.end())
        return true;
    else
        return false;
}
```

Как и многие другие алгоритмы поиска стандартной библиотеки, `find` возвращает `end()`, чтобы указать, что поиск завершился неудачно и искомый элемент не найден. Эквивалентное, более короткое определение `has_c()` имеет следующий вид:

```
bool has_c(const string& s, char c) // Есть ли в s символ c?
{
    return find(s.begin(),s.end(),c)!=s.end();
}
```

Более интересным упражнением был бы поиск местоположений всех вхождений символа в строку. Мы можем вернуть множество вхождений как вектор итераторов строк. Возврат вектора эффективен, потому что вектор обеспечивает семантику перемещения (§5.2.1). Предполагая, что мы захотим изменить символы в найденных местоположениях, мы передаем неконстантную строку:

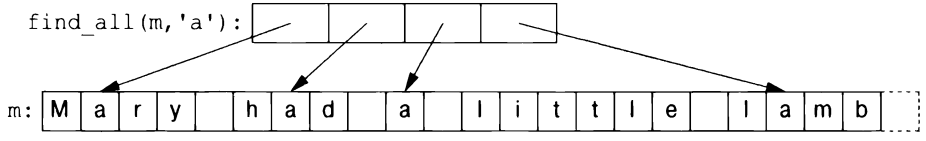
```
// Поиск всех местоположений символа c в строке s:
vector<string::iterator> find_all(string& s, char c)
{
    vector<string::iterator> res;
    for (auto p = s.begin(); p!=s.end(); ++p)
```

```
        if (*p==c)
            res.push_back(p);
    return res;
}
```

Мы проходим по строке с помощью обычного цикла, перемещая итератор `p` вперед по одному элементу за раз с использованием оператора `++` и просматривая элементы с помощью оператора разыменования `*`. Мы могли бы проверить работоспособность `find_all()` следующим образом:

```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))
        if (*p!='a')
            cerr << "Ошибка!\n";
}
```

Этот вызов `find_all()` графически можно представить следующим образом.



Итераторы и стандартные алгоритмы работают одинаково с каждым стандартным контейнером, для которого их использование имеет смысл. Следовательно, можно обобщить функцию `find_all()`:

```
// Поиск всех вхождений v в c:
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)
{
    vector<typename C::iterator> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

Ключевое слово `typename` необходимо, чтобы сообщить компилятору, что итератор `C::iterator` представляет собой тип, а не значение некоторого типа, например целое число `7`. Эту деталь реализации можно скрыть, вводя псевдоним типа (§6.4.2) для `Iterator`:

```
template<typename T>
using Iterator = typename T::iterator; // Итератор у типа T
```

```
template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)// Поиск всех вхождений v в c:
{
    vector<Iterator<C>> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res;
}
```

Теперь можно написать следующий код:

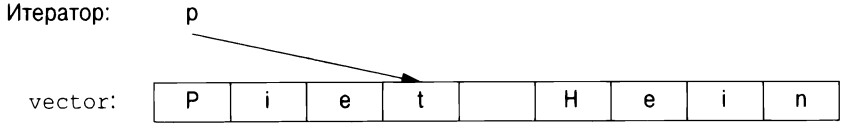
```
void test()
{
    string m {"Mary had a little lamb"};
    for (auto p : find_all(m,'a'))      // p  – string::iterator
        if (*p!='a')
            cerr << "Ошибка строки!\n";

    list<double> ld {1.1, 2.2, 3.3, 1.1};
    for (auto p : find_all(ld,1.1))     // p – list<double>::iterator
        if (*p!=1.1)
            cerr << "Ошибка списка!\n";

    vector<string> vs { "red", "blue", "green",
                        "green", "orange", "green" };
    for (auto p : find_all(vs,"red")) // p – vector<string>::iterator
        if (*p!="red")
            cerr << "Ошибка вектора!\n";
        for (auto p : find_all(vs,"green"))
            *p = "vert";
}
```

Итераторы используются для разделения алгоритмов и контейнеров. Алгоритм управляет своими данными через итераторы и ничего не знает о контейнере, в котором хранятся элементы. И наоборот, контейнер ничего не знает об алгоритмах, работающих с его элементами; все, что он делает, — это предоставляет итераторы по запросу (например, `begin()` и `end()`). Такая модель разделения хранилища данных и алгоритма обеспечивает очень обобщенное и гибкое программное обеспечение.

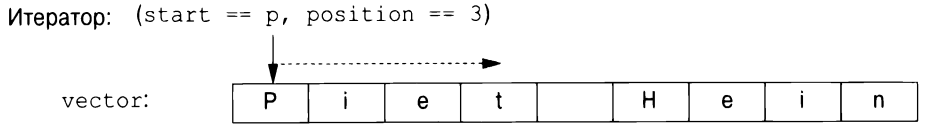
12.3. Типы итераторов

Что же такое итераторы? Любой конкретный итератор является объектом некоторого типа. Однако существует много разных типов итераторов, потому что итератору необходимо хранить информацию, необходимую для выполнения своей работы для определенного типа контейнера. Эти типы итераторов могут различаться в зависимости от контейнеров и специализированных требований, которые к ним предъявляются. Например, итератор у `vector` может

быть обычным указателем, потому что указатель является довольно разумным средством обращения к элементу вектора.

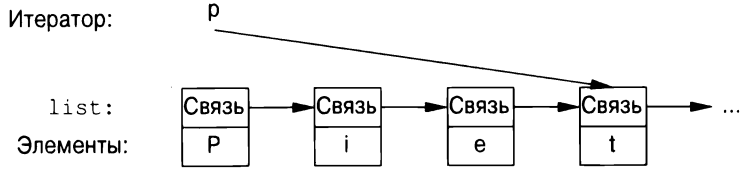


В качестве альтернативы итератор для `vector` можно реализовать как указатель на данные `vector` плюс индекс.



Такой итератор позволяет выполнять проверку выхода за границы диапазона.

Итератор для `list` должен быть чем-то более сложным, чем простой указатель на элемент, потому что элемент списка в общем случае не знает, где находится следующий элемент этого списка. Таким образом, итератор списка может быть указателем на связь со следующим элементом.



Общими для всех итераторов являются их семантика и наименование их операций. Например, применение оператора ++ к любому итератору дает итератор, который указывает на следующий элемент. Точно так же оператор * дает элемент, на который указывает итератор. Фактически любой объект, который подчиняется нескольким простым правилам, подобным этим, является итератором. *Iterator* — это концепт (§7.2, §12.7). Кроме того, пользователям редко нужно знать тип конкретного итератора; каждый контейнер "знает" типы своих итераторов и делает их доступными под обычными именами `iterator` и `const_iterator`. Например, `list<Entry>::iterator` — это общий тип итератора для списка `list<Entry>`. Нам редко приходится беспокоиться о деталях определения этого типа.

12.4. Итераторы потоков

Итераторы представляют собой общую полезную концепцию для обработки последовательностей элементов в контейнерах. Однако контейнеры — это не единственное место, где мы находим последовательности элементов. На-

пример, входной поток создает последовательность значений, и мы записываем последовательность значений в выходной поток. Следовательно, понятие итераторов может быть с пользой применено для ввода и вывода.

Чтобы создать ostream_iterator, нужно указать, какой поток будет использоваться, и тип объектов, которые будут в него записываться. Например:

```
ostream_iterator<string> oo {cout}; // Запись строк в cout
```

Результатом присваивания значения *oo является запись присваиваемого значения в cout. Например:

```
int main()
{
    *oo = "Hello, ";  // Означает cout<<"Hello, "
    ++oo;
    *oo = "world!\n"; // Означает cout<<"world!\n"
}
```

Это еще один способ записи канонического сообщения в стандартный вывод. Запись ++oo представляет собой имитацию записи в массив через указатель.

Аналогично istream_iterator позволяет рассматривать входной поток как контейнер, предназначенный только для чтения. Здесь мы также должны указать используемый поток и тип ожидаемых значений:

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

Входные итераторы используются в паре, представляющей последовательность, так что мы должны предоставить istream_iterator для указания конца ввода. Это делает istream_iterator по умолчанию:

```
istream_iterator<string> eos {};
```

Как правило, istream_iterator и ostream_iterator не используются непосредственно. Вместо этого они передаются алгоритмам в качестве аргументов. Например, мы можем написать простую программу, которая читает файл, сортирует прочитанные слова, удаляет дубликаты и записывает результат в другой файл:

```
int main()
{
    string from, to;
    // Получение имен входного и выходного файлов:
    cin >> from >> to;

    ifstream is {from};                  // Поток ввода из файла from
    istream_iterator<string> ii {is};    // Входной итератор
    istream_iterator<string> eos {};     // Конец ввода
```

```
    ofstream os {to};                       // Поток вывода в файл to
    ostream_iterator<string> oo{os,"\n"}; // Выходной итератор

    // b – вектор, инициализированный входными данными:
    vector<string> b {ii,eos};
    sort(b.begin(),b.end());                // Сортировка буфера

    // Копирование буфера в выходной поток
    // с отбрасыванием повторяющихся значений:
    unique_copy(b.begin(),b.end(),oo);

    // Возврат состояния ошибки (§1.2.1, §10.4):
    return !is.eof() || !os;
}
```

`ifstream` — это поток `istream`, который может быть присоединен к файлу, а `ofstream` — это поток `ostream`, который может быть присоединен к файлу (§10.7). Второй аргумент `ostream_iterator` используется для разграничения выходных значений.

На самом деле эта программа длиннее, чем она должна быть. Мы читаем строки в `vector`, затем выполняем `sort()`, а затем записываем их, исключая дубликаты. Более элегантное решение заключается в том, чтобы вообще не хранить дубликаты. Это можно сделать, сохранив строки в множество, которое не сохраняет дубликаты и хранит свои элементы в упорядоченном виде (§11.4). Таким образом, мы могли бы заменить две строки, использующие вектор, одной, использующей множество, и заменить `unique_copy()` более простой функцией `copy()`:

```
set<string> b {ii,eos};     // Сбор строк из входного файла
copy(b.begin(),b.end(),oo); // Вывод строк в выходной файл
```

Мы используем имена `ii`, `eos` и `oo` по одному разу, так что программу можно сократить еще сильнее:

```
int main()
{
    string from, to;
    // Получение имен входного и выходного файлов:
    cin >> from >> to;

    ifstream is {from}; // Входной поток для файла from
    ofstream os {to};   // Выходной поток для файла to

    // Чтение входных данных:
    set<string> b {istream_iterator<string>{is},
                   istream_iterator<string>{}};
    // Копирование данных в выходной поток:
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});
```

```
    // Возврат состояния ошибки (§1.2.1, §10.4):
    return !is.eof() || !os;
}
```

Улучшает ли это последнее упрощение удобочитаемость — вопрос вкуса и опыта.

12.5. Предикаты

В приводимых до настоящего времени примерах алгоритмы просто "встроены" в действие, которое необходимо выполнить для каждого элемента последовательности. Однако зачастую мы хотим сделать это действие параметром алгоритма. Например, алгоритм `find` (§12.2, §12.6) обеспечивает удобный способ поиска определенного значения. Более общий вариант ищет элемент, для которого выполняется указанное требование — *предикат*. Например, мы можем захотеть выполнить поиск в `map` первого значения, большего 42. Отображение позволяет получить доступ к своим элементам как к последовательности пар (*ключ,значение*), чтобы мы могли выполнить поиск в `map<string,int>` пары `pair<const string,int>`, где `int` больше `42`:

```
void f(map<string,int>& m)
{
    auto p = find_if(m.begin(),m.end(),Greater_than{42});
    // ...
}
```

Здесь `Greater_than` — функциональный объект (§6.3.2), хранящий значение (`42`), с которым выполняется сравнение:

```
struct Greater_than
{
    int val;
    Greater_than(int v) : val{v} { }
    bool operator()(const pair<string,int>& r) const
    {
        return r.second > val;
    }
};
```

В качестве альтернативы можно воспользоваться лямбда-выражением (§6.3.2):

```
auto p = find_if(m.begin(), m.end(),
             [](const pair<string,int>& r){return r.second>42;});
```

Предикат не должен модифицировать элементы, к которым применяется.

12.6. Обзор алгоритмов

Общее определение алгоритма — "*набор конечного числа правил, задающих последовательность выполнения операций для решения задачи определенного типа ... [и] имеет пять важных особенностей: Конечность ... Определенность ... Ввод ... Вывод ... Эффективность*" [31, §1.1]. В контексте стандартной библиотеки C++ алгоритм является шаблоном функции, работающим с последовательностями элементов.

Стандартная библиотека предоставляет десятки алгоритмов. Алгоритмы определены в пространстве имен `std` и представлены в заголовочном файле `<algorithm>`. Эти алгоритмы стандартной библиотеки принимают в качестве входных данных последовательности. Полуоткрытая последовательность от `b` до `e` записывается как `[b:e)`. Вот несколько примеров.

Избранные стандартные алгоритмы	
`f=for_each(b,e,f)`	Для каждого элемента `x` в `[b:e)` выполнить `f(x)`
`p=find(b,e,x)`	`p` — первый элемент в `[b:e)`, такой, что `*p==x`
`p=find_if(b,e,f)`	`p` — первый элемент в `[b:e)`, такой, что `f(*p)`
`n=count(b,e,x)`	`n` — количество элементов `*q` в `[b:e)`, таких, что `*q==x`
`n=count_if(b,e,f)`	`n` — количество элементов `*q` в `[b:e)`, таких, что `f(*q)`
`replace(b,e,v,v2)`	Замена элементов `*q` в `[b:e)`, таких, что `*q==v`, на `v2`
`replace_if(b,e,f,v2)`	Замена элементов `*q` в `[b:e)`, таких, что `f(*q)`, на `v2`
`p=copy(b,e,out)`	Копирование `[b:e)` в `[out:p)`
`p=copy_if(b,e,out,f)`	Копирование элементов `*q` из `[b:e)`, таких, что `f(*q)`, в `[out:p)`
`p=move(b,e,out)`	Перемещение `[b:e)` в `[out:p)`
`p=unique_copy(b,e,out)`	Копирование `[b:e)` в `[out:p)`; соседние дубликаты не копируются
`sort(b,e)`	Сортировка элементов `[b:e)` с использованием `<` в качестве критерия сортировки
`sort(b,e,f)`	Сортировка элементов `[b:e)` с использованием `f` в качестве критерия сортировки
`(p1,p2)=equal_range(b,e,v)`	`[p1:p2)` является подпоследовательностью отсортированной последовательности `[b:e)` со значением `v`; по сути, бинарный поиск `v`
`p=merge(b,e,b2,e2,out)`	Слияние двух отсортированных последовательностей, `[b:e)` и `[b2:e2)`, в `[out:p)`
`p=merge(b,e,b2,e2,out,f)`	Слияние двух отсортированных последовательностей, `[b:e)` и `[b2:e2)`, в `[out:p)` с использованием `f` для сравнения

Эти и многие другие алгоритмы (например, §14.3) могут применяться к элементам контейнеров, строк и встроенных массивов.

Некоторые алгоритмы, такие как `replace()` и `sort()`, изменяют значения элементов, но ни один алгоритм не добавляет или не удаляет элементы контейнера. Причина в том, что последовательность не идентифицирует контейнер, который содержит ее элементы. Чтобы добавить или удалить элементы, вам нужно что-то, знающее о конкретном контейнере (например, `back_inserter`; §12.1), или непосредственное обращение к контейнеру (например, `push_back()` или `erase()`; §11.2).

Для операций, передаваемых в качестве аргументов, очень часто применяются лямбда-выражения. Например:

```
vector<int> v = {0,1,2,3,4,5};
for_each(v.begin(),v.end(),[](int&x){x=x*x;}); // v=={0,1,4,9,16,25}
```

Алгоритмы стандартной библиотеки, как правило, более тщательно разработаны, специфицированы и реализованы, чем средний созданный вручную цикл, поэтому их нужно знать и использовать, предпочитая коду, написанному на "голом" языке.

12.7. Концепты (C++20)

В конце концов алгоритмы стандартной библиотеки будут определены с использованием концептов (глава 7, "Концепты и обобщенное программирование"). Предварительные обсуждения этого вопроса можно найти в Ranges Technical Specification [37], а реализации можно найти в Интернете. Концепты определены в `<experimental/Ranges>`, но, надеюсь, что-то очень похожее будет добавлено в пространство имен `std` в C++20.

Диапазоны `Range` представляют собой обобщение последовательностей C++98, определяемых парами `begin()`/`end()`. Диапазон — это понятие, определяющее, что может представлять собой последовательность элементов. Его можно определить как

- пару итераторов `{begin,end}`;
- пару `{begin,n}`, где `begin` представляет собой итератор, а `n` — количество элементов;
- пару `{begin,pred}`, где `begin` представляет собой итератор, а `pred` — предикат; если `pred(p)` возвращает `true` для итератора `p`, мы достигли конца последовательности. Это позволяет нам иметь бесконечные последовательности и последовательности, генерируемые "на лету".

Этот концепт Range позволяет писать sort(v), а не sort(v.begin(), v.end()), как нам приходится работать с STL с 1994 года. Например:

```
template<BoundedRange R>
requires Sortable<R>
void sort(R& r)
{
    return sort(begin(r),end(r));
}
```

Отношением для Sortable по умолчанию является less.

В дополнение к Range спецификация Ranges TS предлагает множество полезных концептов. Эти концепты находятся в <experimental/Ranges/concepts>. Точные их определения можно найти в [37].

Фундаментальные концепты языка	
Same<T,U>	T представляет собой тот же тип, что и U
DerivedFrom<T,U>	T является производным от U
ConvertibleTo<T,U>	T может быть преобразован в U
CommonReference<T,U>	T и U совместно используют общий ссылочный тип
Common<T,U>	T и U совместно используют общий тип
Integral<T>	T является целочисленным типом
SignedIntegral<T>	T является знаковым целочисленным типом
UnsignedIntegral<T>	T является беззнаковым целочисленным типом
Assignable<T,U>	U может быть присвоен T
SwappableWith<T,U>	T может быть обменен с U
Swappable<T>	SwappableWith<T,T>

Common важен для указания алгоритмов, которые должны работать с различными связанными типами, оставаясь надежным с математической точки зрения. Common<T,U> — это тип C, который можно использовать для сравнения T с U, сначала преобразуя оба значения в тип C. Например, мы можем захотеть сравнить std::string со строкой в стиле C (char*) или значения int и double, но не std::string с int. Для этого соответствующим образом специализируется common_type_t, используемый в определении Common:

```
using common_type_t<std::string,char*> = std::string;
using common_type_t<double ,int> = double;
```

Определение Common немного сложнее, но решает сложную фундаментальную проблему. К счастью, нам не нужно определять специализацию common_type_t, если только мы не хотим использовать операции над сме-

шанными типами, для которых библиотека (пока еще) не имеет подходящих определений. `Common` или `CommonReference` используется в определениях большинства концептов и алгоритмов, которые могут сравнивать значения разных типов.

На концепты, связанные со сравнением, сильно повлияла книга [40].

Концепты сравнений	
`Boolean<T>`	`T` может использоваться как булева величина
`WeaklyEqualityComparableWith<T,U>`	`T` и `U` могут сравниваться на равенство с использованием операторов `==` и `!=`
`WeaklyEqualityComparable<T>`	`WeaklyEqualityComparableWith<T,T>`
`EqualityComparableWith<T,U>`	`T` и `U` могут сравниваться на равенство с использованием оператора `==`
`EqualityComparable<T>`	`EqualityComparableWith<T,T>`
`StrictTotallyOrderedWith<T,U>`	`T` и `U` могут сравниваться с использованием операторов `<`, `<=`, `>` и `>=`, дающих полное упорядочение
`StrictTotallyOrdered<T>`	`StrictTotallyOrderedWith<T,T>`

Применение как `WeaklyEqualityComparableWith`, так и `WeaklyEqualityComparable` демонстрирует отсутствие (до настоящего времени) возможности перегрузки.

Концепты, связанные с объектами	
`Destructible<T>`	`T` может быть уничтожен, а его адрес может быть получен с помощью унарного оператора `&`
`Constructible<T,Args>`	`T` может быть построен из списка аргументов типа `Args`
`DefaultConstructible<T>`	`T` может быть создан с помощью конструктора по умолчанию
`MoveConstructible<T>`	`T` может быть создан с помощью перемещающего конструктора
`CopyConstructible<T>`	`T` может быть создан с помощью копирующего и перемещающего конструкторов
`Movable<T>`	`MoveConstructable<T>`, `Assignable<T&,T>` и `Swapable<T>`
`Copyable<T>`	`CopyConstructable<T>`, `Moveable<T>` и `Assignable<T, const T&>`
`Semiregular<T>`	`Copyable<T>` и `DefaultConstructable<T>`
`Regular<T>`	`SemiRegular<T>` и `EqualityComparable<T>`

`Regular` — идеал для типов. Тип, соответствующий концепту `Regular`, грубо говоря, работает как `int` и упрощает большую часть наших размышлений о том, как использовать этот тип (§7.2). Отсутствие по умолчанию оператора `==` для классов означает, что большинство классов относятся к `SemiRegular`, хотя большинство из них могут и должны быть `Regular`.

Концепты, связанные с вызовами	
`Invocable<F,Args>`	`F` может быть вызван со списком аргументов типа `Args`
`InvocableRegular<F,Args>`	`Invocable<F,Args>` и сохраняет равенство
`Predicate<F,Args>`	`F` может быть вызван со списком аргументов типа `Args` и возвращает `bool`
`Relation<F,T,U>`	`Predicate<F,T,U>`
`StrictWeakOrder<F,T,U>`	`Relation<F,T,U>`, предоставляющий строгое слабое упорядочение

Функция `f()` называется *сохраняющей равенство* (equality preserving), если из `x==y` следует, что `f(x)==f(y)`.

Строгое слабое упорядочение (strict weak ordering) — это то, что стандартная библиотека обычно предполагает для сравнений, таких как `<`; поищите соответствующую информацию в учебниках или Интернете, если вас заинтересовал этот вопрос.

`Relation` и `StrictWeakOrder` различаются только семантикой. Мы не можем (в настоящее время) представить это различие в коде, поэтому просто выражаем наши намерения с помощью имен.

Концепты, связанные с итераторами	
`Iterator<I>`	К `I` могут быть применены операторы инкремента (`++`) и разыменования (`*`)
`Sentinel<S,I>`	`S` является ограничителем для типа итератора, т.е. `S` является предикатом над значением типа `I`
`SizedSentinel<S,I>`	Ограничитель `S`, где к `I` может быть применен оператор `-`
`InputIterator<I>`	`I` — входной итератор; оператор `*` может быть применен только для чтения
`OutputIterator<I>`	`I` — выходной итератор; оператор `*` может быть применен только для записи
`ForwardIterator<I>`	`I` — однонаправленный итератор, поддерживающий многопроходность
`BidirectionalIterator<I>`	`I` — `ForwardIterator` с поддержкой оператора `--`

Окончание табл.

Концепты, связанные с итераторами	
`RandomAccessIterator<I>`	`I` — `BidirectionalIterator` с поддержкой операторов `+`, `-`, `+=`, `-=` и `[]`
`Permutable<I>`	`I` — `ForwardIterator<I>`, где `I` разрешает перемещать и обменивать элементы
`Mergeable<I1,I2,R,O>`	Можно сливать отсортированные последовательности, определяемые `I1` и `I2`, в `O` с использованием `Relation<R>`
`Sortable<I>`	Можно сортировать последовательности, определяемые `I`, с использованием отношения меньше
`Sortable<I,R>`	Можно сортировать последовательности, определяемые `I`, с использованием `Relation<R>`

Различные разновидности (категории) итераторов используются для выбора наилучшей реализации для данного алгоритма; см. §7.2.2 и §13.9.1. Пример `InputIterator` см. в §12.4.

Основная идея ограничителя состоит в том, что мы можем перебирать диапазон, начиная с определенного итератора, пока для элемента не станет истинным указанный предикат. Таким образом, итератор `p` и ограничитель `s` определяют диапазон [`p:s(*p)`). Например, мы могли бы определить предикат для ограничения для обхода строки в стиле C, используя в качестве итератора указатель:

```
[](const char* p) {return *p==0; }
```

Изложение информации о `Mergeable` и `Sortable` в данной книге упрощено по сравнению с [37].

Концепты диапазонов	
`Range<R>`	`R` представляет собой диапазон с начальным итератором и ограничителем
`SizedRange<R>`	`R` представляет собой диапазон с получением размера за константное время
`View<R>`	`R` представляет собой диапазон с копированием, перемещением и прием за константное время
`BoundedRange<R>`	`R` представляет собой диапазон с идентичными типами итератора и ограничителя
`InputRange<R>`	`R` представляет собой диапазон, тип итератора которого удовлетворяет концепту `InputIterator`

Окончание табл.

Концепты диапазонов	
`OutputRange<R>`	`R` представляет собой диапазон, тип итератора которого удовлетворяет концепту `OutputIterator`
`ForwardRange<R>`	`R` представляет собой диапазон, тип итератора которого удовлетворяет концепту `ForwardIterator`
`BidirectionalRange<R>`	`R` представляет собой диапазон, тип итератора которого удовлетворяет концепту `BidirectionalIterator`
`RandomAccessRange<R>`	`R` представляет собой диапазон, тип итератора которого удовлетворяет концепту `RandomAccessIterator`

В [37] имеются и другие концепты, но для начала вполне достаточно и этих.

12.8. Алгоритмы над контейнерами

Для того чтобы не ожидать официального принятия диапазонов, можно определить собственные простые алгоритмы для диапазонов. Например, можно легко предоставить сокращение, чтобы писать просто `sort(v)` вместо `sort(v.begin(),v.end())`:

```
namespace Estd {
    using namespace std;

    template<typename C>
    void sort(C& c)
    {
        sort(c.begin(),c.end());
    }

    template<typename C, typename Pred>
    void sort(C& c, Pred p)
    {
        sort(c.begin(),c.end(),p);
    }
    // ...
}
```

Я поместил контейнерные версии `sort()` (и других алгоритмов) в их собственное пространство имен `Estd` ("extended `std`" — расширенный `std`), чтобы не мешать другим пользователям применять пространство имен `std`, а также чтобы затем было проще заменить эту "затычку" для концепта `Range` использованием настоящего концепта.

12.9. Параллельные алгоритмы

Если одна и та же задача должна быть выполнена для многих элементов данных, можно выполнять ее параллельно для каждого элемента данных при условии, что вычисления для различных элементов данных независимы:

- *параллельное выполнение*: задания выполняются несколькими потоками выполнения (зачастую работающими на разных ядрах процессора);
- *векторизованное выполнение*: задания выполняются на одном потоке с использованием векторизации, известной как *SIMD* ("Single Instruction, Multiple Data" — одна команда, много данных).

Стандартная библиотека предлагает поддержку для обоих вариантов, и мы можем точно указать требование последовательного выполнения; в заголовочном файле `<execution>` имеются следующие индикаторы стратегий:

- `seq`: последовательное выполнение;
- `par`: параллельное выполнение (если таковое возможно);
- `par_unseq`: параллельное и/или непоследовательное (векторизованное) выполнение (если таковое возможно).

Рассмотрим `std::sort()`:

```
sort(v.begin(),v.end());      // Последовательное
sort(seq,v.begin(),v.end()); // Последовательное (как и по умолчанию)
sort(par,v.begin(),v.end()); // Параллельное
// Параллельное и/или векторизованное:
sort(par_unseq,v.begin(),v.end());
```

Стоит ли распараллеливать и/или векторизовать выполнение, зависит от алгоритма, количества элементов в последовательности, аппаратного обеспечения и использования этого аппаратного обеспечения программами, работающими на нем. Следовательно, *индикаторы стратегии выполнения* являются просто подсказками. Компилятор и/или планировщик времени выполнения решат, какой параллелизм использовать. Это все очень нетривиальные задачи, и очень важно не делать никаких утверждений об эффективности применения параллельности без проведения детальных измерений.

Для большинства алгоритмов стандартной библиотеки, включая все таблицы в §12.6, за исключением `equal_Range`, может быть запрошено распараллеливание и векторизация с использованием `par` и `par_unseq`, как в случае `sort()`. Почему не `equal_Range()`? Потому что до сих пор никто не придумал для него достойного параллельного алгоритма.

Многие параллельные алгоритмы используются в основном для числовых данных; см. §14.3.1.

При запросе параллельного выполнения следует убедиться в отсутствии возможности гонки данных (§15.2) и взаимоблокировки (§15.5).

12.10. Советы

[1] Алгоритмы STL работают с одной или несколькими последовательностями; §12.1.

[2] Входная последовательность является полуоткрытой и определяется парой итераторов; §12.1.

[3] Алгоритм поиска обычно возвращает конец входной последовательности как указатель неудачности выполненного поиска; §12.2.

[4] Алгоритмы непосредственно не добавляют и не убирают элементы из последовательностей, переданных им в качестве аргументов; §12.2, §12.6.

[5] При написании цикла подумайте, не может ли он быть выражен как общий алгоритм; §12.2.

[6] Используйте предикаты и другие функциональные объекты для придания стандартным алгоритмам более широкого диапазона смыслов; §12.5, §12.6.

[7] Предикат не должен модифицировать свой аргумент; §12.5.

[8] Следует знать алгоритмы стандартной библиотеки и предпочитать их написанным вручную циклам; §12.6.

[9] Если написание пары итераторов становится утомляющим, вводите алгоритмы для контейнеров/диапазонов; §12.8.

13

Утилиты

Время, которое вы тратите
с удовольствием,
не тратится впустую.

— Бертран Рассел

- **Введение**
- **Управление ресурсами**
  - `unique_ptr` и `shared_ptr`
  - `move()` и `forward()`
- **Проверка выхода за границы диапазона: gsl::span**
- **Специализированные контейнеры**
  - `array`
  - `bitset`
  - `pair` и `tuple`
- **Альтернативы**
  - `variant`
  - `optional`
  - `any`
- **Аллокаторы**
- **Время**
- **Адаптация функций**
  - **Лямбда-выражения в качестве адаптеров**
  - `mem_fn()`
  - `function`
- **Функции типов**
  - `iterator_traits`
  - **Предикаты типов**
  - `enable_if`
- **Советы**

13.1. Введение

Не все компоненты стандартной библиотеки входят в состав средств с четко помеченным функциональным предназначением, таким как "контейнеры" или "ввод-вывод". В этом разделе приводится несколько примеров небольших широко используемых компонентов. Такие компоненты (классы и шаблоны) часто называют *словарными типами* (vocabulary types), потому что они являются частью общего словаря, который мы используем для описания наших проектов и программ. Такие библиотечные компоненты часто выступают в качестве строительных блоков для более мощных библиотечных средств, включая другие компоненты стандартной библиотеки. Чтобы быть полезными, функция или тип не обязаны быть сложными или тесно связанными с массой других функций и типов.

13.2. Управление ресурсами

Одной из ключевых задач любой нетривиальной программы является управление ресурсами. Ресурс — это то, что должно быть захвачено, а позже (явно или неявно) освобождено. Примерами являются память, блокировки, сокеты, дескрипторы потоков выполнения и файловые дескрипторы. Для длительно работающей программы неспособность своевременно освободить ресурс ("утечка") может привести к серьезному снижению производительности и, возможно, даже к аварийному завершению программы. Даже для коротких программ утечка может стать помехой, например из-за нехватки ресурсов, увеличивающей время выполнения на порядки.

Компоненты стандартной библиотеки разработаны так, чтобы не допускать утечки ресурсов. Для этого они полагаются на базовую языковую поддержку управления ресурсами с использованием пар "конструктор/деструктор", чтобы гарантировать, что ресурс не переживет объект, ответственный за него. Примером является использование пары "конструктор/деструктор" в `Vector` для управления временем жизни его элементов (§4.2.2), и все контейнеры стандартной библиотеки реализованы аналогичным образом. Важно отметить, что этот подход корректно взаимодействует с обработкой ошибок с использованием исключений. Например, этот метод используется для классов блокировки стандартной библиотеки:

```
mutex m;                        // Используется для защиты доступа
                                // к совместно используемым данным
// ...

void f()
{
```

```
    scoped_lock<mutex> lck {m}; // Захват мьютекса m
    // ... работа с совместно используемыми данными ...
}
```

Поток `thread` не будет работать до тех пор, пока конструктор `lck` не захватит `mutex` (§15.5). Соответствующий деструктор освободит ресурсы. То есть в данном примере деструктор `scoped_lock` освободит `mutex`, когда поток управления покинет `f()` (с помощью `return`, путем "попадания в конец функции" или генерации исключения).

Это применение идиомы RAII ("захват ресурса есть инициализация", §4.2.2). RAII имеет фундаментальное значение для идиоматической обработки ресурсов в C++. Контейнеры (такие, как `vector` и `map`, `string` и `iostream`) управляют своими ресурсами (такими, как дескрипторы файлов и буфера) аналогичным образом.

13.2.1. `unique_ptr` и `shared_ptr`

До сих пор рассматривавшиеся примеры заботились об объектах, определенных в области видимости, освобождая ресурсы, которые они захватывают, при выходе из области видимости. Но как насчет объектов, размещенных в динамической памяти? В заголовочном файле `<memory>` стандартная библиотека предоставляет два "интеллектуальных указателя", которые помогают управлять объектами в динамической памяти:

[1] `unique_ptr` для представления единственного владения;

[2] `shared_ptr` для представления совместного владения.

Основное использование "интеллектуальных указателей" — предотвращение утечек памяти, вызванных небрежным программированием. Например:

```
void f(int i, int j)          // X* и unique_ptr<X>
{
    X* p = new X;             // Выделение памяти new X
    unique_ptr<X> sp {new X}; // Выделение памяти new X и передача
                              // указателя в unique_ptr
    // ...
    if (i < 99) throw Z{};    // Может генерировать исключение
    if (j < 77) return;       // "Ранний" выход из функции

    // ... использование p и sp ..

    delete p;                 // Уничтожение *p
}
```

Здесь мы "забыли" удалить `p` при `i<99` и `j<77`. `unique_ptr` же гарантирует, что его объект будет должным образом уничтожен независимо от того, как осуществляется выход из функции `f()` (с помощью исключения, выполнения `return` или достижения конца тела функции). По иронии судьбы мы

могли бы решить проблему, просто *не* используя указатель и *не* используя new:

```
void f(int i, int j) // Использование локальной переменной
{
    X x;
    // ...
}
```

К сожалению, злоупотребление оператором new (а также указателями и ссылками), похоже, становится все более и более серьезной проблемой.

Однако, когда вам действительно нужна семантика указателей, unique_ptr оказывается очень легким механизмом без накладных расходов памяти и времени по сравнению с правильным применением встроенного указателя. Его применение включает передачу объектов из динамической памяти в функции и их возврат из них:

```
// Создает X и передает его в unique_ptr:
unique_ptr<X> make_X(int i)
{
    // ... проверка i и прочие действия ...
    return unique_ptr<X>{new X{i}};
}
```

unique_ptr представляет собой дескриптор отдельного объекта (или массива) почти так же, как vector представляет собой дескриптор последовательности объектов. Оба они управляют временем жизни других объектов (используя идиому RAII) и оба используют семантику перемещения, чтобы сделать возврат с помощью return простым и эффективным.

shared_ptr похож на unique_ptr, с тем отличием, что shared_ptr копируются, а не перемещаются. Интеллектуальные указатели shared_ptr для объекта совместно используют владение этим объектом; объект уничтожается, когда уничтожается последний из указывающих на него shared_ptr. Например:

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
    shared_ptr<fstream> fp {new fstream(name ,mode)};
    if (!*fp)    // Убеждаемся, что файл корректно открыт
        throw No_file{};

    f(fp);
    g(fp);
    // ...
}
```

Теперь файл, открытый конструктором fp, будет закрыт последней функцией, которая (явно или неявно) уничтожит копию fp. Обратите внимание, что f() или g() может запускать задание, сохраняющее копию fp, или сохранять копию, которая переживет user(), каким-либо другим способом. Таким образом, shared_ptr предоставляет разновидность сборки мусора, которая учитывает управление ресурсами на основе деструкторов объектов. Этот интеллектуальный указатель не является ни бесплатным, ни чрезмерно дорогостоящим, но он затрудняет прогнозирование времени жизни совместно используемого объекта. Используйте shared_ptr, только если вам действительно нужно совместное владение.

Создание объекта в динамической памяти с последующей передачей его адреса интеллектуальному указателю оказывается немного многословным и чревато ошибками, такими как забытая передача указателя в unique_ptr или передача в shared_ptr указателя на что-то, что находится не в динамической памяти. Чтобы избежать таких проблем, стандартная библиотека (в заголовочном файле <memory>) предоставляет функции для создания объекта и возврата соответствующего интеллектуального указателя — make_shared() и make_unique(). Например:

```
struct S
{
    int i;
    string s;
    double d;
    // ...
};

auto p1 = make_shared<S>(1,"Ankh Morpork",4.65); // p1: shared_ptr<S>
auto p2 = make_unique<S>(2,"Oz",7.62);           // p2: unique_ptr<S>
```

Теперь p2 — это объект unique_ptr<S>, указывающий на объект, выделенный в динамической памяти, с типом S со значением {2,"Oz"s,7.62}.

Использование make_shared() не просто более удобно, чем отдельное создание объекта с использованием new и его последующей передачей в shared_ptr, но и значительно более эффективно, поскольку в этом случае не требуется отдельное выделение памяти для счетчика использований, существенное для реализации shared_ptr.

При наличии unique_ptr и shared_ptr мы можем реализовать политику полностью "без голого new" (§4.2.2) во многих программах. Тем не менее эти "интеллектуальные указатели" концептуально все еще являются указателями, а следовательно, являются лишь моим вторым выбором для управления ресурсами — после контейнеров и других типов, которые управляют ресурсами на более высоком концептуальном уровне. В частности, shared_ptr

сами по себе не предоставляют никаких правил, по которым их владельцы могут читать и/или записывать совместно используемый объект. Простого устранения проблем управления ресурсами недостаточно для решения проблем гонки данных (§15.7) и других вопросов.

Когда мы используем интеллектуальные указатели (например, `unique_ptr`), а не дескрипторы ресурса с операциями, разработанными специально для данного ресурса (такие, как `vector` или `thread`)? Неудивительно, что ответ звучит как "Когда нам нужна семантика указателя".

- Когда мы совместно используем объект, нам нужны указатели (или ссылки) для обращения к этому объекту, поэтому очевидным выбором становится `shared_ptr` (если, конечно, у объекта нет очевидного единственного владельца).
- Когда мы ссылаемся на полиморфный объект в классическом объектно-ориентированном коде (§4.5), нам нужен указатель (или ссылка), потому что мы не знаем точный тип объекта, на который ссылаемся (и даже его размер), поэтому очевидным выбором становится `unique_ptr`.
- Для совместно используемого полиморфного объекта обычно требуется `shared_ptr`.

Нам *не* нужно использовать указатель для возврата коллекции объектов из функции; контейнер, который является дескриптором ресурса, сделает это просто и эффективно (§5.2.2).

13.2.2. `move()` и `forward()`

Выбор между перемещением и копированием в основном выполняется неявно (§3.6). Компилятор предпочитает перемещение, когда объект должен быть уничтожен (как в случае возврата из функции с помощью `return`), потому что предполагается, что это более простая и эффективная операция. Однако иногда нам приходится указывать свой выбор явно. Например, `unique_ptr` является единственным владельцем объекта. Следовательно, его нельзя скопировать:

```
void f1()
{
    auto p = make_unique<int>(2);
    auto q = p;   // Ошибка: нельзя копировать unique_ptr
    // ...
}
```

Если вы все равно хотите использовать `unique_ptr`, вы должны его перемещать. Например:

```
void f1()
{
    auto p = make_unique<int>(2);
    auto q = move(p);   // В p теперь хранится nullptr
    // ...
}
```

Вообще-то, `std::move()` ничего не перемещает. Вместо этого данная функция переводит свой аргумент в ссылку на r-значение, тем самым говоря, что его аргумент больше не будет использоваться и, следовательно, может быть перемещен (§5.2.2). Эта возможность должна была быть названа как-то вроде `rvalue_cast`. Как и другие приведения, оно подвержено ошибкам и его лучше избегать. Оно существует только для обслуживания нескольких важных случаев. Рассмотрим простой обмен:

```
template <typename T>
void swap(T& a, T& b)
{
    T tmp {move(a)}; // Конструктор T видит r-значение: перемещение
    a = move(b);     // Присваивание T видит r-значение: перемещение
    b = move(tmp);   // Присваивание T видит r-значение: перемещение
}
```

Мы не хотим постоянно копировать потенциально большие объекты и потому запрашиваем перемещение с помощью `std::move()`.

Как и в случае других приведений, имеются заманчивые, но опасные применения `std::move()`. Рассмотрим следующий фрагмент кода:

```
string s1 = "Hello";
string s2 = "World";
vector<string> v;
v.push_back(s1);        // Используется аргумент "const string&";
                        // push_back() выполняет копирование
v.push_back(move(s2)); // Использование перемещения
```

Здесь `s1` копируется (функцией `push_back()`), в то время как `s2` перемещается. Это иногда (только иногда) делает `push_back()` для `s2` более дешевым. Проблема в том, что при этом остается объект, из которого выполнено перемещение. Если мы вновь воспользуемся `s2`, то получим проблемы:

```
cout << s1[2];   // Вывод 'l'
cout << s2[2];   // Аварийное завершение?
```

Я считаю, что широкое использование `std::move()` слишком чревато ошибками. Не используйте его, если не можете продемонстрировать существенное и необходимое улучшение производительности. При дальнейшей технической поддержке и внесении обновлений в код перемещенный объект может быть случайно использован.

Состояние удаленного объекта в общем случае не определено, но все типы стандартной библиотеки оставляют перемещенный объект в состоянии, в котором он может быть уничтожен и присвоен. Было бы неразумно не следовать этому примеру. Для контейнера (например, вектора или строки) состояние после перемещения будет "пустым контейнером". Для многих типов пустое состояние представляет собой значение по умолчанию: оно имеет смысл и дешево устанавливается.

Передача аргументов является важным вариантом использования, который требует перемещения (§7.4.2). Иногда мы хотим передать набор аргументов другой функции, ничего не меняя (для достижения "прямой передачи"):

```
template<typename T, typename ... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    // Передача каждого аргумента:
    return unique_ptr<T>{new T{std::forward<Args>(args)...}};
}
```

Функция `forward()` стандартной библиотеки отличается от более простой `std::move()`, правильно обрабатывая тонкости, связанные с l- и r-значениями (§5.2.2). Используйте `std::forward()` исключительно для передачи и не передавайте что-либо дважды; как только вы передали объект, вы больше не можете его использовать.

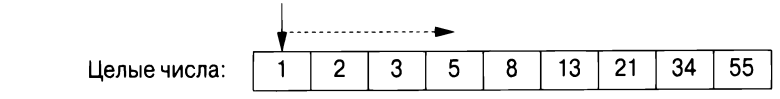
13.3. Проверка выхода за границы диапазона: `gsl::span`

Традиционно ошибки выхода за границы диапазона были основным источником серьезных ошибок в программах на C и C++. Использование контейнеров (глава 11, "Контейнеры"), алгоритмов (глава 12, "Алгоритмы") и цикла `for` по диапазону значительно уменьшили эту проблему, но можно сделать больше. Основным источником ошибок выхода за границы диапазона является то, что люди передают указатели (обычные или интеллектуальные), а затем, чтобы узнать количество элементов, на которые они указывают, полагаются на соглашение. Наилучший совет для кода вне дескрипторов ресурсов состоит в том, чтобы предположить, что указатель указывает не более чем на один объект [CG:F.22], но без соответствующей поддержки этот совет мало что дает. Нам может помочь `string_view` (§9.3) из стандартной библиотеки, но это представление доступно только для чтения и только для символов. Большинству программистов нужно большее.

В работе [61] предлагаются некоторые рекомендации и небольшая библиотека для их поддержки [23], включая тип `span` для ссылки на диапазон эле-

ментов. Этот span предложен в стандарт, но пока что это просто код, который вы можете загрузить, если он вам нужен.

string_span представляет собой пару (указатель, длина), обозначающую последовательность элементов.



span предоставляет доступ к непрерывной последовательности элементов. Элементы могут храниться разными способами, в том числе в векторах и встроенных массивах. Как и указатель, span не владеет символами, на которые указывает. В этом он похож на string_view (§9.3) и на пару итераторов STL (§12.3).

Рассмотрим общий стиль интерфейса:

```
void fpn(int* p, int n)
{
    for (int i = 0; i<n; ++i)
        p[i] = 0;
}
```

Мы предполагаем, что p указывает на n целых чисел. К сожалению, это предположение — просто соглашение, поэтому мы не можем использовать его для написания цикла for для диапазона, а компилятор не может реализовать дешевую и эффективную проверку выхода за границы диапазона. Кроме того, наше предположение может попросту быть неверным:

```
void use(int x)
{
    int a[100];
    fpn(a,100);     // OK
    fpn(a,1000);    // Просто опечатка! (Ошибка диапазона в fpn)
    fpn(a+10,100); // Ошибка диапазона в fpn
    fpn(a,x);       // Выглядит невинно, но...
}
```

Можно сделать лучше, если использовать span:

```
void fs(span<int> p)
{
    for (int x : p)
        x = 0;
}
```

Использовать fs можно, например, так:

```
void use(int x)
{
    int a[100];
    fs(a);          // Неявное создание span<int>{a,100}
    fs(a,1000);     // Ошибка :ожидается span
    fs({a+10,100}); // Ошибка диапазона в fs
    fs({a,x});      // Очевидно подозрительно
}
```

То есть распространенный случай создания `span` непосредственно из массива теперь безопасен (компилятор сам вычисляет количество элементов) и прост для записи. В других случаях вероятность ошибок снижается, потому что программист должен явно составить диапазон.

Распространенный случай, когда диапазон передается от функции к функции, оказывается проще, чем для интерфейсов (указатель,счетчик), и, очевидно, не требует дополнительной проверки:

```
void f1(span<int> p);
void f2(span<int> p)
{
    // ...
    f1(p);
}
```

При использовании для индексации (например, `r[i]`) выполняется проверка выхода за границы диапазона, и в случае ошибки генерируется исключение `gsl::fail_fast`. Проверка выхода за границы диапазона может быть подавлена для кода, критичного к производительности. Когда `span` станет стандартом, я ожидаю, что `std::span` будет использовать контракты [20] для управления ответами на нарушения границ диапазона.

Обратите внимание, что для цикла необходима только одна проверка диапазона. Таким образом, для распространенного случая, когда тело функции, использующей `span`, представляет собой цикл по диапазону `span`, эта проверка оказывается практически бесплатной.

Диапазон символов поддерживается непосредственно и называется `gsl::string_span`.

13.4. Специализированные контейнеры

Стандартная библиотека предоставляет несколько контейнеров, которые не вписываются идеально в каркас STL (глава 11, "Контейнеры", глава 12, "Алгоритмы"). Примерами являются встроенные массивы, `array` и `string`. Я иногда называю их "почти контейнерами", но это не совсем справедливо: они содержат элементы, поэтому являются контейнерами, но у каждого есть ограничения или дополнительные возможности, которые делают их выпада-

ющими из контекста STL. Кроме того, их отдельное описание упрощает описание STL.

"Почти контейнеры"	
`T[N]`	Встроенный массив: непрерывная последовательность фиксированного размера из `N` элементов типа `T`; неявно преобразуется в `T*`
`array<T,N>`	Непрерывная последовательность фиксированного размера из `N` элементов типа `T`; подобна встроенному массиву, но с решением большинства проблем
`bitset<N>`	Последовательность фиксированного размера из `N` битов
`vector<bool>`	Последовательность битов, компактно хранящаяся в специализации `vector`
`pair<T,U>`	Два элемента типов `T` и `U`
`tuple<T...>`	Последовательность произвольного количества элементов произвольных типов
`basic_string<C>`	Последовательность символов типа `C`; предоставляет строковые операции
`valarray<T>`	Массив числовых значений типа `T`; предоставляет числовые операции

Почему стандартная библиотека предоставляет так много контейнеров? Они служат распространенным, но различным (хотя и часто перекрывающимся) потребностям. Если бы стандартная библиотека их не предоставляла, многим программистам пришлось бы разрабатывать и реализовывать собственные контейнеры.

- `pair` и `tuple` — гетерогенные; все прочие контейнеры гомогенные (все их элементы одного и того же типа).
- Элементы `array`, `vector` и `tuple` располагаются в памяти непрерывно; `forward_list` и `map` представляют собой связанные структуры.
- `bitset` и `vector<bool>` хранят биты и обращаются к ним через прокси-объекты; все прочие контейнеры стандартной библиотеки могут хранить различные типы и непосредственно обращаться к хранимым элементам.
- `basic_string` требует, чтобы все были некоторой разновидностью символов и обеспечивали возможность работы со строками, как, например, конкатенация или операции, чувствительные к локали.
- `valarray` требует, чтобы все элементы были числами и обеспечивали возможность выполнения числовых операций.

Все эти контейнеры можно рассматривать как предоставляющие специализированные услуги, необходимые для больших сообществ программистов. Ни один контейнер не может удовлетворить все эти потребности одновременно, потому что некоторые из них противоречивы, например "способность увеличиваться" и "гарантированно размещаться в фиксированном местоположении" или "элементы при добавлении не перемещаются" и "элементы располагаются в памяти непрерывно".

13.4.1. array

`array`, определенный в заголовочном файле `<array>`, представляет собой последовательность элементов определенного типа с фиксированным размером, который указывается во время компиляции. Таким образом, массив может быть размещен со своими элементами в стеке, в объекте или в статической памяти. Элементы расположены в области видимости, где определен `array`. Лучше всего воспринимать `array` как встроенный массив с жестко определенным размером, без неявных, потенциально неожиданных преобразований к типам указателей и с несколькими вспомогательными функциями. Нет никаких накладных расходов (времени или памяти), связанных с использованием `array` по сравнению с использованием встроенного массива. `array` *не* следует модели "дескриптора элементов" контейнеров STL. Вместо этого `array` непосредственно содержит свои элементы.

`array` может быть инициализирован с помощью списка инициализации:

```
array<int,3> a1 = {1,2,3};
```

Количество элементов в инициализаторе должно быть равно (или меньше) количеству элементов, указанному в объявлении `array`.

Количество элементов не является необязательным:

```
array<int> ax = {1,2,3}; // Ошибка: не указан размер
```

Количество элементов должно быть константным выражением:

```
void f(int n)
{
    array<string,n> aa = {"John's", "Queens'"}; // Ошибка: размер -
    // ...                                      // не константное выражение
}
```

Если вам нужен контейнер с переменным количеством элементов, используйте `vector`.

При необходимости `array` может быть явно передан в функцию в стиле C, ожидающую указатель. Например:

```
void f(int* p, int sz); // Интерфейс в стиле C
void g()
{
    array<int,10> a;

    f(a,a.size());        // Ошибка: преобразования нет
    f(&a[0],a.size());    // Использование в стиле C
    f(a.data(),a.size()); // Использование в стиле C

    // Использование в стиле C++/STL:
    auto p = find(a.begin(),a.end(),777);

    // ...
}
```

Зачем нам использовать `array`, если `vector` намного более гибкий? `array` менее гибок, поэтому он проще. Иногда существенную роль играет преимущество в производительности, которое достигается благодаря прямому доступу к элементам, выделенным в стеке, вместо выделения элементов в динамической памяти с косвенным обращением через `vector` (дескриптор) и последующим их освобождением. С другой стороны, стек является ограниченным ресурсом (особенно в некоторых встроенных системах), и его переполнение оказывается весьма неприятным.

Зачем использовать `vector`, если можно использовать встроенный массив? `vector` знает свой размер, поэтому его легко использовать с алгоритмами стандартной библиотеки и его можно копировать с помощью оператора =. Однако главная причина, по которой я предпочитаю массив, состоит в том, что он избавляет меня от неожиданных и неприятных преобразований в указатели. Рассмотрим следующий код:

```
void h()
{
    Circle a1[10];
    array<Circle,10> a2;
    // ...
    Shape* p1 = a1; // OK: ждем неприятностей

    // Ошибка: нет преобразования array<Circle,10> в Shape*:
    Shape* p2 = a2;

    p1[3].draw();   // А вот и неприятности
}
```

Комментарий о неприятностях подразумевает, что `sizeof(Shape)<size of(Circle)`, так что индексация `Circle[]` с использованием `Shape*` дает неверное смещение. Все стандартные контейнеры обладают этим преимуществом перед встроенными массивами.

13.4.2. bitset

Такие аспекты системы, как состояние входного потока, часто представляются в виде набора флагов, соответствующих бинарным условиям (таким, как "хорошо/плохо", "истина/ложь" и "включено/выключено"). C++ эффективно поддерживает концепцию небольших наборов флагов с помощью побитовых операций над целыми числами (§1.4). Класс bitset<N> обобщает это понятие, предоставляя операции над последовательностью из N битов [0;N), где значение N известно во время компиляции. Для наборов битов, которые не вписываются в long long int, использование bitset гораздо удобнее, чем непосредственное использование целых чисел. Для небольших множеств bitset обычно оптимизирован. Если вы хотите именовать биты, а не нумеровать их, можете использовать set (§11.4) или перечисление (§2.5).

bitset может быть инициализирован целым числом или строкой:

```
bitset<9> bs1 {"110001111"};
bitset<9> bs2 {0b1'1000'1111}; // Бинарный литерал с использованием
                               // разделителей между цифрами (§1.4)
```

К bitset можно применять обычные побитовые операторы (§1.4), а также операторы сдвига влево и вправо (<< и >>):

```
bitset<9> bs3 = ~bs1;    // Дополнение: bs3=="001110000"
bitset<9> bs4 = bs1&bs3; // Все нули
bitset<9> bs5 = bs1<<2;  // Сдвиг влево: bs5 = "000111100"
```

Операторы сдвига (здесь — <<) "вдвигают" в число нулевые биты.

Операции to_ullong() и to_string() предоставляют обратные конструкторам операции. Например, вот как мы можем выписать бинарное представление значения типа int:

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;   // Предполагает 8 бит в байте
                                   // (см. также §14.7)
    cout << b.to_string() << '\n'; // Вывод битов числа i
}
```

Этот код выводит биты, представленные как 1 и 0, слева направо, с самым старшим значащим битом слева, так что для аргумента 123 будет выведено

```
00000000000000000000000001111011
```

Для этого примера проще использовать оператор вывода bitset непосредственно:

```
void binary2(int i)
{
    bitset<8*sizeof(int)> b = i;  // Предполагает 8 бит в байте
```

```
                                    // (см. также §14.7)
    cout << b << '\n';              // Вывод битов числа i
}
```

13.4.3. pair и tuple

Часто нам нужны некоторые данные, которые являются просто данными, т.е. набором значений, а не объектом класса с четко определенной семантикой и инвариантом для его значения (§3.5.2). В таких случаях часто бывает идеальной простая структура с соответствующим набором именованных членов. Можно также позволить стандартной библиотеке написать для нас такое определение. Например, алгоритм стандартной библиотеки equal_range возвращает пару итераторов, задающих подпоследовательность, соответствующую предикату:

```
template<typename Forward_iterator, typename T, typename Compare>
    pair<Forward_iterator,Forward_iterator>
    equal_range(Forward_iterator first, Forward_iterator last,
                const T& val, Compare cmp);
```

Для заданной отсортированной последовательности [first;last) алгоритм equal_range() возвращает значение типа pair, представляющее подпоследовательность, значения которой отвечают предикату cmp. Его можно использовать для выполнения поиска в отсортированной последовательности записей типа Record:

```
auto less = [](const Record& r1,    // Сравнение имен
              const Record& r2) { return r1.name<r2.name;};
void f(const vector<Record>& v)     // В предположении отсортированности
{                                   // v по полю "name"

    auto er = equal_range(v.begin(),v.end(),Record{"Reg"},less);

    // Вывод всех записей с именем "Reg":
    for (auto p = er.first; p!=er.second; ++p)
        cout << *p;                 // Считаем, что для Record определен
}                                   // оператор <<
```

Первый член pair называется first, а второй — second. Это наименование не особенно креативно и поначалу может показаться немного странным, но такое последовательное именование — благо, когда мы хотим написать обобщенный код. Там, где имена first и second слишком общие, можно использовать структурное связывание (§3.6.3):

```
void f2(const vector<Record>& v) // В предположении отсортированности
{                                // v по полю "name"

    auto [first,last] = equal_range(v.begin(),v.end(),
                                    Record{"Reg"},less);
```

```
    // Вывод всех записей с именем "Reg":
    for (auto p = first; p!=last; ++p)
        cout << *p;                 // Считаем, что для Record определен
}                                   // оператор <<
```

pair стандартной библиотеки (из заголовочного файла <utility>) довольно часто используется в стандартной библиотеке и в других местах. pair предоставляет такие операторы, как =, == и <, если это делают ее элементы. Вывод типа позволяет легко создать пару pair без явного упоминания ее типа. Например:

```
void f(vector<string>& v)
{
    pair p1 {v.begin(),2};              // Один способ
    auto p2 = make_pair(v.begin(),2); // Другой способ
    // ...
}
```

И p1, и p2 имеют тип pair<vector<string>::iterator,int>.

Если требуется больше двух элементов (или меньше), можно использовать tuple (из того же заголовочного файла <utility>). tuple представляет собой гетерогенную последовательность элементов. Например:

```
// Явное указание типа:
tuple<string,int,double> t1 {"Shark",123,3.14};

// Вывод типа как tuple<string,int,double>:
auto t2 = make_tuple(string{"Herring"},10,1.23);

// Вывод типа как tuple<string,int,double>:
tuple t3 {"Cod"s,20,9.99};
```

Более старый код тяготеет к использованию make_tuple(), так как вывод аргументов типа шаблона из аргументов конструктора появился только в C++17.

Доступ к членам tuple выполняется с помощью шаблона функции get:

```
string s = get<0>(t1);  // Получение первого элемента: "Shark"
int x = get<1>(t1);     // Получение второго элемента: 123
double d = get<2>(t1);  // Получение третьего элемента: 3.14
```

Элементы tuple нумеруются (начиная с нуля); используемые индексы должны быть константами.

Доступ к членам кортежа tuple по индексу является общим, уродливым и в определенной мере чреватым ошибками. К счастью, элемент кортежа с уникальным типом в этом кортеже может быть "именован" по его типу:

```
auto s = get<string>(t1);  // Получение string: "Shark"
auto x = get<int>(t1);     // Получение int: 123
auto d = get<double>(t1);  // Получение double: 3.14
```

Можно использовать `get<>` и для записи:

```
get<string>(t1) = "Tuna"; // Запись в string
get<int>(t1) = 7;         // Запись в int
get<double>(t1) = 312;    // Запись в double
```

Как и пары, кортежи можно присваивать и сравнивать, если их элементы могут это делать. Как и к элементам `tuple`, к элементам `pair` можно получить доступ, используя `get<>()`.

Структурное связывание (§3.6.3) применимо к `tuple` так же, как и к `pair`. Однако, когда коду не требуется обобщенность, простая структура с именованными членами часто приводит к более понятному и легче поддерживаемому коду.

13.5. Альтернативы

Стандартная библиотека предлагает три типа для выражения альтернатив:

- `variant` — для представления одной из определенного множества альтернатив (заголовочный файл — `<variant>`);
- `optional` — для представления значения определенного типа или отсутствия значения (заголовочный файл — `<optional>`);
- `any` — для представления одного из неограниченного множества альтернативных типов (заголовочный файл — `<any>`).

Эти три типа предлагают пользователю схожие функциональные возможности. К сожалению, они не предлагают единого интерфейса.

13.5.1. `variant`

`variant<A,B,C>` зачастую является более удобной и безопасной альтернативой явному использованию `union` (§2.4). Возможно, простейшим примером применения является возврат либо значения, либо кода ошибки:

```
variant<string,int> compose_message(istream& s)
{
    string mess;
    // ... Чтение из s и составление сообщения ...
    if (no_problems)
        return mess;          // Возврат string
    else
        return error_number;  // Возврат int
}
```

При присваивании или инициализации variant значением он запоминает тип этого значения. Позже мы можем узнать, какой именно тип содержит variant, и извлечь значение. Например:

```
auto m = compose_message(cin));
if (holds_alternative<string>(m))
{
    cout << m.get<string>();
}
else {
    int err = m.get<int>();
    // ... Обработка ошибки ...
}
```

Этот стиль нравится некоторым программистам, которым не нравятся исключения (см. §3.5.3), но есть и более интересные применения variant. Например, простому компилятору может потребоваться различать разные виды узлов с разными представлениями:

```
using Node = variant<Expression,Statement,Declaration,Type>;
void check(Node* p)
{
    if (holds_alternative<Expression>(*p))
    {
        Expression& e = get<Expression>(*p);
        // ...
    }
    else if (holds_alternative<Statement>(*p))
    {
        Statement& s = get<Statement>(*p);
        // ...
    }
    // ... Declaration и Type ...
}
```

Такая схема проверки альтернатив для принятия решения о соответствующих действиях настолько распространена и относительно неэффективна, что заслуживает непосредственной поддержки:

```
void check(Node* p)
{
    visit(overloaded {
            [](Expression& e) { /* ... */ },
            [](Statement& s)  { /* ... */ },
            // ... Declaration и Type ...
    }, *p);
}
```

Это в основном эквивалентно вызову виртуальной функции, но потенциально быстрее. Как и во всех случаях, в которых критична производительность, это

"потенциально быстрее" следует проверять измерениями. Для большинства применений разница в производительности незначительна.

К сожалению, нужен `overloaded`, который не является стандартным. Это "кусок магии", который создает множество перегрузки из набора аргументов (обычно — лямбда-выражений):

```
template<class... Ts>
struct overloaded : Ts...
{
    using Ts::operator()...;
};

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>; // Правила вывода
```

"Посетитель" `visit` применяет `()` к `overload`, который в соответствии с правилами вывода выбирает наиболее подходящее лямбда-выражение для вызова.

Правила вывода — это механизм для разрешения тонких неоднозначностей, прежде всего для конструкторов шаблонов классов в фундаментальных библиотеках (§6.2.3).

Если мы попытаемся обратиться к `variant`, хранящему тип, отличный от ожидаемого, то будет сгенерировано исключение `bad_variant_access`.

13.5.2. `optional`

`optional<A>` можно рассматривать как частный случай `variant` (наподобие `variant<A,nothing>`) или как обобщение идеи о том, что `A*` либо указывает на объект, либо имеет значение `nullptr`.

`optional` может быть полезным для функций, которые могут как возвращать объект, так и не возвращать его:

```
optional<string> compose_message(istream& s)
{
    string mess;
    // ... Чтение s и составление сообщения ...
    if (no_problems)
        return mess;
    return {};      // Пустой optional
}
```

При этом можно написать

```
if (auto m = compose_message(cin))
    cout << *m;  // Обратите внимание на разыменование (*)
else {
    // ... Обработка ошибки ...
}
```

Это нравится некоторым программистам, которые недолюбливают исключения (см. § 3.5.5). Обратите внимание на любопытное использование *. optional рассматривается как указатель на свой объект, а не как сам объект.

optional, являющийся эквивалентом nullptr, — пустой объект {}. Например:

```
int cat(optional<int> a, optional<int> b)
{
    int res = 0;
    if (a) res += *a;
    if (b) res += *b;
    return res;
}

int x = cat(17,19);
int y = cat(17,{});
int z = cat({},{});
```

При попытке обратиться к optional, в котором не хранится значение, результат является неопределенным; исключение при этом *не* генерируется. Таким образом, optional не гарантирует безопасность с точки зрения типов.

13.5.3. any

any может содержать произвольный тип и при этом знает, какой тип (если таковой есть) он содержит. Это, по сути, неограниченная версия variant:

```
any compose_message(istream& s)
{
    string mess;
    // ... Чтение s и составление сообщения ...
    if (no_problems)
        return mess;          // Возврат string
    else
        return error_number; // Возврат int
}
```

Когда вы присваиваете или инициализируете any значением, объект запоминает тип этого значения. Позже мы можем узнать, какой именно тип имеет значение any, и извлечь это значение. Например:

```
auto m = compose_message(cin));
string& s = any_cast<string>(m);
cout << s;
```

Если мы пытаемся получить доступ к any с хранимым типом, отличным от ожидаемого, генерируется исключение bad_any_access. Есть также способы доступа к any, которые не полагаются на исключения.

13.6. Аллокаторы

По умолчанию контейнеры стандартной библиотеки выделяют память с помощью new. Операторы new и delete предоставляют общую свободную память (также именуемую динамической памятью или кучей), которая может содержать объекты произвольного размера и со временем жизни, контролируемым пользователем. Это влечет определенные затраты времени и памяти, которые могут быть устранены во многих частных случаях. Поэтому контейнеры стандартной библиотеки предоставляют возможность устанавливать распределители памяти (аллокаторы) с определенной семантикой там, где это необходимо. Эта возможность использовалась для решения широкого спектра проблем, связанных с производительностью (например, аллокаторы, работающие с пулом), безопасностью (аллокаторы, очищающие память как часть процесса ее освобождения), распределением памяти по отдельности для каждого потока, и с неоднородными архитектурами памяти (выделение памяти в определенных хранилищах с соответствующими типами указателей). Эта книга — не место для обсуждения этих важных, но очень уж специализированных и зачастую очень сложных методов. Тем не менее я приведу один пример, связанный с реальной проблемой, для которой решением оказалось применение пула распределения.

Важная, с продолжительным временем работы система использовала очередь событий (см. §15.6), используя векторы vector как события, которые передавались в виде shared_ptr. Таким образом, последний пользователь события неявно его удалял:

```
struct Event
{
    vector<int> data = vector<int>(512);
};

list<shared_ptr<Event>> q;

void producer()
{
    for (int n = 0; n!=LOTS; ++n)
    {
        lock_guard lk {m};   // m – a mutex (§15.5)
        q.push_back(make_shared<Event>());
        cv.notify_one();
    }
}
```

С точки зрения логики все хорошо и просто, код надежный и легко поддерживаемый. К сожалению, этот код привел к сильной фрагментации памяти. После прохождения 100 000 событий от 16 производителей к 4 потребителям было потреблено более 6 Гбайт памяти.

Традиционное решение проблем фрагментации — переписать код с использованием аллокаторов пулов. Аллокатор пула — это распределитель, который управляет объектами единого фиксированного размера и выделяет пространство для множества объектов одновременно, а не использует отдельные выделения. К счастью, C++17 предлагает прямую поддержку этого решения. Распределитель пула определен в подпространстве имен `pmr` (polymorphic memory resource — полиморфный ресурс памяти) в `std`:

```
pmr::synchronized_pool_resource pool; // Создание пула
struct Event
{
    vector<int> data =
            vector<int>{512,&pool};   // Event использует пул
};

list<shared_ptr<Event>> q {&pool};    // q использует пул

void producer()
{
    for (int n = 0; n!=LOTS; ++n)
    {
        scoped_lock lk {m};           // m - mutex (§15.5)
        q.push_back(allocate_shared<Event,
                    pmr::polymorphic_allocator<Event>>{&pool});
        cv.notify_one();
    }
}
```

Теперь, после 100 000 событий, прошедших от 16 производителей к 4 потребителям, было использовано менее 3 Мбайт памяти — примерно 2000-кратное улучшение! Естественно, объем используемой памяти (в отличие от памяти, потраченной на фрагментацию) не изменяется. После устранения фрагментации использование памяти оставалось стабильным с течением времени, поэтому система могла работать месяцами.

Подобные методы применялись с хорошими результатами с первых дней C++, но обычно они требовали переписывания кода, чтобы он использовал специализированные контейнеры. Теперь стандартные контейнеры принимают аллокатор в качестве необязательного аргумента. По умолчанию же контейнеры используют `new` и `delete`.

13.7. Время

В заголовочном файле `<chrono>` стандартная библиотека предоставляет функциональные возможности для работы со временем. Например, вот основной способ замера прошедшего времени:

```
// Подпространство имен std::chrono; см. §3.4:
using namespace std::chrono;

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1-t0).count() << "мс\n";
```

Часы возвращают `time_point` (момент времени). Вычитание двух `time_point` дает нам `duration` (промежуток времени). Различные часы дают свои результаты в разных единицах времени (используемые мною часы изменяют `nanoseconds` — наносекунды), так что обычно желательно конвертировать `duration` в известные единицы. Это делает `duration_cast`.

Не делайте никаких предположений об "эффективности" кода без предварительного измерения времени. Предположения о производительности являются наиболее ненадежными.

Чтобы упростить запись и минимизировать ошибки, заголовочный файл `<chrono>` предлагает суффиксы для единиц времени (§5.4.4). Например:

```
this_thread::sleep(10ms+33us); // Ожидание 10 мс + 33 мкс
```

Эти суффиксы определены в пространстве имен `std::chrono_literals`.

Элегантное и эффективное расширение для `<chrono>`, поддерживающее более длинные интервалы времени (например, годы и месяцы), календари и часовые пояса, будет добавлено в стандарт C++20. В настоящее время оно доступно и широко используется [25, 26]. С ним можно писать такие вещи, как

```
auto spring_day = apr/7/2018;
cout << weekday(spring_day) << '\n'; // Вывод "Saturday"
```

Оно даже умеет обрабатывать дополнительные (високосные) секунды.

13.8. Адаптация функций

При передаче функции в качестве аргумента функции тип аргумента должен точно соответствовать ожидаемому, выраженному в объявлении вызываемой функции. Если предполагаемый аргумент "почти соответствует ожиданиям", у нас есть три хорошие альтернативы:

- использовать лямбда-выражение (§13.8.1);
- использовать `std::mem_fn()`, чтобы сделать из функции-члена функциональный объект (§13.8.2);
- определить функцию как принимающую `std::function` (§13.8.3).

Имеется множество других способов, но обычно лучше всего работает один из трех перечисленных.

13.8.1. Лямбда-выражения в качестве адаптеров

Рассмотрим классический пример "рисования всех фигур":

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

Как и все алгоритмы стандартной библиотеки, `for_each()` вызывает свой аргумент, используя традиционный синтаксис вызова функции `f(x)`, но `draw()` класса `Shape` использует обычную объектно-ориентированную запись `x->f()`. Лямбда-выражение легко становится посредником между двумя записями.

13.8.2. mem_fn()

Для заданной функции-члена функциональный адаптер `mem_fn(mf)` предоставляет функциональный объект, который может быть вызван как свободная функция, не являющаяся членом. Например:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

До введения в C++11 лямбда-выражений `mem_fn()` и его эквиваленты были основным средством отображения объектно-ориентированного стиля вызова на функциональный.

13.8.3. function

Тип стандартной библиотеки `function` представляет собой тип, который может хранить любой объект, могущий быть вызванным с помощью оператора вызова `()`, т.е. объект типа `function` является функциональным объектом (§6.3.2). Например:

```
int f1(double);
function<int(double)> fct1 {f1}; // Инициализация f1
int f2(string);
function fct2 {f2};              // Тип fct2 – function<int(string)>
function fct3 =                  // Тип fct3 – function<void(Shape*)>
    [](Shape* p) { p->draw(); };
```

В случае `fct2` я позволил типу `function` быть выведенным из инициализатора: `int(string)`.

Очевидно, что шаблоны `function` полезны для обратных вызовов, для передачи операций в качестве аргументов, для передачи функциональных объектов и т.д. Однако это может привести к некоторым накладным расхо-

дам времени выполнения по сравнению с прямыми вызовами, а кроме того, `function`, будучи объектом, не участвует в перегрузке. Если вам нужно перегрузить функциональные объекты (включая лямбда-выражения), подумайте о применении `overloaded` (§13.5.1).

13.9. Функции типов

Функция типа (type function) — это функция, которая вычисляется во время компиляции, с типом в качестве аргумента или возвращающая тип. Стандартная библиотека предоставляет множество функций типов, которые помогают разработчикам библиотек (и программистам в целом) писать код, который использует преимущества языка, стандартной библиотеки и кода в целом.

Для числовых типов `numeric_limits` из заголовочного файла `<limits>` представляет различную полезную информацию (§14.7). Например:

```
// Наименьшее положительное значение типа float:
constexpr float min = numeric_limits<float>::min();
```

Точно так же могут быть найдены размеры объекта с помощью встроенного оператора `sizeof` (§1.4). Например:

```
constexpr int szi = sizeof(int); // Количество байтов в int
```

Такие функции типов являются частью механизмов C++ для вычислений во время компиляции, которые обеспечивают более строгую проверку типов и лучшую производительность, чем это было бы возможно в противном случае. Использование таких функций часто называют *метапрограммированием* или (при использовании шаблонов) *шаблонным метапрограммированием*. Здесь я просто представлю два средства, предоставляемые стандартной библиотекой: `iterator_traits` (§13.9.1) и предикаты типов (§13.9.2). Концепты (§7.2) делают некоторые из этих методов излишними и упрощают многие из остальных, но концепты все еще не включены в стандарт и не являются общедоступными, поэтому представленные здесь методы широко используются программистами.

13.9.1. `iterator_traits`

Алгоритм стандартной библиотеки `sort()` использует пару итераторов, которые должны определять последовательность (глава 12, "Алгоритмы"). Кроме того, эти итераторы должны обеспечивать произвольный доступ к элементам этой последовательности, т.е. они должны быть *итераторами с произвольным доступом*. Некоторые контейнеры, такие как `forward_list`, не предлагают такой возможности. В частности, `forward_list` является однос-

вязным списком, поэтому индексация его элементов дорогостоящая, а разумного способа вернуться к предыдущему элементу не существует. Однако, как и большинство контейнеров, `forward_list` предлагает *однонаправленные итераторы*, которые можно использовать для обхода последовательности с помощью алгоритмов и циклов `for` (§6.2).

Стандартная библиотека предоставляет механизм `iterator_traits`, который позволяет проверять, какой тип итератора предоставляется. С учетом этой возможности можно улучшить `sort()` для диапазона из §12.8 так, чтобы он принимал либо `vector`, либо `forward_list`. Например:

```
void test(vector<string>& v, forward_list<int>& lst)
{
    sort(v);    // sort для вектора
    sort(lst);  // sort для односвязного списка
}
```

Методы, необходимые для выполнения этой работы, полезны в общем случае.

Сначала я пишу две вспомогательные функции, которые принимают дополнительный аргумент, указывающий, будет ли функция использоваться для итераторов с произвольным доступом или для однонаправленных итераторов. Версия с аргументами произвольного доступа тривиальна:

```
// Версия для итераторов произвольного доступа:
template<typename Ran>
// Для обращения к элементам [beg:end) можно использовать индексы:
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)
{
    sort(beg,end);    // Просто сортируем последовательность
}
```

Версия для однонаправленных итераторов копирует список в `vector`, сортирует и копирует обратно:

```
// Версия для однонаправленных итераторов:
template<typename For>
// Последовательность [beg:end) можно обойти:
void sort_helper(For beg, For end, forward_iterator_tag)
{
    // Инициализация вектора последовательностью [beg:end):
    vector<Value_type<For>> v {beg,end};
    // Использование сортировки для последовательности с
    sort(v.begin(),v.end());      // произвольным доступом
    copy(v.begin(),v.end(),beg);  // Копирование обратно в список
}
```

`Value_type<For>` — это тип элементов `For`, называемый *типом значения*. Каждый итератор стандартной библиотеки имеет член `value_type`. Я получаю запись `Value_type<For>`, определяя псевдоним типа (§6.4.2):

```
template<typename C>
using Value_type = typename C::value_type; // Значение типа C
```

Таким образом, для `vector<X>` запись `Value_type<X>` представляет собой `X`.

Настоящая "магия типов" состоит в выборе вспомогательной функции:

```
template<typename C>
void sort(C& c)
{
    using Iter = Iterator_type<C>;
    sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Здесь я использую две функции типа: `Iterator_type<C>` возвращает тип итератора `C` (т.е. `C::iterator`), а затем `Iterator_category<Iter>{}` создает значение "дескриптора", указывающее разновидность предоставленного итератора:

- `std::random_access_iterator_tag`, если итератор `C` поддерживает произвольный доступ;
- `std::forward_iterator_tag`, если итератор `C` поддерживает однонаправленную итерацию.

Теперь можно выбирать между двумя алгоритмами сортировки во время компиляции. Этот метод, называемый *диспетчеризацией дескрипторов*, является одним из нескольких методов, используемых в стандартной библиотеке и других местах для повышения гибкости и производительности.

Мы могли бы определить `Iterator_type` следующим образом:

```
template<typename C>
using Iterator_type = typename C::iterator; // Тип итератора C
```

Однако, чтобы распространить эту идею на типы без типов-членов, такие как указатели, поддержка диспетчеризации дескрипторов в стандартной библиотеке представлена в виде шаблона класса `iterator_traits` из заголовочного файла `<iterator>`. Специализация для указателей выглядит следующим образом:

```
template<class T>
struct iterator_traits<T*>
{
    using difference_type = ptrdiff_t;
    using value_type = T;
    using pointer = T*;
    using reference = T&;
    using iterator_category = random_access_iterator_tag;
};
```

Теперь можно написать:

```
template<typename Iter>
using Iterator_category =                    // Категория Iter
    typename std::iterator_traits<Iter>::iterator_category;
```

Теперь `int*` может использоваться в качестве итератора с произвольным доступом, несмотря на отсутствие типа-члена; `Iterator_category<int*>` представляет собой `random_access_iterator_tag`.

Многие свойства и методы, основанные на свойствах, станут ненужными при появлении концептов (§7.2). Рассмотрим версию примера `sort()` с концептами:

```
template<RandomAccessIterator Iter> // Для vector и других типов
void sort(Iter p, Iter q);          // с произвольным доступом

template<ForwardIterator Iter>      // Для list и других типов
void sort(Iter p, Iter q)           // с однонаправленным обходом
{
    vector<Value_type<Iter>> v {p,q};
    sort(v);                        // sort для произвольного доступа
    copy(v.begin(),v.end(),p);
}

template<Range R>
void sort(R& r)
{
    sort(r.begin(),r.end());        // Вызов "правильного" sort
}
```

Прогресс налицо.

13.9.2. Предикаты типов

В заголовочном файле `<type_traits>` стандартная библиотека предлагает простые функции типов, называемые *предикатами типов*, которые отвечают на фундаментальные вопросы о типах. Например, я могу определить шаблонную функцию `Is_arithmetic()`, отвечающую на вопрос о том, является ли некоторый тип арифметическим:

```
bool b1 = Is_arithmetic<int>();     // int – арифметический тип
bool b2 = Is_arithmetic<string>(); // string – не арифметический тип
```

Другими примерами являются `is_class`, `is_pod`, `is_literal_type`, `has_virtual_destructor` и `is_base_of`. Все они особенно полезны при написании шаблонов. Например:

```
template<typename Scalar>
class complex
{
```

```
    Scalar re, im;
    public:
    static_assert(Is_arithmetic<Scalar>(),
                  "complex требует арифметические типы");
    // ...
};
```

Чтобы улучшить удобочитаемость по сравнению с непосредственным использованием стандартной библиотеки, я определил функцию типа:

```
template<typename T>
constexpr bool Is_arithmetic()
{
    return std::is_arithmetic<T>::value;
}
```

Старые программы используют вместо скобок `()` непосредственно `::value`, но я считаю, что это довольно уродливо и раскрывает детали реализации[1].

13.9.3. enable_if

Очевидные способы использования предикатов типов включают условия для `static_assert`, `if` времени компиляции и `enable_if`. `enable_if` стандартной библиотеки — это широко используемый механизм для условного введения определений. Рассмотрим определение "интеллектуального указателя":

```
template<typename T>
class Smart_pointer
{
    T& operator*();
    T& operator->(); // Должно работать тогда и только
};                   // тогда, когда T является классом
```

Оператор `->` должен быть определен тогда и только тогда, когда `T` является типом класса. Например, `Smart_pointer<vector<T>>` должен иметь оператор `->`, а `Smart_pointer<int>` — не должен. Мы не можем использовать `if` времени компиляции, потому что мы не внутри функции. Вместо этого мы пишем

```
template<typename T>
class Smart_pointer
{
    T& operator*();
```

[1] Стандарт C++17 позволяет получать булевы значения, добавляя суффикс `_v`; например, `is_arithmetic_v<T>` является псевдонимом выражения `is_arithmetic<T>::value`. — *Примеч.ред.*

```
    std::enable_if<Is_class<T>(),T&>  // Определен тогда и только
                operator->();         //  тогда, когда T — класс
};
```

Моя функция типа `Is_class()` определена с использованием свойства типа `is_class` точно так же, как определена `Is_aritmetic()` в §13.9.2.

Если `Is_class<T>()` имеет значение `true`, возвращаемый тип `operator->()` — `T&`; в противном случае определение оператора `operator->()` игнорируется.

Синтаксис `enable_if` странный, неудобный в использовании и во многих случаях станет ненужным при появлении концептов (§7.2). Тем не менее `enable_if` является основой современного шаблонного метапрограммирования и многих компонентов стандартной библиотеки. Он основан на тонкой языковой возможности под названием SFINAE ("Substitution Failure Is Not An Error" — "ошибка подстановки ошибкой не является").

13.10. Советы

[1] Чтобы быть полезной, библиотека не обязана быть большой или сложной; §13.1.

[2] Ресурс — это все, что должно быть захвачено, а затем (явно или неявно) освобождено; §13.2.

[3] Для управления ресурсами используйте дескрипторы ресурсов (идиома RAII); §13.2; [CG:R.1].

[4] Используйте `unique_ptr` для обращения к объектам полиморфных типов; §13.2.1; [CG:R.20].

[5] Используйте `shared_ptr` (только) для обращения к совместно используемым объектам; §13.2.1; [CG:R.20].

[6] Предпочитайте дескрипторы ресурсов с конкретной семантикой интеллектуальным указателям; §13.2.1.

[7] Предпочитайте `unique_ptr`, а не `shared_ptr`; §5.3, §13.2.1.

[8] Используйте `make_unique()` для создания `unique_ptr`; §13.2.1; [CG:R.22].

[9] Используйте `make_shared()` для создания `shared_ptr`; §13.2.1; [CG:R.23].

[10] Предпочитайте интеллектуальные указатели сборке мусора; §5.3, §13.2.1.

[11] Не используйте `std::move()`; §13.2.2; [CG:ES.56].

[12] Используйте `std::forward()` исключительно для передачи; §13.2.2.

[13] Никогда не выполняйте чтение объекта после применения к нему `std::move()` или `std::forward()`; §13.2.2.

[14] Предпочитайте `span` интерфейсу, состоящему из указателя и количества элементов; §13.3; [CG:F.24].

[15] Используйте `array` там, где вам нужна последовательность с `constexpr` размером; §13.4.1.

[16] Предпочитайте `array` встроенным массивам; §13.4.1; [CG:SL.con.2].

[17] Используйте `bitset`, если вам нужны `N` битов, и `N` не обязательно является числом битов во встроенном целочисленном типе; §13.4.2.

[18] Не злоупотребляйте типами `pair` и `tuple`; именованные структуры часто приводят к более удобочитаемому коду; §13.4.3.

[19] При использовании `pair` используйте вывод аргументов шаблона или `make_pair()`, чтобы избежать излишней спецификации типов; §13.4.3.

[20] При использовании `tuple` используйте вывод шаблонных аргументов и `make_tuple()`, чтобы избежать излишней спецификации типов; §13.4.3; [CG:T.44].

[21] Предпочитайте `variant` явному применению `union`; §13.5.1; [CG: C.181].

[22] Используйте аллокаторы для предотвращения фрагментации памяти; §13.6.

[23] Выполняйте хронометрирование ваших программ, прежде чем делать какие-либо утверждения об их эффективности; §13.7.

[24] Используйте для отчета об измерении времени приведение `duration_cast` к соответствующими единицам измерения времени; §13.7.

[25] При указании `duration` используйте правильные единицы измерения времени; §13.7.

[26] Используйте `mem_fn()` или лямбда-выражения для создания функциональных объектов, которые могут вызывать функции-члены, когда выполняется вызов с использованием традиционной записи вызова функции; §13.8.2.

[27] Используйте `function`, когда вам необходимо хранить нечто, что может быть вызвано; §13.8.3.

[28] Можно писать код, явно зависящий от свойств типов; §13.9.

[29] Предпочитайте концепты свойствам типов и применению `enable_if`, где это возможно; §13.9.

[30] Используйте псевдонимы и предикаты типов для упрощения записи; §13.9.1, §13.9.2.

14

Числовые вычисления

Цель вычислений — понимание, а не цифры...
— Р.У. Хэмминг

...но для студента цифры часто оказываются лучшей дорогой к пониманию.
— А. Ральстон

14.1. Введение

C++ не был разработан, в первую очередь, для числовых вычислений. Однако числовые вычисления обычно выполняются в контексте другой деятельности, такой как доступ к базам данных, сетевое взаимодействие, управление приборами, графика, моделирование и финансовый анализ, поэтому C++ становится привлекательным средством для вычислений, являющихся частью более крупной системы. Кроме того, численные методы прошли долгий путь от простых циклов по векторам чисел с плавающей точкой. Там, где как часть вычислений необходимы более сложные структуры данных, становятся актуальными сильные стороны C++. В итоге сегодня C++ широко используется для научных, инженерных, финансовых и других вычислений с помощью сложных численных методов. Как следствие появились средства и методы, поддерживающие такие вычисления. В этой главе описываются части стандартной библиотеки, которые поддерживают числовые вычисления.

14.2. Математические функции

В заголовочном файле <cmath> находятся *стандартные математические функции*, такие как sqrt(), log() или sin() для аргументов типов float, double и long double.

	Стандартные математические функции
abs(x)	Абсолютное значение
ceil(x)	Наименьшее целое, не меньшее x
floor(x)	Наибольшее целое, не большее x
sqrt(x)	Квадратный корень; значение x должно быть неотрицательным
cos(x)	Косинус
sin(x)	Синус
tan(x)	Тангенс
acos(x)	Арккосинус; результат неотрицателен
asin(x)	Арксинус; возвращается результат, ближайший к нулю
atan(x)	Арктангенс
cosh(x)	Гиперболический косинус
sinh(x)	Гиперболический синус
tanh(x)	Гиперболический тангенс
exp(x)	Экспонента (*e* в степени x)
log(x)	Натуральный логарифм (по основанию *e*); значение x должно быть положительным
log10(x)	Десятичный логарифм

Версии функций для типа complex (§14.4) находятся в заголовочном файле <complex>. Для каждой функции возвращаемый тип совпадает с типом аргумента.

Об ошибках сообщается путем установки errno из заголовочного файла <cerrno> в EDOM для ошибки области определения и ERANGE — для ошибки области значений. Например:

```
void f()
{
    errno = 0;      // Сброс старого состояния ошибки
    sqrt(-1);
    if (errno==EDOM)
        cerr << "sqrt() не определена для отрицательного аргумента";
    errno = 0;      // Сброс старого состояния ошибки
    pow(numeric_limits<double>::max(),2);
```

```
    if (errno == ERANGE)
        cerr << "Результат pow() слишком велик для double";
}
```

Еще несколько математических функций находятся в заголовочном файле <cstdlib>, а так называемые *специальные математические функции*, такие как beta(), rieman_zeta() и sph_bessel(), также находятся в заголовочном файле <cmath>.

14.3. Числовые алгоритмы

В заголовочном файле <numeric> находится небольшое множество обобщенных числовых алгоритмов, таких как accumulate().

Числовые алгоритмы	
x=accumulate(b,e,i)	x — сумма i и элементов последовательности [b:e)
x=accumulate(b,e,i,f)	accumulate с использованием f вместо +
x=inner_product(b,e, b2,i)	x — скалярное произведение [b:e) и [b2:b2+(e-b)), т.е. сумма i и (*p1)*(*p2) для каждого p1 из [b:e) и соответствующего p2 из [b2:b2+(e-b))
x=inner_product(b,e, b2,i,f,f2)	inner_product с использованием f и f2 вместо + и *
p=partial_sum(b,e,out)	i-й элемент в [out:p) является суммой элементов [b:b+i]
p=partial_sum(b,e,out,f)	partial_sum с использованием f вместо +
p=adjacent_difference (b,e,out)	i-й элемент в [out:p) равен *(b+i)-*(b+i-1) для i>0; если e-b>0, то *out равно *b
p=adjacent_difference (b,e,out,f)	adjacent_difference с использованием f вместо -
iota(b,e,v)	Каждому элементу в [b:e) присваивается значение ++v; таким образом, последовательность становится равной v+1,v+2,...
x=gcd(n,m)	x — наибольший общий делитель целых чисел n и m
x=lcm(n,m)	x — наименьшее общее кратное целых чисел n и m

Эти алгоритмы обобщают распространенные операции, такие как вычисление суммы, позволяя применять их ко всем видам последовательностей. Они также делают операцию, применяемую к элементам этих последовательностей, параметром. Для каждого алгоритма общая версия дополняется версией, применяющей наиболее распространенный оператор для этого алгоритма. Например:

```
list<double> lst {1, 2, 3, 4, 5, 9999.99999};
auto s = accumulate(lst.begin(),lst.end(),0.0); // Сумма: 10014.9999
```

Описанные алгоритмы работают для любой последовательности стандартной библиотеки и могут иметь операции, предоставляемые в качестве аргументов (§14.3).

14.3.1. Параллельные алгоритмы

В заголовочном файле <numeric> числовые алгоритмы имеют немного различающиеся параллельные версии (§12.9).

Параллельные числовые алгоритмы	
`x=reduce(b,e,v)`	`x=accumulate(b,e,v)`, за исключением порядка вычислений
`x=reduce(b,e)`	`x=reduce(b,e,V{})`, где `V` — тип значения `b`
`x=reduce(pol,b,e,v)`	`x=reduce(b,e,v)` со стратегией выполнения `pol`
`x=reduce(pol,b,e)`	`x=reduce(pol,b,e,V{})`, где `V` — тип значения `b`
`p=exclusive_scan (pol,b,e,out)`	`p=partial_sum(b,e,out)` в соответствии со стратегией `pol`, исключая `i`-й элемент из `i`-й суммы
`p=inclusive_scan (pol,b,e,out)`	`p=partial_sum(b,e,out)` со стратегией выполнения `pol` и включением `i`-го элемента в `i`-ю сумму
`p=transform_reduce (pol,b,e,f,v)`	`f(x)` для каждого `x` из `[b:e)`, затем `reduce`
`p=transform_exclusive_ scan(pol,b,e,out,f,v)`	`f(x)` для каждого `x` из `[b:e)`, затем `exclusive_scan`
`p=transform_inclusive_ scan(pol,b,e,out,f,v)`	`f(x)` для каждого `x` из `[b:e)`, затем `inclusive_scan`

Для простоты я не показал версии алгоритмов, которые принимают в качестве аргумента функтор, а не просто используют + и =. За исключением `reduce()`, я также не показал версии со стратегией выполнения по умолчанию (последовательное выполнение) и значением по умолчанию.

Так же, как и для параллельных алгоритмов в заголовочном файле <algorithm> (§12.9), мы можем определить стратегию выполнения:

```
vector<double> v {1, 2, 3, 4, 5, 9999.99999};
// Вычисление суммы с использованием double в качестве накопителя:
auto s = reduce(v.begin(),v.end());

vector<double> large;
// ... Заполнение large большим количеством значений ...
```

```
// Вычисление суммы с использованием доступного параллелизма:
auto s2 = reduce(par_unseq,large.begin(),large.end());
```

Параллельные алгоритмы (например, `reduce()`) отличаются от последовательных (например, `accumulate()`) тем, что допускают выполнение операций над элементами в неопределенном порядке.

14.4. Комплексные числа

Стандартная библиотека поддерживает семейство типов комплексных чисел по аналогии с классом `complex`, описанным в §4.2.1. Для поддержки комплексных чисел, в которых скаляры являются числами одинарной точности с плавающей запятой (`float`), двойной точности с плавающей запятой (`double`) и другими, тип `complex` стандартной библиотеки является шаблоном:

```
template<typename Scalar>
class complex
{
public:
    // Аргументы функции по умолчанию; см. §3.6.1:
    complex(const Scalar& re ={}, const Scalar& im ={});
    // ...
};
```

Для комплексных чисел поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld {fl+sqrt(db)};
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

Функции `sqrt()` и `pow()` (возведение в степень) находятся среди обычных математических функций, определенных в заголовочном файле стандартной библиотеки <complex> (§14.2).

14.5. Случайные числа

Случайные числа полезны во многих контекстах, таких как тестирование, игры, моделирование и безопасность. Разнообразие областей применения отражается в широком выборе генераторов случайных чисел, предоставляемых стандартной библиотекой в заголовочном файле <random>. Генератор случайных чисел состоит из двух частей.

[1] Собственно *генератора* (engine), производящего последовательность случайных или псевдослучайных чисел.

[2] *Распределения* (distribution), которое отображает полученные значения в математическое распределение в некотором диапазоне.

Примерами распределений являются `uniform_int_distribution` (все целые числа получаются с одинаковой вероятностью), `normal_distribution` (нормальное (гауссово) распределение) и `exponential_distribution` (экспоненциальное распределение); каждое из них применяется для определенного диапазона. Например:

```
using my_engine = default_random_engine;            // Генератор
using my_distribution = uniform_int_distribution<>; // Распределение

my_engine re {};                    // Генератор по умолчанию
my_distribution one_to_six {1,6}; // Распределение в диапазоне 1..6

auto dice = [](){ return one_to_six(re); } // Создание ГСЧ
int x = dice();                            // Бросание кости: x∈[1:6]
```

Благодаря своему бескомпромиссному вниманию к общности и производительности случайные числа стандартной библиотеки были охарактеризованы одним экспертом как "то, чем хочет быть каждая библиотека случайных чисел, когда вырастет". Однако эту часть стандартной библиотеки вряд ли можно считать дружелюбной по отношению к новичкам. Использование инструкций `using` и лямбда-выражений делает код немного более понятным.

Для новичков (с любыми базовыми знаниями) серьезным препятствием может стать очень общий интерфейс библиотеки случайных чисел. Однако для начала работы зачастую достаточно простого генератора случайных чисел с равномерным распределением. Например:

```
Rand_int rnd {1,10}; // Генератор случайных чисел в диапазоне [1:10]
int x = rnd();       // x – случайное число в диапазоне [1:10]
```

Итак, как мы можем получить такой генератор? Нам нужно что-то наподобие рассмотренного выше `dice()`, что объединяет генератор с распределением внутри класса `Rand_int`:

```
class Rand_int
{
public:
    Rand_int(int low, int high) :dist{low,high} { }
    int operator()() { return dist(re); } // Генерация int
    void seed(int s) { re.seed(s); }      // Инициализация ГСЧ
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
};
```

Это определение по-прежнему находится на "уровне эксперта", но применение `Rand_int()` возможно уже на первой неделе курса C++ для новичков. Например:

```
int main()
{
    constexpr int max = 9;
    Rand_int rnd {0,max};          // ГСЧ с равномерным распределением

    vector<int> histogram(max+1); // Вектор подходящего размера
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];        // Заполнение гистограммы частотами

    for (int i = 0; i!=histogram.size(); ++i)     // Вывод гистограммы
    {
        cout << i << '\t';
        for (int j=0; j!=histogram[i]; ++j) cout << '*';
        cout << endl;
    }
}
```

Результатом оказывается (обнадеживающе скучное) равномерное распределение (с разумными статистическими отклонениями):

```
0 **********************
1 ****************
2 ********************
3 **********************
4 ****************
5 ************************
6 **************************
7 ***********
8 ************************
9 *************************
```

Стандартной графической библиотеки в C++ не существует, поэтому я использую "ASCII-графику". Очевидно, что существует множество программ с открытым исходным кодом, коммерческой графики и графических библиотек для C++, но я в этой книге ограничусь стандартными средствами ISO.

14.6. Векторная арифметика

`vector`, описанный в §11.2, был разработан в качестве общего механизма для хранения значений, который был бы гибким и вписывался в архитектуру контейнеров, итераторов и алгоритмов. Однако он не поддерживает математические векторные операции. Добавление таких операций к `vector` было бы простым, но его универсальность и гибкость исключают возможности оптимизации, которые часто считаются необходимыми для серьезных числен-

ных методов. Поэтому стандартная библиотека предоставляет (в заголовочном файле <valarray>) векторный шаблон valarray, который является менее общим и более поддающимся оптимизации для численных вычислений:

```
template<typename T>
class valarray
{
    // ...
};
```

Для valarray поддерживаются обычные арифметические операции и наиболее распространенные математические функции. Например:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    // Числовые операторы *, +, / и = для массивов:
    valarray<double> a = a1*3.14+a2/a1;
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

В дополнение к арифметическим операциям valarray предлагает быстрый доступ, облегчающий реализацию многомерных вычислений.

14.7. Границы числовых значений

В заголовочном файле <limit> стандартная библиотека предоставляет классы, которые описывают свойства встроенных типов — такие, как максимальный показатель степени для float или количество байтов в int. Например, мы можем проверить во время компиляции, знаковым ли типом является char:

```
static_assert(numeric_limits<char>::is_signed,"Символы беззнаковые!");
static_assert(100000<numeric_limits<int>::max(),"Слишком малый int!");
```

Обратите внимание, что вторая проверка работает только потому, что numeric_limits<int>::max() является constexpr-функцией (§1.6).

14.8. Советы

[1] Проблемы, связанные с числовыми вычислениями, зачастую довольно тонкие. Если вы не уверены на 100% в математических аспектах числовых вычислений, обратитесь за советом к специалисту или проведите эксперименты (или сделайте и то, и другое); §14.1.

[2] Не пытайтесь работать с серьезными числовыми вычислениями, ограничиваясь возможностями "голого" языка — используйте библиотеки; §14.1.

[3] Перед тем как писать цикл для вычисления значения из последовательности, рассмотрите возможность применения `accumulate()`, `inner_product()`, `partial_sum()` и `adjacent_difference()`; §14.3.

[4] Используйте `std::complex` для комплексной арифметики; §14.4.

[5] Для получения генератора случайных чисел свяжите генератор с распределением; §14.5.

[6] Следите, чтобы ваши случайные числа были достаточно случайными; §14.5.

[7] Не используйте `rand()` из стандартной библиотеки C; для серьезного применения этот генератор недостаточно случаен; §14.5.

[8] Используйте `valarray` для численных вычислений, когда эффективность времени выполнения важнее гибкости по отношению к операциям и типам элементов; §14.6.

[9] Свойства числовых типов доступны посредством `numeric_limits`; §14.7.

[10] Используйте `numeric_limits` для проверки пригодности числовых типов для предполагаемого применения; §14.7.

15

Параллельные вычисления

Все следует упрощать до тех пор,
пока это возможно, но не более того.

— А. Эйнштейн

15.1. Введение

Параллелизм — выполнение нескольких задач одновременно — широко используется для повышения пропускной способности (путем использования нескольких процессоров для единого вычисления) или для снижения времени отклика (позволяя одной части программы выполняться, в то время как другая ожидает ответа). Все современные языки программирования поддерживают эту возможность. Поддержка, предоставляемая стандартной библиотекой C++, представляет собой переносимый и безопасный с точки зрения типов вариант того, что использовалось в C++ более 20 лет и почти повсеместно поддерживается современным оборудованием. Поддержка стандартной библиотеки, в первую очередь, направлена на поддержку параллелизма на системном уровне, а не на непосредственное предоставление сложных моделей параллелизма более высокого уровня; эти модели могут быть предоставлены

в виде библиотек, созданных с использованием средств стандартной библиотеки.

Стандартная библиотека непосредственно поддерживает одновременное выполнение нескольких потоков в одном адресном пространстве. Для этого C++ предоставляет подходящую модель памяти и набор атомарных операций. Атомарные операции обеспечивают возможность программирования без применения блокировок [16]. Данная модель памяти гарантирует, что, пока программист избегает гонок данных (неконтролируемого одновременного доступа к изменяемым данным), все работает так, как можно было бы наивно ожидать. Однако большинство пользователей видят параллелизм только с точки зрения стандартной библиотеки и библиотек, построенных на ее основе. В этом разделе кратко рассмотрены примеры основных средств поддержки параллелизма стандартной библиотеки: `thread`, `mutex`, операции `lock()`, `packaged_task` и `future`. Эти средства построены непосредственно на примитивах, предлагаемых операционными системами, и не приводят к снижению производительности по сравнению с их непосредственным применением. Они также не обещают какого-либо существенного улучшения производительности по сравнению с тем, что предлагает операционная система.

Не считайте параллелизм панацеей. Если задача может быть выполнена последовательно, часто проще и быстрее именно так и поступить.

В качестве альтернативы использованию явных возможностей параллелизма зачастую можно прибегнуть к параллельному алгоритму, чтобы использовать несколько механизмов повышения производительности (§12.9, §14.3.1).

15.2. Задания и потоки

Вычисление, которое потенциально может быть выполнено одновременно с другими вычислениями, называется *заданием* (task). *Поток* (thread) — это системное представление задания в программе. Задание, которое должно выполняться одновременно с другими заданиями, запускается на выполнение путем создания объекта `std::thread` (описан в заголовочном файле `<thread>`) с заданием в качестве аргумента. Задание представляет собой функцию или функциональный объект:

```
void f();               // Функция
struct F                // Функциональный объект
{
    void operator()(); // Оператор вызова в F (§6.3.2)
};

void user()
{
    thread t1 {f};      // f() выполняется в отдельном потоке
```

```
    thread t2 {F()};    // F()() выполняется в отдельном потоке
    t1.join();          // Ожидание потока t1
    t2.join();          // Ожидание потока t2
}
```

Вызовы `join()` гарантируют, что мы не выйдем из `user()`, пока потоки не будут завершены. "Присоединиться" к потоку `thread` означает "ждать окончания работы потока".

Потоки программы совместно используют одно адресное пространство. В этом потоки отличаются от процессов, которые в общем случае непосредственно не используют данные совместно. Это отличие потоков позволяет им взаимодействовать через совместно используемые объекты (§15.5). Такой обмен информацией обычно контролируется блокировками или другими механизмами, предотвращающими гонку данных (неконтролируемый одновременный доступ к переменной).

Программирование параллельных заданий может быть *очень* сложным. Рассмотрим возможные реализации заданий `f` (функции) и `F` (функционального объекта):

```
void f()
{
    cout << "Hello ";
}
struct F
{
    void operator()() { cout << "Parallel World!\n"; }
};
```

Это пример ошибки: здесь `f` и `F()` используют объект `cout` без какой-либо синхронизации. Полученный результат будет непредсказуемым и может изменяться от запуска программы к запуску, так как порядок выполнения отдельных операций в двух задачах не определен. Программа может выдавать "странный" результат, такой как

```
PaHerallllel o World!
```

Только определенные гарантии в стандарте спасают нас от гонки данных в рамках определения `ostream`, которая может привести к краху приложения.

При определении заданий параллельной программы наша цель состоит в том, чтобы полностью разделить задания, кроме случаев, когда они общаются между собой простым и очевидным образом. Самый простой способ представить себе параллельное задание — представить его в виде функции, которая работает одновременно с функцией, вызвавшей ее. Для этого нам просто нужно передать аргументы, получить результат обратно и убедиться, что у них нет никаких совместно используемых данных (отсутствуют гонки данных).

15.3. Передача аргументов

Как правило, для работы заданию требуются данные. Мы можем легко передавать их (или указатели, или ссылки на данные) в качестве аргументов. Рассмотрим следующий код:

```
void f(vector<double>& v);  // Функция, работающая с v
struct F                    // Функциональный объект, работающий с v
{
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();      // Оператор приложения; §6.3.2
};

int main()
{
    vector<double> some_vec {1,2,3,4,5,6,7,8,9};
    vector<double> vec2 {10,11,12,13,14};

    // f(some_vec) работает в отдельном потоке:
    thread t1 {f,ref(some_vec)};

    // F(vec2)() работает в отдельном потоке:
    thread t2 {F{vec2}};

    t1.join();
    t2.join();
}
```

Очевидно, что `F{vec2}` сохраняет ссылку на вектор-аргумент в `F`. Теперь `F` может использовать этот вектор, и, будем надеяться, никакая другая задача не будет обращаться к `vec2`, пока выполняется `F` (передача `vec2` по значению устраняет этот риск).

Инициализация с помощью `{f,ref(some_vec)}` использует конструктор шаблона `thread` с переменным количеством аргументов, который может принимать произвольную последовательность аргументов (§7.4). `ref()` — это функция типа из `<functional>`, которая, к сожалению, необходима для того, чтобы шаблон переменной рассматривал `some_vec` как ссылку, а не как объект. Без `ref()` аргумент `some_vec` будет передаваться по значению. Компилятор проверяет, может ли первый аргумент быть вызван с переданными аргументами, и создает необходимый функциональный объект для передачи потоку. Таким образом, если `F::operator()()` и `f()` выполняют один и тот же алгоритм, то работа двух заданий грубо эквивалентна: в обоих случаях создается функциональный объект для выполнения `thread`.

15.4. Возврат результатов

В примере в §15.3 я передаю аргументы по неконстантной ссылке. Я делаю это только в том случае, если ожидаю, что задание будет изменять значение указываемых данных (§1.7). Это немного хитрый, но не столь уж редкий способ вернуть результат. Более понятный метод состоит в том, чтобы передать входные данные по константной ссылке, а также в качестве отдельного аргумента передать местоположение для размещения результата:

```
// Входные данные - в v; результат размещается в *res:
void f(const vector<double>& v, double* res);
class F
{
public:
    F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()();          // Размещение результата в *res
private:
    const vector<double>& v;    // Источник входных данных
    double*res;                 // Запись выходных данных
};
double g(const vector<double>&); // Использует возвращенное значение
void user(vector<double>& vec1, vector<double> vec2,
          vector<double> vec3)
{
    double res1;
    double res2;
    double res3;

    // f(vec1,&res1) выполняется в отдельном потоке:
    thread t1 {f,cref(vec1),&res1};

    // F{vec2,&res2}() выполняется в отдельном потоке:
    thread t2 {F{vec2,&res2}};

    // Захват локальных переменных по ссылке:
    thread t3 { [&](){ res3 = g(vec3); } };

    t1.join();
    t2.join();
    t3.join();

    cout << res1 << ' ' << res2 << ' ' << res3 << '\n';
}
```

Этот способ работает; данная методика очень распространена, но я не считаю возврат результатов по ссылке особенно элегантным, поэтому возвращусь к этой теме в §15.7.1.

15.5. Совместное использование данных

Иногда задания должны совместно использовать некоторые данные. В этом случае доступ к ним должен быть синхронизирован, чтобы одновременно к ним могло обращаться не более одного задания. Опытные программисты рассматривают это как упрощение (например, не возникает никаких проблем со многими заданиями, одновременно считывающими неизменяемые данные), тем не менее мы рассмотрим, как обеспечить выполнение условия, чтобы одновременно доступ к заданному набору объектов имело не более одного задания.

Основополагающим элементом решения этой задачи является `mutex` (mutual exclusion object — объект взаимоисключения, мьютекс). Поток `thread` захватывает `mutex` с помощью операции `lock()`:

```
mutex m;                    // Управляющий мьютекс
int sh;                     // Совместно используемые данные
void f()
{
    scoped_lock lck {m}; // Захват мьютекса
    sh += 7;                // Работа с совместно используемыми данными
}                           // Неявное освобождение мьютекса
```

Тип `lck` выводится как `scoped_lock<mutex>` (§6.2.3). Конструктор `scoped_lock` захватывает мьютекс (вызовом `m.lock()`). Если другой поток уже захватил мьютекс, наш поток ожидает ("блокируется"), пока другой поток не завершит свой доступ к данным. Как только другой поток завершит свой доступ к общим данным, `scoped_lock` освобождает мьютекс (вызовом `m.unlock()`). Когда мьютекс освобождается, потоки, ожидающие его, возобновляют выполнение ("пробуждаются"). Средства взаимного исключения и блокировки находятся в заголовочном файле `<mutex>`.

Обратите внимание на использование идиомы RAII (§5.3). Использовать дескрипторы ресурсов, такие как `scoped_lock` и `unique_lock` (§15.6), проще и намного безопаснее, чем явно блокировать и разблокировать мьютексы.

Соответствие между совместно используемыми данными и мьютексом является обычным: программист просто должен знать, какой мьютекс каким данным соответствует. Очевидно, что это чревато ошибками, и в равной степени очевидно, что мы будем пытаться прояснить соответствие с помощью различных языковых средств. Например:

```
class Record
{
public:
    mutex rm;
    // ...
};
```

Не нужно быть гением, чтобы догадаться, что для записи Record с именем rec необходимо захватить rec.rm перед обращением к другим данным в rec, хотя комментарий или лучшее имя могли бы читателю помочь.

Для выполнения некоторых действий нередко требуется одновременный доступ к нескольким совместно используемым ресурсам, что может привести к взаимоблокировке. Например, если thread1 захватывает mutex1, а затем пытается захватить mutex2, в то время как thread2 захватывает mutex2, а затем пытается захватить mutex1, то ни одно из заданий не сможет выполняться. scoped_lock помогает разрешить эту ситуацию, позволяя нам получить несколько блокировок одновременно:

```
void f()
{
    scoped_lock lck {mutex1,mutex2,mutex3}; // Захват трех блокировок
    // ... Работа с совместно используемыми данными ...
}   // Неявное освобождение всех мьютексов
```

Этот scoped_lock разрешит продолжить работу только после захвата всех аргументов-мьютексов и при этом никогда не окажется в заблокированном состоянии, удерживая мьютекс. Деструктор scoped_lock обеспечивает освобождение мьютексов, когда поток покидает область видимости.

Обмен информацией через совместно используемые данные — средство довольно низкоуровневое. В частности, программист должен разработать некоторые способы, чтобы знать, какая работа выполнялась, а какая не выполнялась различными заданиями. Поэтому применение совместно используемых данных уступает понятию вызова и возврата. С другой стороны, некоторые программисты убеждены, что совместное использование должно быть более эффективным, чем копирование аргументов и результатов. Это действительно может быть так, когда речь идет о больших объемах данных, но блокировка и разблокировка являются относительно дорогостоящими операциями. Кроме того, современные машины очень хорошо копируют данные, особенно компактные, такие как элементы векторов. Поэтому не выбирайте способ обмена через совместно используемые данные из-за "эффективности" без размышлений и измерений.

Базовый mutex позволяет одновременно получать доступ к данным одному потоку. Но одним из наиболее распространенных вариантов обмена данными является много читателей и один писатель. Эта идиома "блокировки читателя-писателя" поддерживается мьютексом shared_mutex. Читатель захватывает "совместно используемый" мьютекс так, что другие читатели все еще могут получить доступ к данным, в то время как писатель требует эксклюзивного доступа. Например:

```
shared_mutex mx;          // Совместно используемый мьютекс
void reader()
{
    shared_lock lck {mx}; // Разрешает совместный доступ читателям
    // ... Чтение ...
}

void writer()
{
    unique_lock lck {mx}; // Эксклюзивный доступ
    // ... Запись ...
}
```

15.6. Ожидание событий

Иногда поток должен ожидать какого-то внешнего события, например завершения задания другим потоком, или пока не пройдет определенное количество времени. Самое простое "событие" — это просто прошедшее время. Используя средства для работы со временем, найденные в заголовочном файле <chrono>, можно написать:

```
using namespace std::chrono;    // См.§13.7

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1-t0).count() << " нс прошло\n";
```

Обратите внимание, что мне даже не надо запускать thread; по умолчанию this_thread относится к одному и только к одному потоку.

Я использовал duration_cast, чтобы настроить единицы измерения времени в нужные мне наносекунды.

Базовая поддержка взаимодействия с использованием внешних событий обеспечивается условными переменными condition_variable, описанными в заголовочном файле <condition_variable>. Такая переменная представляет собой механизм, позволяющий одному потоку ожидать другого. В частности, он позволяет потоку ожидать выполнения некоторого *условия* (часто называемого *событием*) в результате работы, выполняемой другими потоками.

condition_variable поддерживает множество форм элегантного и эффективного совместного использования информации, но может быть довольно сложным. Рассмотрим классический пример взаимодействия двух потоков путем передачи сообщений через очередь queue. Для простоты я объявляю

очередь и механизм предотвращения условий гонки в этой очереди глобальными для производителя и потребителя:

```
class Message               // Передаваемый объект
{
    // ...
};

queue<Message> mqueue;      // Очередь сообщений
condition_variable mcond;   // Переменная для сообщения о событиях
mutex mmutex;               // Синхронизация доступа к mcond
```

Типы `queue`, `condition_variable` и `mutex` предоставляются стандартной библиотекой.

Потребитель `consumer()` читает и обрабатывает сообщения `Message`:

```
void consumer()
{
    while(true)
    {
        unique_lock lck {mmutex};   // Захват мьютекса
        mcond.wait(lck,[]{return !mqueue.empty();}); // Освобождение
                    // lck и ожидание; повторный захват lck при
                    // пробуждении; не пробуждаться, пока очередь не
                    // перестанет быть пустой

        auto m = mqueue.front();    // Получение сообщения
        mqueue.pop();
        lck.unlock();               // Освобождение lck
        // ... Обработка m ...
    }
}
```

Здесь я явно защищаю операции с очередью и условной переменной с помощью блокировки `unique_lock` для мьютекса. Ожидание `condition_variable` освобождает свой аргумент-блокировку до тех пор, пока ожидание не закончится (пока очередь не перестанет быть пустой), а затем повторно захватывает ее. Явная проверка условия (в данном случае — `!mqueue.empty()`) защищает от пробуждения просто для того, чтобы обнаружить, что какое-то другое задание пробудилось раньше, так что условие больше не выполняется.

Я использовал `unique_lock`, а не `scoped_lock` по двум причинам.

- Нам нужно передать блокировку функции `wait()` условной переменной `condition_variable`. Блокировка `scoped_lock`, в отличие от `unique_lock`, не может копироваться.
- Мы хотим разблокировать `mutex`, защищающий условную переменную, перед обработкой сообщения. `unique_lock` предлагает такие

операции, как `lock()` и `unlock()`, для низкоуровневого управления синхронизацией.

С другой стороны, `unique_lock` может работать только с одним мьютексом.

Соответствующий производитель `producer()` имеет следующий вид:

```
void producer()
{
    while(true)
    {
        Message m;
        // ... Заполнение сообщения ...
        scoped_lock lck {mmutex}; // Защита операций
        mqueue.push(m);
        mcond.notify_one();       // Уведомление
    }   // Освобождение блокировки (в конце области видимости)
}
```

15.7. Обмен информацией с заданиями

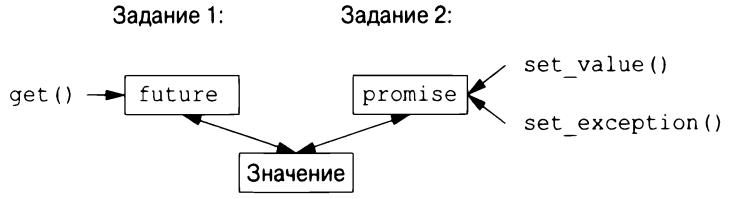
Стандартная библиотека предоставляет несколько средств, позволяющих программистам работать на концептуальном уровне заданий (потенциально работающих параллельно), а не непосредственно на низком уровне потоков и блокировок.

- `future` и `promise` для возврата значения из задания, запущенного в отдельном потоке.
- `packaged_task` для помощи в запуске задач и подключении механизмов для возврата результата.
- `async()` для запуска заданий способом, очень похожим на вызов функции.

Все эти средства находятся в заголовочном файле `<future>`.

15.7.1. `future` и `promise`

Важным моментом в отношении `future` и `promise` является то, что они позволяют передавать значения между двумя заданиями без явного использования блокировки; "система" эффективно реализует передачу. Основная идея проста: когда задание хочет передать значение другому, оно помещает значение в `promise`. Каким-то образом реализация заставляет это значение появиться в соответствующем `future`, из которого оно может быть прочитано (как правило, тем, кем задание было запущено). Это можно представить графически следующим образом.



Если у нас есть future<X> с именем fx, мы можем получить из него значение типа X с помощью функции get():

```
X v = fx.get(); // При необходимости ожидаем вычисление значения
```

Если значение еще не готово, наш поток блокируется, пока не получит требуемое значение. Если значение не может быть вычислено, get() может сгенерировать исключение (из системы или переданное из задачи, из которой мы пытались получить это значение).

Основная цель promise — предоставить простые операции передачи значения (именуемые set_value() и set_exception()), соответствующие get() у future. Имена future (будущее) и promise (обещание) сложились исторически; пожалуйста, не обвиняйте и не благодарите меня за этот неограниченный источник каламбуров.

Если у вас есть promise и вам нужно передать результат типа X в future, вы можете сделать одно из двух: передать значение или передать исключение. Например:

```
void f(promise<X>& px)  // Задание: поместить result в px
{
    // ...
    try {
        X res;
        // ... Вычисление значения res ...
        px.set_value(res);
    }
    catch (...) {       // Ой! Не удалось вычислить res
        // Передача исключения в поток future:
        px.set_exception(current_exception());
    }
}
```

Здесь current_exception() указывает перехваченное исключение.

Чтобы обработать исключение, переданное через future, вызывающая get() функция должна быть готова где-то его перехватить. Например:

```
void g(future<X>& fx)   // Задание: получение результата из fx
{
    // ...
    try {
        X v = fx.get(); // При необходимости ожидание вычисления
```

```
        // ... Использование v ...
    }
    catch (...) {       // Ой! Не удалось вычислить v
        // ... Обработка ошибки ...
    }
}
```

Если ошибку не нужно обрабатывать в самой функции `g()`, код сводится к минимуму:

```
void g(future<X>& fx)   // Задание: получение результата из fx
{
    // ...
    X v = fx.get(); // При необходимости ожидание вычисления
    // ... Использование v ...
}
```

15.7.2. packaged_task

Как мы можем получить `future` в задании, которому нужен результат, и соответствующий `promise` в потоке, который должен этот результат вычислить? Тип `packaged_task` предназначен для упрощения настройки заданий, связанных с работой с `future` и `promise` в потоках. `packaged_task` предоставляет код-обертку для размещения возвращаемого значения или исключения из задачи в `promise` (подобно коду из §15.7.1). Если вы попросите об этом, вызвав `get_future`, `packaged_task` выдаст вам `future`, соответствующее его `promise`. Например, мы можем настроить два задания так, чтобы каждое суммировало половину элементов `vector<double>`, используя алгоритм стандартной библиотеки `accumulate()` (§14.3):

```
// Вычисление суммы элементов [beg:end) с начальным значением init:
double accum(double* beg, double* end, double init)
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double); // Тип задания

    packaged_task<Task_type> pt0 {accum}; // Упаковка задания
    packaged_task<Task_type> pt1 {accum}; // (т.е. accum)

    future<double> f0 {pt0.get_future()}; // Получение future pt0
    future<double> f1 {pt1.get_future()}; // Получение future pt1

    double* first = &v[0];
    thread t1 {move(pt0),first,           // Запуск потока для pt0
               first+v.size()/2,0};
```

```
    thread t2 {move(pt1),                    // Запуск потока для pt1
              first+v.size()/2,first+v.size(),0};
    // ...
    return f0.get()+f1.get();                // Получение результата
}
```

Шаблон `packaged_task` принимает в качестве аргумента шаблона тип задания (здесь — `Task_type`, псевдоним для `double(double*,double*, double)`) и задание в качестве аргумента конструктора (в данном случае — `accum`). Операция `move()` необходима потому, что `packaged_task` не может быть скопирован. Причина, по которой `packaged_task` не может быть скопирован, заключается в том, что он является дескриптором ресурса: он владеет своим `promise` и (косвенно) ответственен за любые ресурсы, которыми может владеть его задание.

Обратите внимание на отсутствие явного упоминания блокировок в этом коде: мы можем сосредоточиться на заданиях, которые необходимо выполнить, а не на механизмах, используемых для управления связью между ними. Эти две задачи будут выполняться в отдельных потоках, а следовательно, потенциально параллельно.

15.7.3. `async()`

В этой главе я придерживался образа мышления, который считаю самым простым, но все же одним из самых сильных: рассматривайте задание как функцию, которая может выполняться одновременно с другими заданиями. Это далеко не единственная модель, поддерживаемая стандартной библиотекой C++, но она хорошо подходит для широкого спектра нужд. При необходимости могут использоваться более тонкие и хитрые модели (например, стили программирования, основанные на совместно используемой памяти).

Для запуска заданий, которые могут выполняться асинхронно, мы можем использовать функцию `async()`:

```
// Запуск нескольких заданий для достаточно большого v:
double comp4(vector<double>& v)
{
    if (v.size() < 10000) // Стоит ли возиться с параллельностью?..
        return accum(v.begin(),v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0=async(accum,v0,v0+sz/4,0.0);       // Первая четверть
    auto f1=async(accum,v0+sz/4,v0+sz/2,0.0);  // Вторая четверть
    auto f2=async(accum,v0+sz/2,v0+sz*3/4,0.0);// Третья четверть
    auto f3=async(accum,v0+sz*3/4,v0+sz,0.0);  // Четвертая четверть
```

```
    // Накопление и объединение результатов:
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

По сути, `async()` отделяет "часть вызова" функции от части "получения результата" и отделяет их обе от фактического выполнения задания. Используя `async()`, вам не нужно думать о потоках и блокировках. Вместо этого вы мыслите просто с точки зрения заданий, которые потенциально асинхронно вычисляют свои результаты. Существует очевидное ограничение: даже не думайте об использовании `async()` для задач, которые совместно используют ресурсы, требующие блокировки. При использовании `async()` вы даже не знаете, сколько потоков будет использоваться, потому что `async()` принимает решение, основанное на том, что ей известно о системных ресурсах, доступных во время вызова. Например, `async()` может проверить, имеются ли какие-либо простаивающие ядра (процессоры), прежде чем решить, сколько потоков использовать.

Использование предположения о стоимости вычислений для принятия решения о запуске потока, такое как `v.size()<10000`, очень примитивно и может привести к грубым ошибкам производительности. Однако данная книга — не место для надлежащего обсуждения того, как управлять потоками. Не воспринимайте эту оценку как нечто большее, чем простое и, вероятно, неправильное предположение.

Редко бывает необходимо вручную распараллеливать алгоритм стандартной библиотеки, такой как `accumulate()`, потому что готовые параллельные алгоритмы, такие как `reduce(par_unseq,/*...*/)`, обычно лучше с этим справляются (§14.3. 1). Тем не менее такая методика широко применима.

Обратите внимание, что `async()` — это не просто механизм, предназначенный для параллельных вычислений для повышения производительности. Например, его также можно использовать для запуска задания для получения информации от пользователя, оставляя "основную программу" заниматься чем-то другим (§15.7.3).

15.8. Советы

[1] Используйте параллельность для снижения времени отклика или для повышения пропускной способности; §15.1.

[2] Работайте на самом высоком уровне абстракции, который можете себе позволить; §15.1.

[3] Рассматривайте процессы как альтернативы потокам; §15.1.

[4] Средства параллельности стандартной библиотеки безопасны с точки зрения типов; §15.1.

[5] Модель памяти существует для того, чтобы уберечь большинство программистов от необходимости думать на уровне архитектуры компьютера; §15.1.

[6] Модель памяти делает память выглядящей примерно такой, какой ее ожидает видеть неискушенный программист; §15.1.

[7] Атомарные переменные позволяют программировать без применения блокировок; §15.1.

[8] Оставьте программирование без блокировок экспертам; §15.1.

[9] Иногда последовательное решение проще и быстрее параллельного; §15.1.

[10] Избегайте гонок данных; §15.1, §15.2.

[11] Предпочитайте параллельные алгоритмы непосредственному использованию параллельности; §15.1, §15.7.3.

[12] `thread` представляет собой безопасный с точки зрения типов интерфейс к системному потоку; §15.2.

[13] Используйте `join()` для ожидания завершения `thread`; §15.2.

[14] Избегайте явного совместного использования данных, насколько это возможно; §15.2.

[15] Предпочитайте применение идиомы RAII явным блокировкам/разблокированиям; §15.5; [CG:CP.20].

[16] Используйте `scoped_lock` для управления мьютексами; §15.5.

[17] Используйте `scoped_lock` для захвата нескольких блокировок; §15.5; [CG:CP.21].

[18] Используйте `shared_lock` для реализации блокировок "читатель-писатель"; §15.5;

[19] Определяйте `mutex` вместе с данными, которые он защищает; §15.5; [CG:CP.50].

[20] Используйте `condition_variable` для управления связями между потоками `thread`; §15.6.

[21] Используйте `unique_lock` (а не `scoped_lock`), когда нужно копировать блокировку или необходима низкоуровневая работа с синхронизацией; §15.6.

[22] Используйте `unique_lock` (а не `scoped_lock`) при работе с `condition_variable`; §15.6.

[23] Не выполняйте ожидание без условия; §15.6; [CG:CP.42].

[24] Минимизируйте время нахождения в критической области; §15.6 [CG:CP.43].

[25] Рассуждайте в терминах заданий, которые могут выполняться параллельно, а не непосредственно в терминах потоков `thread`; §15.7.

[26] Цените простоту; §15.7.

[27] Предпочитайте применение `packaged_task` и `future` непосредственному применению `thread` и `mutex`; §15.7.

[28] Возвращайте результат с использованием `promise` и получайте его из `future`; §15.7.1; [CG:CP.60].

[29] Используйте `packaged_task` для обработки исключений, сгенерированных заданиями и для организации возврата значений; §15.7.2.

[30] Используйте `packaged_task` и `future` для выражения запроса ко внешнему сервису и ожидания его ответа; §15.7.2.

[31] Используйте `async()` для запуска простых задач; §15.7.3; [CG:CP.61].

16

История и совместимость

Поспешай медленно
(festina lente)
— Октавиан Август

16.1. История

Я изобрел C++, написал его первые определения и создал его первую реализацию. Я выбрал и сформулировал критерии проектирования для C++, раз-

работал основные языковые функции, разработал или помогал разрабатывать многие ранние библиотеки и в течение 25 лет отвечал за обработку предложений по расширению в комитете по стандартам C++.

C++ был разработан для предоставления возможностей Simula по организации программ [15] в сочетании с эффективностью и гибкостью C для системного программирования [29]. Язык программирования Simula был первоначальным источником механизмов абстракции C++. Понятие класса (с производными классами и виртуальными функциями) было заимствовано именно из него. Однако позже в C++ появились шаблоны и исключения с совсем другими источниками вдохновения.

Эволюция C++ всегда находилась в контексте его применения. Я провел много времени, слушая пользователей и выясняя мнения опытных программистов. В частности, мои коллеги из AT&T Bell Laboratories были просто необходимы для развития C++ во времена его первого десятилетия.

Этот раздел представляет собой краткий обзор; в нем мы не будем пытаться упомянуть каждую языковую функцию и библиотечный компонент. Кроме того, мы не будем углубляться в детали. Для получения дополнительной информации и, в частности, большего количества имен людей, внесших свой вклад в развитие C++, читайте две мои статьи на конференциях ACM *History of Programming Languages* [50, 54] и мою книгу *Design and Evolution of C++* [51]. В них подробно описаны дизайн и развитие C++ и задокументировано влияние на него других языков программирования.

Большинство документов, подготовленных в рамках разработки стандартов ISO C++, доступны в Интернете [64]. В своем FAQ я стараюсь поддерживать информацию о тех, кто предложил и усовершенствовал те или иные стандартные возможности C++ [56]. C++ не является работой безликого анонимного комитета или предположительно всемогущего "пожизненного диктатора"; это работа многих преданных, опытных, трудолюбивых людей.

16.1.1. Временная диаграмма развития C++

Работа, которая привела к созданию C++, началась осенью 1979 года под названием "C с классами". Далее приведена упрощенная временная диаграмма развития C++.

1979 Начата работа над "C с классами". Первоначальный набор возможностей включал классы и производные классы, открытый/закрытый доступ, конструкторы и деструкторы, а также объявления функций с проверкой аргументов. Первая библиотека поддерживала невытесняющие параллельные задания и генераторы случайных чисел

1984 "С с классами" переименован в C++. К тому времени C++ получил виртуальные функции, перегрузку функций и операторов, ссылки и библиотеки ввода-вывода и комплексных чисел

1985 Первый коммерческий выпуск C++ (14 октября). Стандартная библиотека включала потоки ввода-вывода, комплексные числа и задания (невытесняющее планирование)

1985 (14 октября) Выход в свет книги *The C++ Programming Language* [45]

1989 Выход в свет книги *The Annotated C++ Reference Manual* [18]

1991 Выход в свет книги *The C++ Programming Language, Second Edition* [49], представившей обобщенное программирование с использованием шаблонов и обработку ошибок с использованием исключений, а также идиому RAII

1997 Выход в свет книги *The C++ Programming Language, Third Edition* [52] с описанием ISO C++, включая пространства имен, `dynamic_cast` и много уточнений по поводу шаблонов. В стандартную библиотеку добавлен каркас STL с обобщенными контейнерами и алгоритмами

1998 Стандарт ISO C++ [5]

2002 Началась работа над пересмотренным стандартом, в просторечии названном C++0x

2003 Была выпущена версия стандарта ISO C++ с "исправлениями ошибок". В Техническом отчете C++ были представлены новые компоненты стандартной библиотеки, такие как регулярные выражения, неупорядоченные контейнеры (хеш-таблицы) и указатели для управления ресурсами, которые позже стали частью C++11

2006 В техническом отчете ISO C++ о производительности рассматриваются вопросы стоимости и предсказуемости применяемых методов, в основном связанные с программированием встроенных систем [6]

2011 Стандарт ISO C++11 [8]. В нем представлены унифицированная инициализация, семантика перемещения, вывод типов из инициализаторов (`auto`), цикл `for` по диапазону, шаблоны с переменным количеством аргументов, лямбда-выражения, псевдонимы типов, пригодная для параллелизма модель памяти и многое другое. В стандартную библиотеку добавлено несколько компонентов, включая потоки, блокировки и большинство компонентов из Технического отчета 2003 года

2013 Первая полная реализация C++11

2013 *The C++ Programming Language, Fourth Edition* с описанием C++11

2014 Стандарт ISO C++14 [9], дополнивший C++11 шаблонами переменных, разделителями цифр, обобщенными лямбда-выражениями и нескольки-

ми улучшениями стандартной библиотеки. Завершение первых реализаций C++14

2015 Начат проект C++ Core Guidelines [61]

2015 Одобрена техническая спецификация концептов

2017 Стандарт ISO C++17 [10] предлагает разнообразный набор новых функциональных возможностей, в том числе гарантии порядка вычислений, структурированное связывание, выражения свертки, библиотеку файловой системы, параллельные алгоритмы, а также типы `variant` и `optional`. Завершены первые реализации C++17

2017 Одобрены технические спецификации модулей и диапазонов

2020 Стандарт ISO C++20 (запланирован)

Во время разработки C++11 был известен как C++0x. Как это часто бывает в крупных проектах, мы были слишком оптимистичны в отношении даты завершения. В заключение мы пошутили, что x в C++0x означал шестнадцатеричную запись, так что C++0x стал C++0B. С другой стороны, комитет (как и основные поставщики компиляторов) вовремя представил C++14 и C++17.

16.1.2. Ранние годы

Первоначально я разработал и реализовал язык, потому что хотел распределить службы ядра UNIX по многопроцессорным и локальным сетям (которые теперь называются многоядерными и кластерными). Для этого мне нужно было точно указать части системы и то, как они взаимодействовали. Simula [15] была бы идеальной для этого, если бы не соображения производительности. Мне также нужно было иметь дело непосредственно с аппаратным обеспечением и предоставить высокопроизводительные механизмы параллельного программирования, для которых C был бы идеальным, если бы не его слабая поддержка модульности и проверки типов. Результат добавления классов в стиле Simula к C (классическому C; §16.3.1) дал "C с классами", который использовался для крупных проектов, в которых его возможности для написания программ, минимально использующих время и память, были подвергнуты жестким испытаниям. В нем отсутствовали перегрузка операторов, ссылки, виртуальные функции, шаблоны, исключения и многие другие детали [42]. Первое использование C++ вне исследовательской организации началось в июле 1983 года.

Название "C++" было придумано Риком Маскитти (Rick Mascitti) летом 1983 года и выбрано мной в качестве замены названия "C с классами". Название обозначает эволюционную природу изменений от предшественника — C, ведь "++" — это оператор инкремента в C. Немного более короткое имя "C+" является синтаксической ошибкой; кроме того, оно уже использовалось как

название языка, не связанного с C++. Ценители семантики C считают название "C++" хуже, чем "++C". Язык не был назван "D", потому что был расширением C, не пытался исправить проблемы, удаляя функциональные возможности, и уже существовало несколько потенциальных наследников C с именем "D". Еще одну интерпретацию имени "C++" вы найдете в приложении к [35].

C++ был разработан прежде всего для того, чтобы мне и моим друзьям не приходилось программировать на ассемблере, C или различных модных на то время языках высокого уровня. Его основная цель состояла в том, чтобы сделать написание хороших программ проще и приятнее для отдельного программиста. В первые годы не было никакого проекта C++ на бумаге; проектирование, документирование и реализация выполнялись одновременно. Не было ни "проекта C++", ни комитета по разработке C++. На протяжении всего времени C++ развивался, чтобы справляться с проблемами, с которыми сталкиваются пользователи, и в результате дискуссий между моими друзьями, моими коллегами и мной.

Первый дизайн C++ (тогда он назывался "C с классами") включал объявления функций с проверкой типов аргументов и неявными преобразованиями, классы с различием между `public` интерфейсом и `private` реализацией, производные классы, а также конструкторы и деструкторы. Я использовал макросы для обеспечения примитивной параметризации [42]. Все это оказалось в неэкспериментальном использовании к середине 1980 года. В конце этого года я смог представить набор языковых средств, поддерживающих согласованный набор стилей программирования. Оглядываясь назад, я считаю наиболее важным введение конструкторов и деструкторов. В терминологии того времени [41]

> *Функция "new" создает среду выполнения для функций-членов, а функция "delete" обращает все вспять.*

Вскоре после этого "функция new" и "функция delete" были переименованы в "конструктор" и "деструктор". В этом — корень стратегий C++ по управлению ресурсами (вызвавших потребность в исключениях) и ключ ко многим технологиям, делающим код пользователя коротким и понятным. Если в то время и были другие языки, которые поддерживали несколько конструкторов, способных выполнять общий код, я их не знал (и сейчас не знаю). Деструкторы были новинкой C++.

C++ был выпущен в продажу в октябре 1985 года. К тому времени я добавил встраивание (§1.3, §4.2.1), `const` (§1.6), перегрузку функций (§1.3), ссылки (§1.7), перегрузку операторов (§4.2. 1) и виртуальные функции (§4.4). Из этих возможностей, безусловно, наиболее противоречивой была поддержка полиморфизма времени выполнения в форме виртуальных функций. Я знал

о его ценности от Simula, но оказалось невозможным убедить в его ценности большинство людей в мире системного программирования. Системные программисты склонны рассматривать косвенные вызовы функций с подозрением, а людям, знакомым с другими языками, поддерживающими объектно-ориентированное программирование, было трудно поверить, что виртуальные функции могут быть достаточно быстрыми, чтобы быть полезными в системном коде. И наоборот, многим программистам с объектно-ориентированным опытом пришлось привыкнуть (а многие до сих пор испытывают трудности) к идее, что вы используете вызовы виртуальных функций только для выражения выбора, который должен быть сделан во время выполнения. Сопротивление виртуальным функциям может быть связано с сопротивлением идее о том, что вы можете получить лучшие системы с помощью более регулярной структуры кода, поддерживаемой языком программирования. Многие программисты на C, похоже, убеждены, что на самом деле важны полная гибкость и тщательный индивидуальный подход к каждой детали программы. Мое мнение было (и остается) таким, что нам нужна любая помощь, которую мы можем получить от языков и инструментов: внутренняя сложность систем, которые мы пытаемся построить, всегда находится на грани того, что мы можем выразить.

В ранних документах (например, в [44] и [51]) C++ описывался следующим образом:

C++ — язык программирования общего назначения, который

- *лучше языка программирования C;*
- *поддерживает абстракцию данных;*
- *поддерживает объектно-ориентированное программирование.*

Обратите внимание: *не* "C++ является объектно-ориентированным языком программирования". Здесь "поддерживает абстракцию данных" означает сокрытие информации, классы, не являющиеся частью иерархии классов, и обобщенное программирование. Изначально обобщенное программирование поддерживалось очень слабо, с использованием макросов [42]. Шаблоны и концепты появились намного позже.

Большая часть дизайна C++ была разработана на флипчартах в кабинетах моих коллег. В первые годы были неоценимыми отзывы Стю Фельдмана (Stu Feldman), Александра Фрейзера (Alexander Fraser), Стива Джонсона (Steve Johnson), Брайана Кернигана (Brian Kernighan), Дуга Макилроя (Doug McIlroy) и Денниса Ритчи (Dennis Ritchie).

Во второй половине 1980-х годов я продолжал добавлять языковые возможности в ответ на комментарии пользователей. Наиболее важными из них были шаблоны [48] и обработка исключений [32], которые на момент начала

разработки стандарта считались экспериментальными. При разработке шаблонов я был вынужден выбирать между гибкостью, эффективностью и ранней проверкой типов. В то время никто не знал, как одновременно получить все три свойства. Я чувствовал, что, чтобы конкурировать с кодом в стиле C для требовательных системных приложений, я должен был выбрать первые два свойства. Оглядываясь назад, я думаю, что выбор был правильным, а поиск лучшей проверки типов шаблонов продолжается [17, 24, 63, 57]. Проектирование исключений было сосредоточено на многоуровневом распространении исключений, передаче произвольной информации обработчику ошибок и интеграции исключений и управления ресурсами с использованием локальных объектов с деструкторами для представления и освобождения ресурсов. Я неуклюже назвал эту важнейшую технику *Захват ресурса есть инициализация* (Resource Acquisition Is Initialization), а другие вскоре сократили ее до аббревиатуры RAII (§4.2.2).

Я обобщил механизмы наследования C++ для поддержки нескольких базовых классов [46]. Это называлось *множественным наследованием* и считалось трудным и противоречивым. Я считал эту возможность гораздо менее важной, чем шаблоны или исключения. Многократное наследование абстрактных классов (часто называемых *интерфейсами*) в настоящее время широко распространено в языках, поддерживающих статическую проверку типов и объектно-ориентированное программирование.

Язык C++ развивался рука об руку с некоторыми из ключевых библиотечных возможностей. Например, я разработал классы комплексных чисел [43], векторов, стеков и потоков ввода-вывода [44] вместе с механизмами перегрузки операторов. Первые классы строк и списков были разработаны Джонатаном Шопиро (Jonathan Shopiro) и мной в рамках одной и той же работы. Классы строк и списков Джонатана были первыми классами, широко используемыми как часть библиотеки. Строковый класс стандартной библиотеки C++ вырос из этих ранних попыток. Библиотека заданий, описанная в [47], была частью первой программы "C с классами", написанной в 1980 году. В ней были сопрограммы и планировщик. Я написал его и связанные с ним классы для поддержки симуляций в стиле Simula. К сожалению, чтобы получить стандартизированную и общедоступную поддержку параллелизма (глава 15, "Параллельные вычисления"), нам пришлось ждать до 2011 года (30 лет!). Сопрограммы, вероятно, будут частью C++20 [12]. На разработку шаблонов повлияли различные шаблоны `vector`, `map`, `list` и `sort`, разработанные Эндрю Кенигом (Andrew Koenig), Алексом Степановым (Alex Stepanov), мной и другими программистами.

Самым важным нововведением в стандартной библиотеке 1998 года была библиотека STL, каркас алгоритмов и контейнеров (глава 11, "Контейнеры",

и 12, "Алгоритмы"). Это была работа Алекса Степанова (Alex Stepanov, с Дэйвом Массером (Dave Musser), Мэн Ли (Meng Lee) и др.), основанная на более чем десятилетней работе над обобщенным программированием. STL оказалась чрезвычайно влиятельной библиотекой как в сообществе C++, так и за его пределами.

C++ вырос в среде с множеством устоявшихся и экспериментальных языков программирования (например, Ada [27], Algol 68 [66] и ML [36]). В то время я был хорошо знаком примерно с 25 языками, и их влияние на C++ описано в [51] и [54]. Однако определяющее влияние всегда исходило от приложений, с которыми мне приходилось сталкиваться. Это была целенаправленная политика, нацеленная на то, чтобы разработка C++ была "управляемая проблемами", а не подражательная.

16.1.3. Стандарты ISO C++

Взрывной рост использования C++ вызвал некоторые изменения. В 1987 году стало ясно, что неизбежна формальная стандартизация C++ и что нам нужно начать готовить почву для усилий по стандартизации [51]. Результатом стало сознательное стремление поддерживать связь между разработчиками компиляторов C++ и их основными пользователями. Это было сделано с помощью бумажной и электронной почты и личных встреч на конференциях по C++ и в других местах.

AT&T Bell Labs внесла большой вклад в развитие C++, позволив мне поделиться проектами пересмотренных версий справочного руководства по C++ с разработчиками и пользователями. Поскольку многие из этих людей работали в компаниях, которые могут рассматриваться как конкурирующие с AT&T, значение этого вклада не следует недооценивать. Менее просвещенная компания могла бы вызвать серьезные проблемы фрагментации языка, просто ничего не делая. Как оказалось, около ста человек из десятков организаций прочитали и прокомментировали то, что стало общепринятым справочным руководством и базовым документом для усилий по стандартизации ANSI C++. Их имена можно найти в [18]. Комитет X3J16 ANSI был созван в декабре 1989 года по инициативе Hewlett-Packard. В июне 1991 года эта ANSI (американская национальная) стандартизация C++ стала частью ISO (международной) стандартизации C++. Комитет ISO C++ называется "WG21". С 1990 года эти совместные комитеты по стандартам C++ стали основным форумом для развития C++ и уточнения его определения. Я участвовал в этих комитетах повсюду. В частности, будучи председателем рабочей группы по расширениям (позже названной группой эволюции) с 1990 по 2014 год, я непосредственно отвечал за обработку предложений об основных изменениях в C++ и добавление новых языковых функций. Первоначальный проект стан-

дарта для публичного рассмотрения был выпущен в апреле 1995 года. Первый стандарт ISO C++ (ISO/IEC 14882-1998) [5] был ратифицирован голосованием 22:0 в 1998 году. Выпуск "исправления ошибок" этого стандарта вышел в 2003 году, поэтому иногда можно услышать, что люди ссылаются на C++03, но, по сути, — это тот же язык, что и C++98.

C++11, в течение многих лет известный как C++0x, представляет собой работу членов WG21. Комитет работал все более и более напряженно, и это, вероятно, и привело к лучшей (и более строгой) спецификации (но и к ограничению количества новинок) [54]. Первоначальный проект стандарта для публичного рассмотрения был выпущен в 2009 году. Второй стандарт ISO C++ (ISO/IEC 14882-2011) [8] был ратифицирован голосованием 21:0 в августе 2011 года.

Одна из причин длительного разрыва между этими двумя стандартами заключается в том, что у большинства членов комитета (включая меня) сложилось ошибочное впечатление, что правила ISO требуют "периода ожидания" между выпуском стандарта и началом работы над новыми возможностями языка. Поэтому серьезная работа над новыми языковыми возможностями началась только в 2002 году. Другие причины включали увеличение размера современных языков и их базовых библиотек. В терминах страниц текста стандарта язык вырос примерно на 30%, а стандартная библиотека — примерно на 100%. Большая часть увеличения произошла не из-за новой функциональности, а из-за более подробной спецификации. Кроме того, работа над новым стандартом C++, очевидно, должна была быть очень осторожной, чтобы не скомпрометировать старый код из-за несовместимых изменений. Есть миллиарды строк кода на C++, которые комитет нарушать не должен. Стабильность на протяжении десятилетий — это важная "возможность" языка.

C++11 значительно увеличил стандартную библиотеку и способствовал завершению разработки набора функциональных возможностей, необходимых для стиля программирования, который представляет собой синтез парадигм и идиом, оказавшихся успешными в C++98.

Основными целями C++11 были:

- сделать C++ лучшим языком для системного программирования и построения библиотек;
- сделать C++ проще для обучения и изучения.

Эти цели документированы и детально изложены в [54].

Основные усилия были направлены на то, чтобы сделать параллельное системное программирование безопасным с точки зрения типов и переносимым. Это включало модель памяти (§15.1) и поддержку программирования без блокировок. Это была работа Ганса Бома (Hans Boehm), Брайана Макнай-

та (McKnight) и других участников рабочей группы по параллелизму. Кроме того, мы добавили библиотеку потоков `thread`.

После C++11 было достигнуто общее согласие о том, что 13 лет между стандартами — это слишком много. Герб Саттер (Herb Sutter) предложил комитету принять политику принятия стандартов с фиксированными интервалами времени, "модель поезда". Я настойчиво приводил доводы в пользу короткого интервала между стандартами, чтобы минимизировать вероятность задержек (потому что кое-кто настаивал на дополнительном времени, чтобы позволить включить "еще одну существенную деталь"). Мы договорились об амбициозном трехлетнем графике с идеей, что мы должны чередовать второстепенные и основные выпуски стандарта.

C++14 преднамеренно был сделан второстепенным выпуском, целью которого было "завершение C++11". Это отражает ту реальность, что при фиксированной дате выпуска всегда будут находиться функциональные возможности языка, которые нам нужны, но которые не готовы вовремя. Кроме того, после широкого применения неизбежно будут обнаружены пробелы в наборе функциональных возможностей.

Чтобы обеспечить более быстрое развитие и параллельную разработку независимых возможностей и лучше использовать энтузиазм и навыки многих добровольцев, комитет использует механизмы ISO для разработки и публикации "Технических спецификаций" (Technical Specification — TS). Похоже, это неплохое решение для компонентов стандартной библиотеки, хотя может привести к большему количеству этапов в процессе разработки, а следовательно, к задержкам. Что касается языковых возможностей, то здесь TS, кажется, работают не так хорошо. Возможно, причина в том, что немногие существенные языковые функции действительно независимы, потому что работа по разработке формулировок стандартов не сильно различается между стандартом и TS и экспериментировать с реализациями компилятора может меньшее количество программистов.

C++17 должен был стать основным выпуском. Под "основным" я подразумеваю наличие функций, которые изменят наши взгляды на дизайн и структуру нашего программного обеспечения. По этому определению C++17 был в лучшем случае средним выпуском. Он включал в себя множество незначительных расширений, но функции, которые внесли бы существенные изменения (например, концепты, модули и сопрограммы), либо были не готовы, либо погрязли в противоречиях и отсутствии направленного проектирования. В результате C++17 включает в себя всего понемногу, но ничего такого, что существенно изменило бы жизнь программиста на C++, который уже усвоил уроки C++11 и C++14. Я надеюсь, что C++20 станет обещанной и столь необходимой крупной версией и что основные новинки станут широко доступ-

ны задолго до 2020 года. Опасности заключаются в "проектировании комитетом" — в раздувании функциональных возможностей, отсутствии согласованного стиля и близоруких решениях. В комитете с более чем ста членами, присутствующими на каждом собрании, и большим количеством удаленных участников по сети такие нежелательные явления почти неизбежны. Продвигаться к более простому и последовательному языку очень сложно.

16.1.4. Стандарты и стиль

Стандарт говорит о том, что будет работать и как именно. Он ничего не говорит о хорошем и эффективном использовании имеющихся возможностей. Существуют значительные различия между пониманием технических деталей возможностей языка программирования и их эффективным использованием в сочетании с другими возможностями, библиотеками и инструментами для создания более качественного программного обеспечения. Под "более качественным" я имею в виду "более легко обслуживаемое, менее подверженное ошибкам и более быстрое". Нам нужно разрабатывать, популяризировать и поддерживать согласованные стили программирования. Кроме того, мы должны поддержать эволюцию старого кода в более современные, эффективные и согласованные стили.

С ростом языка и его стандартной библиотеки проблема популяризации эффективных стилей программирования стала критической. Очень трудно заставить большие группы программистов отказаться от чего-то, что и так работает, в пользу чего-то лучшего. До сих пор есть люди, которые считают C++ незначительным дополнением к C, и люди, которые считают объектно-ориентированные стили программирования 1980-х годов, основанные на массивной иерархии классов, вершиной развития. Многие пытаются использовать C++11 в средах с большим количеством старого кода C++. С другой стороны, есть также много тех, кто с энтузиазмом использует новые возможности. Например, некоторые программисты убеждены, что истинным C++ является только код, использующий огромное количество шаблонного метапрограммирования.

Так что же такое *современный* C++? В 2015 году я намеревался ответить на этот вопрос, разработав набор советов по кодированию, с поддержкой их ясно сформулированными обоснованиями. Вскоре я обнаружил, что я не одинок в решении этой проблемы, и вместе с программистами из разных частей мира, в частности, из Microsoft, Red Hat и Facebook, мы начали проект "Базовые рецепты C++" [61]. Это амбициозный проект, направленный на обеспечение полной безопасности с точки зрения типов и полной безопасности ресурсов как основы для более простого, быстрого и более удобного для поддержки кода [62]. Конкретные рекомендации по кодированию, сопровождающиеся

обоснованиями, мы подкрепляем инструментами статического анализа и небольшой библиотекой поддержки. Я считаю, что нечто подобное необходимо для существенного продвижения сообщества C++ в направлении активного использования преимуществ современных языковых возможностей, библиотек и вспомогательных инструментов.

16.1.5. Использование C++

В настоящее время C++ — очень широко используемый язык программирования. Количество его пользователей быстро увеличилось с одного в 1979 году до 400 тысяч в 1991 году, т.е. число пользователей удваивалось примерно каждые 7,5 месяца в течение более десяти лет. Естественно, с момента первоначального всплеска роста темпы роста замедлились, но, по моим оценкам, в 2018 году имеется около 4,5 миллиона программистов на C++ [28]. Большая часть этого роста приходится на время после 2005 года, когда экспоненциальный рост скорости процессора прекратился, так что очень важным фактором стала производительность языка. Этот рост был достигнут без какого-либо маркетинга или организованного сообщества пользователей.

C++ — язык, прежде всего, индустриальный, т.е. более заметен в производственной сфере, чем в образовании или при изучении языков программирования. Он вырос в Bell Labs, инспирированный разнообразными жесткими потребностями в области телекоммуникаций и системного программирования (включая драйверы устройств, сети и встроенные системы). Оттуда использование C++ распространилось практически на все отрасли: микроэлектроника, веб-приложения и инфраструктура, операционные системы, финансовые, медицинские, автомобильные, аэрокосмические приложения, физика высоких энергий, биология, производство энергии, машинное обучение, видеоигры, графика, анимация, виртуальная реальность и многое другое. Он в основном используется там, где стоящие перед разработчиками задачи требуют сочетания способности эффективно использовать оборудование и при этом управлять сложностью проекта. Множество применений C++ постоянно растет [50, 60].

16.2. Эволюция возможностей C++

Здесь я перечисляю возможности языка и компоненты стандартной библиотеки, которые были добавлены в C++ в стандартах C++11, C++14 и C++17.

16.2.1. Возможности языка C++11

Такой список возможностей языка может быть сложным для восприятия. Помните, что ни одна языковая возможность не предназначена для использования изолированно от других. В частности, большинство функциональных

возможностей, которые являются новинками C++11, не имеют смысла в изоляции от инфраструктуры, предоставляемой более старыми функциональными возможностями языка.

[1] Унифицированная и общая инициализация с использованием списков `{}` (§1.4, §4.2.3)

[2] Вывод типов из инициализаторов: `auto` (§1.4)

[3] Предотвращение сужения (§1.4)

[4] Обобщенные и гарантированные константные выражения: `constexpr` (§1.6)

[5] Цикл `for` для диапазона (§1.7)

[6] Ключевое слово для нулевого указателя: `nullptr` (§1.7)

[7] Строго типизированные перечисления с областью видимости: `enum class` (§2.5)

[8] Проверки времени компиляции: `static_assert` (§3.5.5)

[9] Отображение `{}`-списков на `std::initializer_list` на уровне языка (§4.2.3)

[10] Ссылки на r-значения, обеспечивающие семантику перемещения (§5.2.2)

[11] Вложенные аргументы шаблонов завершаются >> (не требуется пробел между символами >)

[12] Лямбда-выражения (§6.3.2)

[13] Шаблоны с переменным количеством параметров (§7.4)

[14] Псевдонимы типов и шаблонов (§6.4.2)

[15] Символы Unicode

[16] Целочисленный тип `long long`

[17] Управление выравниванием: `alignas` и `alignof`

[18] Возможность использовать тип выражения как тип в объявлении: `decltype`

[19] Необработанные строковые литералы (§9.4)

[20] Обобщенные POD (Plain Old Data — обычные старые данные)

[21] Обобщенные `union`

[22] Локальные классы в качестве аргументов шаблонов

[23] Синтаксис указания возвращаемого типа в виде суффикса

[24] Синтаксис атрибутов и два стандартных атрибута: `[[carries_dependency]]` и `[[noreturn]]`

[25] Предотвращение распространения исключений: спецификатор `noexcept` (§3.5.1)

[26] Проверка возможности генерации исключений в выражениях: оператор `noexcept`

[27] Возможности C99: расширенные целочисленные типы (например, правила для необязательных более длинных целочисленных типов); конкатенация узких/широких строк; `__STDC_HOSTED__`; `_Pragma(X)`; макросы с переменным количеством аргументов и пустые аргументы макросов

[28] `__func__` как имя строки, хранящей имя текущей функции

[29] `inline` пространства имен

[30] Делегирование конструкторов

[31] Инициализаторы членов в классе (§5.1.3)

[32] Управление генерируемыми функциями: `default` и `delete` (§5.1.1)

[33] Операторы явного преобразования типов

[34] Пользовательские литералы (§5.4.4)

[35] Более явное управление инстанцированием `template`: `extern template`

[36] Аргументы шаблонов по умолчанию для шаблонов функций

[37] Наследование конструкторов

[38] Управление перекрытием: `override` и `final` (§4.5.1)

[39] Более простое и более общее правило SFINAE (Substitution Failure Is Not An Error — ошибка подстановки ошибкой не является)

[40] Модель памяти (§15.1)

[41] Память, локальная для потока: `thread_local`

Более полное описание изменений C++11 по сравнению с C++98 приведено в [59].

16.2.2. Возможности языка C++14

[1] Вывод возвращаемого типа функции; §3.6.2

[2] Усовершенствованные `constexpr`-функции, например разрешен цикл `for` (§1.6)

[3] Шаблоны переменных (§6.4.1)

[4] Бинарные литералы (§1.4)

[5] Разделители цифр (§1.4)

[6] Обобщенные лямбда-выражения (§6.3.3)

[7] Более общий захват у лямбда-выражений

[8] Атрибут `[[deprecated]]`

[9] Несколько более мелких расширений

16.2.3. Возможности языка C++17

[1] Гарантированное исключение копирования (§5.2.2)

[2] Динамическое распределение сверхвыровненных типов

[3] Более строгий порядок вычислений (§1.4)

[4] Литералы UTF-8 (`u8`)

[5] Шестнадцатеричные литералы с плавающей точкой

[6] Выражения свертки (§7.4.1)

[7] Обобщенные значения аргументов шаблонов (параметры шаблонов `auto`)

[8] Вывод типа аргумента шаблона класса (§6.2.3)

[9] `if` времени компиляции (§6.4.3)

[10] Инструкции выбора с инициализаторами (§1.8)

[11] `constexpr` лямбда-выражения

[12] `inline` переменные

[13] Структурное связывание (§3.6.3)

[14] Новые стандартные атрибуты: `[[fallthrough]]`, `[[nodiscard]]` и `[[maybe_unused]]`

[15] Тип `std::byte`

[16] Инициализация `enum` значением его базового типа (§2.5)

[17] Несколько более мелких расширений

16.2.4. Компоненты стандартной библиотеки C++11

Дополнения C++11 к стандартной библиотеке представлены в двух видах: новые компоненты (такие, как библиотека сопоставления регулярным выражениям) и улучшения компонентов C++98 (такие, как конструкторы перемещения для контейнеров).

[1] Конструкторы от `initializer_list` для контейнеров (§4.2.3)

[2] Семантика перемещения для контейнеров (§5.2.2, §11.2)

[3] Односвязный список: `forward_list` (§11.6)

[4] Хеш-контейнеры: `unordered_map`, `unordered_multimap`, `unordered_set` и `unordered_multiset` (§11.6, §11.5)

[5] Указатели для управления ресурсами: `unique_ptr`, `shared_ptr` и `weak_ptr` (§13.2.1)

[6] Поддержка параллельности: `thread` (§15.2), мьютексы (§15.5), блокировки (§15.5) и условные переменные (§15.6)

[7] Высокоуровневая поддержка параллельности: `packaged_thread`, `future`, `promise` и `async()` (§15.7)

[8] `tuple` (§13.4.3)

[9] Регулярные выражения: `regex` (§9.4)

[10] Случайные числа: распределения и генераторы (§14.5)

[11] Имена целочисленных типов, такие как `int16_t`, `uint32_t` и `int_fast64_t`

[12] Контейнер фиксированного размера со смежными элементами последовательности: `array` (§13.4.1)

[13] Копирование и регенерация исключений (§15.7.1)

[14] Сообщение об ошибках с помощью кодов ошибок: `system_error`

[15] Операции `emplace()` для контейнеров (§11.6)

[16] Широкое применение `constexpr`-функций

[17] Систематическое применение `noexcept`-функций

[18] Усовершенствованные функциональные адаптеры: `function` и `bind()` (§13.8)

[19] Преобразования `string` в числовые значения

[20] Аллокаторы с областями видимости

[21] Свойства типов, такие как `is_integral` и `is_base_of` (§13.9.2)

[22] Утилиты для работы со временем: `duration` и `time_point` (§13.7)

[23] Рациональная арифметика времени компиляции: `ratio`

[24] Завершение процесса: `quick_exit`

[25] Большее количество алгоритмов, таких как `move()`, `copy_if()` и `is_sorted()` (глава 12, “Алгоритмы”)

[26] API сборки мусора (§5.3)

[27] Низкоуровневая поддержка параллельности: `atomic`

16.2.5. Компоненты стандартной библиотеки C++14

[1] `shared_mutex` (§15.5)

[2] Пользовательские литералы (§5.4.4)

[3] Доступ к кортежам по типу (§13.4.3)

[4] Гетерогенный поиск в ассоциативном контейнере

[5] Несколько более мелких возможностей

16.2.6. Компоненты стандартной библиотеки C++17

[1] Файловая система (§10.10)

[2] Параллельные алгоритмы (§12.9, §14.3.1)

[3] Математические специальные функции (§14.2)

[4] `string_view` (§9.3)

[5] `any` (§13.5.3)

[6] `variant` (§13.5.1)

[7] `optional` (§13.5.2)

[8] `invoke()`

[9] Элементарные преобразования строк: `to_chars` и `from_chars`

[10] Полиморфный аллокатор (§13.6)

[11] Несколько более мелких расширений

16.2.7. Удаленные и нерекомендуемые возможности

За время существования языка C++ на нем написаны миллиарды работающих строк, и никто точно не знает, какие именно возможности критичны для их работы. Поэтому комитет ISO удаляет старые функциональные возможности языка крайне неохотно и после многих лет предупреждения. Однако иногда проблемные функциональные возможности все же удаляются.

- C++17 окончательно удалил спецификацию исключений:

  ```
  void f() throw(X,Y); // Начиная с C++98; теперь это ошибка
  ```

  Соответственно, удалены средства поддержки спецификаций исключений — `unexcepted_handler`, `set_unexpected()`, `get_unexpected()` и `unexpected()`. Используйте вместо этого `noexcept` (§3.5.1).
- Больше не поддерживаются триграфы.
- `auto_ptr` не рекомендован к употреблению. Вместо него используйте `unique_ptr` (§13.2.1).
- Удалено использование спецификатора `register`.
- Удалено применение оператора ++ к типу `bool`.
- Функциональная возможность C++98 `export` удалена из-за сложности и нереализованности основными производителями. Вместо этого `export` используется как ключевое слово для модулей (§3.3).

- Генерация операций копирования не рекомендована к применению для классов с деструкторами (§5.1.1).
- Удалено присваивание строкового литерала переменной типа `char*`. Вместо этого используйте `const char*` или `auto`.
- Некоторые функциональные объекты стандартной библиотеки C++ и связанные с ними функции не рекомендованы к употреблению. Большинство из них относится к связыванию аргументов. Вместо этого используйте лямбда-выражения и `function` (§13.8).

Объявляя некоторую функциональную возможность не рекомендованной к употреблению, комитет по стандартам выражает желание, чтобы эта функция исчезла. Тем не менее комитет не уполномочен немедленно удалять интенсивно используемую функцию — какой бы избыточной или опасной она ни была. Таким образом, нерекомендованность к употреблению является сильным намеком, что этой возможности следует всячески избегать. Она может исчезнуть в будущем. Компиляторы могут выдавать предупреждения по поводу использования устаревших функциональных возможностей. Однако эти возможности являются частью стандарта, а история показывает, что, как правило, из соображений совместимости они остаются поддерживаемыми "навсегда".

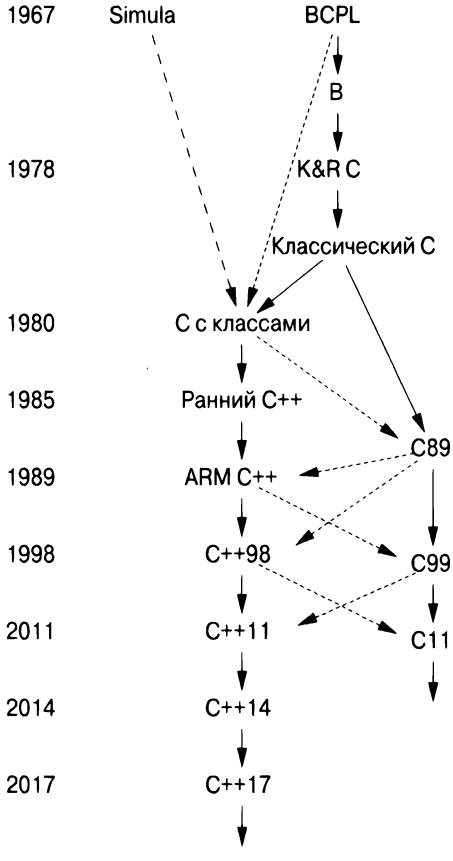
16.3. Совместимость C/C++

С небольшими исключениями C++ является надмножеством C (имеется в виду C11 [4]). Большинство различий связаны с бо́льшим акцентом C++ на проверке типов. Хорошо написанные программы на C, как правило, одновременно являются программами на C++; компилятор может диагностировать все различия между C++ и C. Несовместимости C99/C++11 перечислены в Приложении C стандарта.

16.3.1. C и C++ — братья

У классического C два основных потомка: ISO C и ISO C++. За прошедшие годы и десятилетия эти языки развивались с разной скоростью и в разных направлениях. Одним из результатов является то, что каждый язык обеспечивает поддержку традиционного программирования в стиле C немного по-своему. Получаемая несовместимость может сделать несчастными программистов, которые используют и C, и C++, для тех, кто пишет на одном языке программирования с использованием библиотек, реализованных на другом, а также для разработчиков библиотек и инструментов для C и C++.

Почему я назвал C и C++ братьями? Давайте посмотрим на упрощенное генеалогическое дерево.



Сплошная линия означает массовое наследование элементов, штриховая линия с длинными штрихами — заимствование основных функциональных возможностей, а штриховая линия с обычными штрихами — заимствование незначительных элементов. Здесь ISO C и ISO C++ выглядят как два главных потомка K&R C [29] — как братья. Каждый из них несет в себе ключевые аспекты классического C, и ни один из них не совместим на 100% с классическим C. Я взял термин "классический C" с наклейки, прикрепленной к терминалу Денниса Ритчи. Это K&R C плюс перечисления и присваивание `struct`. BCPL определен в [38], а C89 — в [2].

Обратите внимание, что различия между C и C++ не обязательно являются результатом изменений в C, сделанных в C++. В некоторых случаях несовместимость возникает из-за несовместимого переноса в C функциональных возможностей, длительное время работавших в C++. Примерами являются возможность присваивания `T*` переменной типа `void*` и компоновка глобальных констант [53]. Иногда функциональная возможность могла быть принята

в С несовместимым образом даже после того, как она стала частью стандарта ISO C++ — например, детали значения спецификатора `inline`.

16.3.2. Проблемы совместимости

Имеется немало незначительных несовместимостей между С и C++. Все они могут вызвать проблемы у программиста, но все они могут быть решены в контексте C++. В крайнем случае фрагменты кода на С могут быть скомпилированы компилятором С и скомпонованы с использованием механизма `extern "C"`.

Вероятно, основными проблемами при преобразовании С-программы в программу на C++ являются следующие.

- Неоптимальные дизайн и стиль программирования.
- Неявное преобразование `void*` в `T*` (т.е. преобразование без явного указания оператора приведения).
- Ключевые слова C++, такие как `class` и `private`, используются в коде С как идентификаторы.
- Несовместимая компоновка фрагментов кода, скомпилированных как С, и фрагментов кода, скомпилированных как C++.

16.3.2.1. Проблемы стиля

Естественно, программа на С пишется в стиле С, таком, например, как стиль, используемый в K&R [30]. Это подразумевает широкое использование указателей и массивов и, возможно, множества макросов. Эти средства трудно надежно использовать в большой программе. Управление ресурсами и обработка ошибок часто оказываются приспосабливаемыми к конкретным условиям, основанными на документировании (а не на поддержке языком и инструментами), и часто не полностью документированы и плохо соблюдаются. Простое построчное преобразование программы на С в программу на C++ даст программу, которая часто оказывается лучше проверенной. Но на самом деле я никогда не преобразовывал программу на С в программу на C++, не обнаружив какой-нибудь ошибки. Тем не менее фундаментальная структура такой программы остается неизменной, как и основные источники ошибок. Если у вас была неполная обработка ошибок, утечки ресурсов или переполнения буфера в исходной программе на С, они все равно будут присутствовать в версии на C++. Чтобы получить преимущества от применения C++, вы должны внести изменения в саму структуру кода.

[1] Не рассматривайте C++ просто как С с некоторыми добавленными возможностями. C++ может быть использован таким образом, но это его использование будет неоптимальным. Чтобы получить реальные преи-

мущества от применения C++ по сравнению с C, необходимы иные дизайн и стили реализации.

[2] Используйте стандартную библиотеку C++ как преподаватель новых методов и стилей программирования. Обращайте внимание на отличие от стандартной библиотеки C (например, применение = вместо `strcpy()` для копирования и == вместо `strcmp()` для сравнения строк).

[3] Подстановка макросов в C++ почти никогда не требуется. Используйте `const` (§1.6), `constexpr` (§1.6), `enum` или `enum class` (§2.5) для определения констант, `inline` (§4.2.1), чтобы избежать накладных расходов на вызовы функций, шаблоны (глава 6, "Шаблоны") для определения семейств функций и типов, а также пространства имен (§3.4), чтобы избежать коллизий имен.

[4] Не объявляйте переменную до того, как она понадобится, и немедленно ее инициализируйте. Объявление может находиться везде, где может быть инструкция (§1.8), в инициализаторах циклов `for` (§1.7) и в условиях (§4.5.2).

[5] Не используйте `malloc()`. Оператор `new` (§4.2.2) выполняет те же действия лучше, а вместо `realloc()` попробуйте применить `vector` (§4.2.3, §12.1). Но не делайте простую бездумную замену `malloc()` и `free()` "голыми" операторами `new` и `delete` (§4.2.2).

[6] Избегайте `void*`, объединений и приведений, кроме как глубоко в реализации некоторой функции или класса. Их использование ограничивает поддержку, которую вы можете получить от системы типов, и может нанести ущерб производительности. В большинстве случаев приведение является признаком ошибки проектирования.

[7] Если вы должны использовать явное преобразование типов, прибегните к соответствующему именованному приведению (например, `static_cast`; §16.2.7) для более точного определения того, что вы пытаетесь сделать.

[8] Минимизируйте использование массивов и строк в стиле C. Строки `string` стандартной библиотеки C++ (§9.2), массивы `array` (§13.4.1) и векторы `vector` (§11.2) часто могут использоваться для написания более простого и более удобного в обслуживании кода по сравнению с традиционным стилем C. В общем случае старайтесь не создавать то, что уже предоставлено стандартной библиотекой.

[9] Избегайте арифметики указателей, за исключением очень специализированного кода (такого, как диспетчер памяти) и простого обхода массива (например, `++p`).

[10] Не думайте, что что-то, кропотливо написанное в стиле C (без учета таких особенностей C++, как классы, шаблоны и исключения), будет более эффективным, чем более короткая альтернатива (например, с использованием стандартных библиотечных средств). Часто (но, конечно, не всегда), верно обратное.

16.3.2.2. void*

В C `void*` может использоваться как правый операнд присваивания или инициализации переменной указателя любого типа; в C++ это может быть не так. Например:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int)); /* Не C++; в C++ выделяйте */
    // ...                          /* память с помощью new    */
}
```

Это, пожалуй, самая трудная несовместимость, с которой приходится иметь дело. Обратите внимание, что неявное преобразование `void*` в другой тип указателя в общем случае *не* является безвредным:

```
char ch;
void* pv = &ch;
int*  pi = pv;  // Не C++
*pi = 666;      // Перезапись ch и байтов рядом с ним
```

В обоих языках приводите результат `malloc()` к верному типу. Если вы работаете только с C++, избегайте `malloc()`.

16.3.2.3. Компоновка

C и C++ могут быть реализованы с использованием различных соглашений о связывании. Основной причиной этого является больший акцент на проверку типов в C++. Практическая же причина в том, что C++ поддерживает перегрузку, поэтому могут быть две глобальные функции с одним именем `open()`. Это должно быть отражено в способе работы компоновщика.

Чтобы обеспечить функции на C++ связывание в стиле C (чтобы ее можно было вызывать из фрагмента программы на C) или разрешить вызов функции C из фрагмента программы на C++, объявите ее как `extern "C"`. Например:

```
extern "C" double sqrt(double);
```

Теперь `sqrt(double)` может быть вызвана из фрагмента кода на C или C++. Определение `sqrt(double)` также может быть скомпилировано как функция C или как функция C++.

Только одна функция с заданным именем в области видимости может иметь связывание C (поскольку C не допускает перегрузку функций). Специ-

фикация связывания не влияет на проверку типов, поэтому правила C++ для вызовов функций и проверки аргументов применяются к функции, объявленной как `extern "C"`, по-прежнему.

16.4. Список литературы

1. *The Boost Libraries: free peer-reviewed portable C++ source libraries.* www.boost.org.
2. X3 Secretariat: *Standard – The C Language.* X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.
3. ISO/IEC 9899. *Standard – The C Language.* X3J11/90-013-1999.
4. ISO/IEC 9899. *Standard – The C Language.* X3J11/90-013-2011.
5. ISO/IEC JTC1/SC22/WG21 (editor: Andrew Koenig): *International Standard – The C++ Language.* ISO/IEC 14882:1998.
6. ISO/IEC JTC1/SC22/WG21 (editor: Lois Goldtwaite): *Technical Report on C++ Performance.* ISO/IEC TR 18015:2004(E)
7. *International Standard – Extensions to the C++ Library to Support Mathematical Special Functions.* ISO/IEC 29124:2010.
8. ISO/IEC JTC1/SC22/WG21 (editor: Pete Becker): *International Standard – The C++ Language.* ISO/IEC 14882:2011.
9. ISO/IEC JTC1/SC22/WG21 (editor: Stefanus du Toit): *International Standard – The C++ Language.* ISO/IEC 14882:2014.
10. ISO/IEC JTC1/SC22/WG21 (editor: Richard Smith): *International Standard – The C++ Language.* ISO/IEC 14882:2017.
11. ISO/IEC JTC1/SC22/WG21 (editor: Gabriel Dos Reis): *Technical Specification: C++ Extensions for Concepts.* ISO/IEC TS 19217:2015.
12. ISO/IEC JTC1/SC22/WG21 (editor: Gor Nishanov): *Technical Specification: C++ Extensions for Coroutines.* ISO/IEC TS 22277:2017.
13. *Описание языка C++ и возможностей стандартной библиотеки:* `www.cppreference.com`.
14. Russ Cox: *Regular Expression Matching Can Be Simple And Fast.* January 2007. `swtch.com/~rsc/regexp/regexp1.html`.
15. O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.
16. D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs.* 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.

17. Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.
18. Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.
19. J. Daniel Garcia and B. Stroustrup: *Improving performance and maintainability through refactoring in C++11*. Isocpp.org. August 2015. `http://www.stroustrup.com/improving_garcia_stroustrup_2015.pdf`.
20. G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: *A Contract Design*. P0380R1. 2016-7-11.
21. G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: *Support for contract based programming in C++*. P0542R4. 2018-4-2.
22. Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.
23. N. MacIntosh (Editor): *Guidelines Support Library*. `https://github.com/microsoft/gsl`.
24. Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.
25. Howard Hinnant: *Date*. `https://howardhinnant.github.io/date/date.html`. Github. 2018.
26. Howard Hinnant: *Timezones*. `https://howardhinnant.github.io/date/tz.html`. Github. 2018.
27. Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.
28. Anastasia Kazakova: *Infographic: C/C++ facts*. `https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/` July 2015.
29. Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall. Englewood Cliffs, New Jersey. 1978.
30. Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

    Русский перевод: Брайан Керниган, Деннис Ритчи. *Язык программирования C*. — М.: Издательский дом "Вильямс", 2015. — 304 с.
31. Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

Русский перевод: Дональд Э. Кнут. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.: Издательский дом "Вильямс", 2000. — 720 с.

32. A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised).* Proc USENIX C++ Conference. April 1990.
33. John Maddock: *Boost.Regex.* `www.boost.org`. 2009. 2017.
34. ISO/IEC JTC1/SC22/WG21 (editor: Gabriel Dos Reis): *Technical Specification: C++ Extensions for Modules.* ISO/IEC TS 21544:2018.
35. George Orwell: *1984.* Secker and Warburg. London. 1949.
36. Larry C. Paulson: *ML for the Working Programmer.* Cambridge University Press. Cambridge. 1996.
37. ISO/IEC JTC1/SC22/WG21 (editor: Eric Niebler): *Technical Specification: C++ Extensions for Ranges.* ISO/IEC TS 21425:2017. ISBN 0-521-56543-X.
38. Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler.* Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.
39. Alexander Stepanov and Meng Lee: *The Standard Template Library.* HP Labs Technical Report HPL-94-34 (R. 1). 1994.
40. Alexander Stepanov and Paul McJones: *Elements of Programming.* Addison-Wesley. 2009. ISBN 978-0-321-63537-2.
41. Б. Страуструп. *Личные лабораторные заметки.*
42. B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language.* Sigplan Notices. January 1982. The first public description of "C with Classes."
43. B. Stroustrup: *Operator Overloading in C++.* Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.
44. B. Stroustrup: *An Extensible I/O Facility for C++.* Proc. Summer 1985 USENIX Conference.
45. B. Stroustrup: *The C++ Programming Language.* Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.
46. B. Stroustrup: *Multiple Inheritance for C++.* Proc. EUUG Spring Conference. May 1987.
47. B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming.* Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.
48. B. Stroustrup: *Parameterized Types for C++.* Proc. USENIX C++ Conference, Denver, Colorado. 1988.

49. B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.

50. B. Stroustrup: *A History of C++: 1979–1991*. Proc. ACM History of Programming Languages Conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

51. B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-54330-3.

52. B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover ("Special") Edition. 2000. ISBN 0-201-70073-5.

    Русский перевод: Бьерн Страуструп. Язык программирования C++. Третье издание. — СПб.; М.: "Невский Диалект" — "Издательство БИНОМ", 1999. — 991 с.

53. B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. `www.stroustrup.com/papers.html`.

54. B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

55. B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

    Русский перевод: Бьярне Страуструп. *Программирование. Принципы и практика с использованием C++*. — М.: ООО "И.Д. Вильямс", 2016. — 1328 с.

56. B. Stroustrup: *The C++11 FAQ*. `www.stroustrup.com/C++11FAQ.html`.

57. B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.

58. B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

59. B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley. 2013. ISBN 0-321-56384-0.

60. B. Stroustrup: C++ Applications. `http://www.stroustrup.com/applications.html`.

61. B. Stroustrup and H. Sutter: *C++ Core Guidelines*. `https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md`.

62. B. Stroustrup, H. Sutter, and G. Dos Reis: *A brief introduction to C++'s model for type- and resource-safety.* Isocpp.org. October 2015. Revised December 2015. `http://www.stroustrup.com/resource-model.pdf`.
63. A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++.* Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.
64. ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive.* `www.open-std.org/jtc1/sc22/wg21`.
65. Anthony Williams: *C++ Concurrency in Action – Practical Multithreading.* Manning Publications Co. ISBN 978-1933988771.
66. P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. 1974.

16.5. Советы

[1] Язык программирования C++ определен в стандарте ISO C++ [10].

[2] При выборе стиля для нового проекта или модернизации старого кода опирайтесь на C++ Core Guidelines; §16.1.4.

[3] При изучении C++ не сосредоточивайтесь на изолированных языковых возможностях; §16.2.1.

[4] Не зацикливайтесь на старых возможностях языка и методах проектирования; §16.1.4.

[5] Прежде чем использовать новую функциональную возможность в производственном коде, испытайте ее, написав небольшие программы для тестирования производительности и соответствия стандартам реализаций, которые вы планируете использовать.

[6] При изучении C++ используйте самую современную и полную реализацию стандарта C++, к которой вы можете получить доступ.

[7] Общее подмножество C и C++ не является наилучшим начальным подмножеством C++ для изучения; §16.3.2.1.

[8] Предпочитайте именованные приведения, такие как `static_cast`, приведениям в стиле C; §16.2.7.

[9] При преобразовании программы на языке C в программу на языке C++ сначала убедитесь в последовательном, согласованном использовании объявлений функций (прототипов) и стандартных заголовочных файлов; §16.3.2.

[10] При преобразовании программы на языке C в программу на языке C++ выполните переименование переменных, имена которых являются ключевыми словами C++; §16.3.2.

[11] Для переносимости и безопасности типов, если вы вынуждены использовать C, пишите программы на общем подмножестве C и C++; §16.3.2.1.

[12] При преобразовании программы на языке C в программу на языке C++ приводите результат `malloc()` к нужному типу или, что еще лучше, замените все использования `malloc()` операторами `new`; §16.3.2.2.

[13] При преобразовании применения `malloc()` и `free()` в использование операторов `new` и `delete` рассмотрите возможность использования `vector`, `push_back()` и `reserve()` вместо `realloc()`; §16.3.2.1.

[14] В C++ нет неявных преобразований значений `int` в перечисления; при необходимости используйте явное преобразование типов.

[15] Для каждого стандартного заголовочного файла C `<X.h>`, который помещает имена в глобальное пространство имен, имеется соответствующий заголовочный файл `<cX>`, который помещает имена в пространство имен `std`.

[16] Используйте `extern "C"` при объявлении функций C; §16.3.2.3.

[17] Предпочитайте строки `string` строкам в стиле C (непосредственной работе с массивами типа `char` с завершающим нулевым символом).

[18] Предпочитайте использование `iostream` работе с `stdio`.

[19] Предпочитайте контейнеры (например, `vector`) встроенным массивам.

Предметный указатель

Есть два вида знаний.
Мы можем знать сам предмет или знать,
где найти информацию о нем.

– Сэмюэль Джонсон (Samuel Johnson)

Л

М

Н

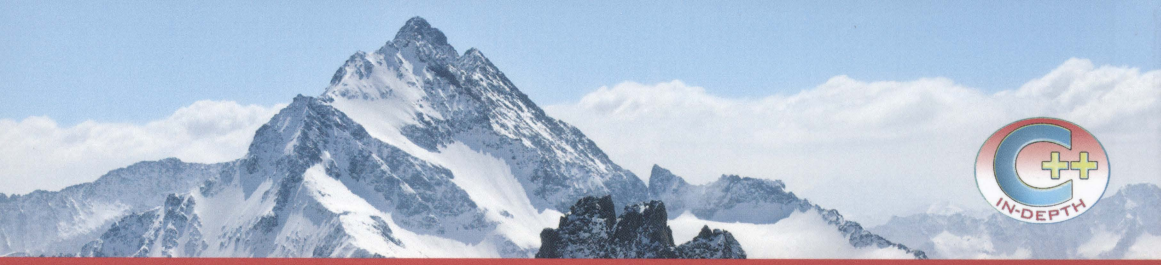
О

П

Р

С

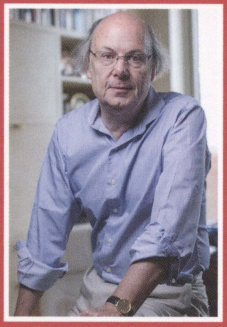
Т



В этой книге создатель языка C++ Бьярне Страуструп описывает, что собой представляет современный C++. Это краткое самодостаточное руководство охватывает основные функциональные возможности языка и основные компоненты стандартной библиотеки — пусть и не с полной глубиной изложения материала, однако на высоком профессиональном уровне. Книга включает множество конкретных примеров, которые облегчают изучение данного языка программирования.

Страуструп представляет функциональные возможности C++ в контексте поддерживаемых ими стилей программирования, таких как объектно-ориентированное и обобщенное программирование. Его книга на удивление всеобъемлюща — она начинается с основ языка программирования C++ и постепенно переходит к таким сложным темам, как многие новые и уже устоявшиеся функциональные возможности C++17, включая семантику перемещения, однородную инициализацию, лямбда-выражения, усовершенствованные контейнеры, случайные числа и параллелизм. Сюда входят и некоторые расширения C++20, например концепты и модули. Заканчивается книга обсуждением дизайна и эволюции C++.

Это руководство не ставит целью научить читателя программировать (для этого служит другая книга того же автора — *Программирование. Принципы и практика с использованием C++. Второе издание*). Не является она и исчерпывающим учебником, который приведет вас на вершины мастерства C++ (здесь можно порекомендовать книгу Страуструпа *Язык программирования C++. Четвертое издание* и множество источников информации в Интернете). Однако если вы являетесь программистом на C или C++, желающим получше ознакомиться с текущим состоянием языка программирования C++, или программистом на другом языке программирования, желающим увидеть точную картину и преимущества современного C++, то более короткого и простого введения в C++, чем эта книга, вам не найти.



Доктор Бьярне Страуструп — изобретатель и первый разработчик языка программирования C++, перу которого принадлежат книги *Программирование. Принципы и практика с использованием C++. Второе издание, Язык программирования C++. Четвертое издание* и многие другие.

Ранее Страуструп работал в Bell Labs, AT&T Labs и Texa[illegible]versity, а в настоящее время является управляющим директором [illegible]ром технологий Morgan Stanley в Нью-Йорке и приглашенны[illegible]ром в Колумбийском университете. Он удостоен многочисле[illegible] в том числе премии 2018 года Национальной инженерной акад[illegible]ме концептуализацию и разработку языка программировани[illegible]. Кроме того, доктор Страуструп является членом Национальной [illegible] академии, а также сотрудником IEEE и ACM.

Фотография на обложке – pchais/Shutterstock
Фотография автора предоставлена Национальной инженерной академией США

ISBN 978-5-907144-12-5

Диалектика
www.williamspublishing.com

Pearson
Addison-Wesley