Глава 1
Основы C++

Моим детям. Никогда не смейтесь, помогая мне осваивать компьютер. В конце концов, я учила вас пользоваться горшком.

— Сью Фитцморис

В этой главе мы рассмотрим основные возможности C++. Как и во всей книге, мы будем рассматривать их с разных точек зрения, но постараемся не вдаваться во все детали, ведь это все равно невозможно. Для более подробной информации о конкретных возможностях языка мы рекомендуем вам обратиться к онлайн-руководствам по адресам `http://www.cplusplus.com/` и `http://ru.cppreference.com`.

1.1. Наша первая программа

В качестве введения в язык программирования C++ рассмотрим следующий пример:

```
#include <iostream>
int main()
{
    std::cout << "Ответом на Великий Вопрос о Жизни,\n"
              << "Вселенной и Всяком Таком является:"
              << std::endl << 6 * 7 << std::endl;
    return 0;
}
```

Вывод этой программы, в полном соответствии с Дугласом Адамсом (Douglas Adams) [2], имеет вид:

```
Ответом на Великий Вопрос о Жизни,
Вселенной и Всяком Таком является:
42
```

Этот короткий пример иллюстрирует несколько возможностей C++.

- Ввод и вывод не являются частью ядра языка и предоставляются библиотекой. Она должна быть включена явно; в противном случае мы ничего не сможем вводить и выводить.

- Стандартный ввод-вывод имеет потоковую модель и потому именуется <iostream>. Для обеспечения ее функциональности мы в первой строке программы указываем директиву ее включения #include <iostream>.
- Каждая программа C++ начинается с вызова функции main. Она возвращает с помощью оператора return целочисленное значение; возврат нуля означает успешное завершение.
- Фигурные скобки {} означают блок/группу кода (именуемую также составной инструкцией).
- std::cout и std::endl определены в <iostream>. Первое из этих выражений представляет собой выходной поток, который выводит текст на экран. std::endl завершает строку. Мы можем также переходить на новую строку с помощью специального символа \n.
- Оператор << можно использовать для передачи объекта в выходной поток, такой как std::cout, для выполнения операции вывода.
- std:: указывает, что использованный тип (или функция) взят из стандартного *пространства имен*. Пространства имен помогают организовывать имена и избегать конфликтов имен (см. раздел 3.2.1).
- Строковые константы (более точно — литералы) заключены в двойные кавычки.
- Выражение 6*7 вычисляется и передается в std::cout как целочисленное значение. В C++ каждое выражение имеет тип. Иногда мы как программисты обязаны явно объявлять тип, а в других случаях компилятор сам выводит его для нас. 6 и 7 являются константными литералами типа int, так что их произведение также представляет собой int.

Прежде чем продолжить чтение, мы настоятельно рекомендуем вам скомпилировать и запустить этот маленький пример на своем компьютере. После того как он будет скомпилирован и запущен, вы сможете немного с ним поиграться, например, добавляя больше арифметических операций и операций вывода (и рассмотреть получаемые сообщения об ошибках). В конце концов, единственным способом действительно выучить язык является его использование. Если вы уже знаете, как использовать компилятор или даже интегрированную среду разработки C++, можете пропустить оставшуюся часть этого раздела.

Linux. В каждом дистрибутиве имеется как минимум компилятор GNU C++, обычно устанавливаемый по умолчанию (см. краткое введение в разделе Б.1). Пусть мы назвали нашу программу hello42.cpp; тогда ее легко скомпилировать с помощью команды

```
g++ hello42.cpp
```

По традиции прошедшего века по умолчанию результирующий бинарный файл называется a.out. Но поскольку программ у нас может быть больше, чем

одна, давайте используем более значимое имя, указав соответствующий флаг в командной строке:

```
g++ hello42.cpp -o hello42
```

Можно также воспользоваться инструментом `make` (рассматривается в разделе 7.2.2.1), который (в своих последних версиях) предоставляет правила построения бинарных файлов по умолчанию. Так что мы можем просто вызвать его командой

```
make hello42
```

После этого инструмент `make` выполнит поиск в текущем каталоге исходного файла программы с данным именем. Он найдет `hello42.cpp`, а так как `.cpp` является стандартным расширением файлов с исходными текстами C++, будет вызван системный компилятор C++ по умолчанию. После компиляции программы мы можем вызвать ее из командной строки как

```
./hello42
```

Наш бинарный файл может выполняться без привлечения какого-либо иного программного обеспечения и может быть скопирован в любую совместимую Linux-систему[1] и выполнен в ней.

Windows. Если вы работаете с MinGW, можете компилировать программу точно так же, как и в Linux. Если вы используете Visual Studio, то сначала создайте проект[2]. Для начинающего простейший путь заключается в использовании шаблона проекта для консольного приложения, как описано, например, по адресу `http://www.cplusplus.com/doc/tutorial/introduction/visualstudio`. При запуске такой программы у вас будет буквально несколько миллисекунд, чтобы успеть прочесть ее вывод до того, как консольное окно закроется. Чтобы увеличить это время до секунды, можно просто добавить непереносимую команду `Sleep(1000);` и включить заголовочный файл `<windows.h>`. При использовании C++11 или более поздней версии ожидание можно реализовать переносимо после включения заголовочных файлов `<chrono>` и `<thread>`:

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

Microsoft предлагает бесплатные версии Visual Studio, называемые “Express”, которые обеспечивают поддержку стандарта языка программирования, как и их профессиональные аналоги. Разница заключается в наличии у профессиональных выпусков большего количества библиотек и возможностей. Поскольку в этой книге они не используются, вы можете смело использовать для компиляции примеров версию “Express”.

---

[1] Зачастую стандартная библиотека компонуется динамически (см. раздел 7.2.1.4), и тогда частью требований совместимости становится наличие той же ее версии на другой системе.

[2] Вообще говоря, это не обязательно, так как Visual Studio, помимо интегрированной среды разработки, предоставляет инструментарий командной строки. — *Примеч. ред.*

Интегрированная среда разработки. Короткие программы наподобие примеров из этой книги легко набрать в обычном редакторе. Бо́льшие проекты лучше создавать с использованием *интегрированных сред разработки* (Integrated Development Environment — IDE), которые позволяют увидеть, где определяется или используется функция, просмотреть документацию, найти или заменить имена во всем проекте и т.д. Такой бесплатной интегрированной средой разработки является KDevelop от KDE, написанная на C++. Это, вероятно, наиболее эффективная интегрированная среда разработки в Linux, хорошо интегрированная с `git` и `CMake`. Eclipse разработана на Java и существенно более медленная. Однако в последнее время было вложено много усилий в ее поддержку C++, так что многим разработчикам она нравится и они достаточно продуктивно с ней работают. Visual Studio — это очень солидная интегрированная среда разработки с некоторыми уникальными возможностями.

Чтобы найти наиболее продуктивную и подходящую для себя интегрированную среду разработки, нужно потратить некоторое время на проведение экспериментов; конечно же, огромную роль будут играть ваши личные предпочтения и вкусы.

1.2. Переменные

C++ является строго типизированным языком программирования (в отличие от множества языков сценариев). Это означает, что каждая переменная имеет тип, и этот тип никогда не изменяется. Переменная объявляется с помощью инструкции, начинающейся с типа, за которым следует имя переменной с необязательной инициализацией (или список таких переменных):

```
int    i1 = 2;        // Выравнивание нужно только для удобочитаемости
int    i2, i3= 5;
float  pi = 3.14159;
double x = -1.5 e6;   // -1500000
double y = -1.5e-6;   // -0.0000015
char   c1 = 'a', c2= 35;
bool   cmp = i1 < pi, // -> true
       happy = true;
```

Две косые черты // указывают начало однострочного комментария, т.е. все начиная от этих двух символов и до конца строки компилятором игнорируется. В принципе, это все, что надо знать о комментариях. Чтобы у вас не было ощущения, что мы пропустили что-то важное по этой теме, рассмотрим их немного подробнее в разделе 1.9.1.

Вернемся к переменным. Их основные типы — именуемые также *встроенными типами* — перечислены в табл. 1.1.

Таблица 1.1. Встроенные типы

Имя	Семантика
`char`	Буквы и очень небольшие целочисленные значения
`short`	Короткое целое число
`int`	Обычное целое число
`long`	Длинное целое число
`long long`	Очень длинное целое число
`unsigned`	Беззнаковая версия перечисленных значений
`signed`	Знаковая версия перечисленных значений
`float`	Число с плавающей точкой одинарной точности
`double`	Число с плавающей точкой двойной точности
`long double`	Длинное число с плавающей точкой
`bool`	Логическое значение

Первые пять типов представляют собой целые числа неуменьшающейся длины. Например, `int` имеет длину, не меньшую, чем длина `short`, т.е. обычно (но не обязательно) оно длиннее. Точная длина каждого типа зависит от реализации; например, тип `int` может быть длиной 16, 32 или 64 бита. Все эти типы могут быть знаковыми (`signed`) или беззнаковыми (`unsigned`). Первый квалификатор не оказывает никакого влияния на целые числа (за исключением `char`), поскольку они являются знаковыми по умолчанию.

Объявляя целочисленный тип как `unsigned`, мы не разрешаем ему принимать отрицательные значения, зато вдвое увеличиваем диапазон допустимых положительных значений (плюс одно, если нуль рассматривается ни как отрицательное, ни как положительное число). Слова `signed` и `unsigned` можно рассматривать как прилагательные для существительных `short`, `int` и других, где `int` — существительное, применяемое по умолчанию, когда имеются только прилагательные.

Тип `char` может быть использован двумя способами — для букв и для очень коротких чисел. За исключением действительно экзотических архитектур, он почти всегда имеет длину 8 битов. Таким образом, он может представлять значения либо от `-128` до `127` (`signed`), либо от `0` до `255` (`unsigned`), и над ним можно выполнять все числовые операции, доступные для целых чисел. Если не указан ни квалификатор `signed`, ни `unsigned`, то, какой из них используется по умолчанию, зависит от реализации. Мы можем также представить любую букву, код которой помещается в 8 битов. Их можно даже смешивать, например `'a'+7` обычно дает `'h'` (в зависимости от используемой кодировки букв). Мы настойчиво рекомендуем не прибегать к таким способам, поскольку возможная путаница, скорее всего, приведет к напрасной трате времени.

Применение `char` или `unsigned char` для небольших чисел может быть полезным при наличии больших контейнеров с ними.

Логические значения лучше всего представимы с помощью типа `bool`. Такая переменная может хранить значение `true` или `false`. Свойство неуменьшающейся длины применимо и к числам с плавающей точкой:

Тип `float` короче или той же длины, что и тип `double`, который, в свою очередь, короче или той же длины, что и тип `long double`. Типичными размерами являются 32 бита для `float`, 64 бита — для `double` и 80 битов — для `long double`.

В следующем разделе мы познакомимся с операциями, применимыми к целочисленным типам и типам с плавающей точкой. В отличие от других языков программирования, таких как Python, в которых кавычки `'` и `"` используются и для символов, и для строк, C++ их различает. Компилятор C++ рассматривает `'a'` как символ "a" (с типом `char`), а `"a"` — как строку, содержащую символ "a" и бинарный нуль в качестве завершающего символа (т.е. типом этой строки является `char[2]`). Если вы работали ранее с Python, обратите на это особое внимание.

Совет

Объявляйте переменные как можно позже, обычно непосредственно перед их первым применением, и, насколько это возможно, не ранее чем вы сможете их инициализировать.

Этот совет делает большие программы более удобочитаемыми. Кроме того, это позволяет компилятору более эффективно использовать память при наличии вложенных областей видимости.

C++11 в состоянии вывести тип переменной, например:

```
auto i4= i3 + 7;
```

Тип `i4` тот же, что и у `i3+7`, т.е. `int`. Хотя тип определен автоматически, он остается неизменным, так что все, что впоследствии будет присвоено переменной `i4`, будет преобразовано в `int`. Позже мы увидим, насколько в современном программировании полезно ключевое слово `auto`. В простых объявлениях переменных, наподобие приводимых в этом разделе, обычно лучше объявлять тип явно. Применение `auto` подробно рассматривается в разделе 3.4.

1.2.1. Константы

Синтаксически константы в C++ представляют собой специальные переменные с дополнительным атрибутом постоянства в C++.

```
const int   ci1  = 2;
const int   ci3;        // Ошибка: не указано значение
const float pi   = 3.14159;
const char  cc   = 'a';
const bool  cmp  = ci1 < pi;
```

Поскольку они не могут быть изменены, обязательным является установка их значений в объявлении. Второе объявление константы нарушает данное правило, и компилятор не простит вам такой проступок.

Константы могут использоваться везде, где разрешено применение переменных, — конечно, до тех пор, пока они не изменяются. С другой стороны, кон-

станты, как показанные выше, известны уже во время компиляции. Это позволяет компилятору применять множество видов оптимизации, так что константы могут даже использоваться в качестве аргументов типов (мы вернемся к этому позже, в разделе 5.1.4).

1.2.2. Литералы

Литералы, такие как `2` или `3.14`, точно типизированы. Проще говоря, целочисленные значения рассматриваются как имеющие тип `int`, `long` или `unsigned long` — в зависимости от количества цифр. Любое число с точкой или показателем степени (например, `3e12` ≡ $3 \cdot 10^{12}$) считается значением типа `double`.

Литералы других типов могут быть записаны путем добавления суффикса из следующей таблицы.

Литерал	Тип
`2`	`int`
`2u`	`unsigned`
`2l`	`long`
`2ul`	`unsigned long`
`2.0`	`double`
`2.0f`	`float`
`2.0l`	`long double`

В большинстве случаев нет необходимости явно объявлять тип литералов, так как неявное преобразование между встроенными числовыми типами обычно дает значения, ожидаемые программистом.

Однако есть три основные причины, по которым необходимо уделять внимание типам литералов.

Доступность. Стандартная библиотека предоставляет тип для комплексных чисел, в котором типы действительной и мнимой частей могут быть параметризованы пользователем:

```
std::complex<float> z(1.3,2.4), z2;
```

К сожалению, предоставляются только операции между самим типом комплексного числа и базовым типом (аргументы в данном случае не преобразуются)[3]. В результате мы не можем умножить `z` на `int` или `double`, а только на `float`:

```
z2 = 2    * z; // Ошибка: нет int * complex<float>
z2 = 2.0  * z; // Ошибка: нет double * complex<float>
z2 = 2.0f * z; // Все в порядке, float * complex<float>
```

Неоднозначность. При перегрузке функции для других типов аргументов (раздел 1.5.4) аргумент наподобие `0` может оказаться неоднозначным, в то время как для квалифицированного аргумента наподобие `0u` может иметься единственное соответствие.

[3] Но такая смешанная арифметика может быть реализована, как показано в [18].

Точность. Вопрос точности встает, когда мы работаем с типом `long double`. Поскольку неквалифицированный литерал имеет тип `double`, возможна потеря значащих цифр до присваивания переменной типа `long double`:

```
long double
    third1 = 0.3333333333333333333,  // Возможна потеря точности
    third2 = 0.33333333333333333331; // Точно
```

Если вам кажется, что в предыдущих трех абзацах мы были слишком краткими, примите к сведению, что более подробное изложение представлено в разделе А.2.1.

Не десятичные числа. Целочисленные литералы, начинающиеся с нуля, трактуются как восьмеричные числа, например:

```
int o1= 042;  // int o1= 34;
int o2= 084;  // Ошибка: ни 8, ни 9 не являются восьмеричными цифрами!
```

Шестнадцатеричные литералы записываются с использованием префикса `0x` или `0X`:

```
int h1= 0x42; // int h1= 66;
int h2= 0xfa; // int h2= 250;
```

В C++14 появились бинарные литералы, которые записываются с использованием префикса `0b` или `0B`:

```
int b1= 0b11111010; // int b1= 250;
```

Для повышения удобочитаемости длинных литералов C++14 допускает разделение цифр апострофами:

```
long               d = 6'546'687'616'861'129l;
unsigned long      ulx = 0x139'ae3b'2ab0'94f3;
int                b = 0b101'1001'0011'1010'1101'1010'0001;
const long double pi = 3.141'592'653'589'793'238'4621;
```

Строковые литералы имеют тип массива символов `char`:

```
char s1[]= "Старый стиль C"; // Лучше так не делать
```

Однако куда удобнее работать со строками типа `string` из библиотеки `<string>`. Такая строка может быть легко создана из строкового литерала:

```
#include <string>
std::string s2= "В C++ лучше делать так";
```

Очень длинный текст можно разбить на несколько подстрок:

```
std::string s3= "Это очень длинный текст, который"
                " не помещается в одну строку";
```

Более подробную информацию о литералах можно найти, например, в [43, §6.2].

1.2.3. Не сужающая инициализация в C++11

Представим, что мы инициализируем переменную типа `long` длинным числом:

```
long l2= 1234567890123;
```

Этот код корректно компилируется и работает, если переменная типа `long` занимает 64 бита, как это происходит на большинстве 64-битовых платформ. Если тип `long` имеет длину всего 32 бита (этого можно добиться путем компиляции с использованием флага наподобие `-m32`), то показанное выше значение оказывается слишком большим. Однако программа продолжает компилироваться (может быть, и с предупреждениями) и выполняться, но с другим значением, в котором отсечены ведущие биты.

В C++11 введена инициализация, обеспечивающая отсутствие потери данных, или, иными словами, не допускающая *сужения* значений. Это достигается с помощью *унифицированной инициализации* (uniform initialization) или *инициализации с фигурными скобками* (braced initialization), которую мы только бегло затрагиваем здесь, а более подробно будем изучать в разделе 2.3.4. Значения в фигурных скобках не могут быть сужены:

```
long l= { 1234567890123 };
```

Теперь компилятор будет проверять, может ли переменная `l` хранить указанное значение в целевой архитектуре.

Защита компилятора от сужения позволяет нам убедиться, что значения не потеряют точности при инициализации. Обычная инициализация `int` числом с плавающей точкой разрешается в силу выполнения неявного преобразования:

```
int i1  =  3.14;   // Компилируется, несмотря на сужение (на ваш риск)
int i1n = {3.14};  // Ошибка сужения: теряется дробная часть
```

Новая разновидность инициализации во второй строке запрещает его, поскольку при этом будет отсечена дробная часть числа с плавающей точкой. Аналогично присваивание отрицательных значений беззнаковым переменным или константам пропускается традиционной инициализацией, но отклоняется новой разновидностью:

```
unsigned u2 = -3;   // Компилируется, несмотря на сужение (на ваш риск)
unsigned u2n={-3};  // Ошибка сужения: отрицательное значение
```

В предыдущих примерах мы использовали в инициализации литералы, и компилятор проверял, представимо ли конкретное значение указанным типом:

```
float f1= { 3.14 }; // OK
```

Значение 3.14 не может быть представлено с абсолютной точностью в двоичном формате с плавающей точкой, но компилятор может установить значение `f1` близким к 3.14. Когда `float` инициализируется переменной или константой `double` (не литералом), необходимо рассмотреть все возможные значения `double` и то, являются ли они преобразуемыми в тип `float` без потерь.

```
double d;
...
float f2= {d}; // Ошибка сужения
```

Обратите внимание, что сужение между двумя типами может быть взаимным:

```
unsigned u3= {3};
int      i2= {2};
unsigned u4= {i2}; // Ошибка сужения: возможно отрицательное значение
int      i3= {u3}; // Ошибка сужения: возможно слишком большое значение
```

Типы `signed int` и `unsigned int` имеют одинаковый размер, но не все значения каждого типа представимы другим типом.

1.2.4. Области видимости

Области видимости определяют время жизни и видимость (не статических) переменных и констант и способствуют установлению структуры в наших программах.

1.2.4.1. Глобальные переменные

Каждая переменная, которую мы намерены использовать в программе, должна быть объявлена с указанием ее типа в некоторой более ранней точке кода. Переменная может находиться в глобальной или локальной области видимости. Глобальная переменная объявляется вне всех функций. После объявления обращаться к глобальным переменным можно в любом месте кода, даже внутри функций. Это выглядит очень удобным, в первую очередь, потому, что делает переменные легко доступными, но с ростом размера программы отслеживать изменения глобальных переменных становится более трудно и болезненно. В некоторый момент каждое изменение кода потенциально способно вызвать лавину ошибок.

Совет

Не используйте глобальные переменные.

Если вы используете их, рано или поздно вы об этом пожалеете. Поверьте нам. Глобальные константы вроде

```
const double pi= 3.14159265358979323846264338327950288419716939;
```

можно применять, поскольку они не вызывают побочных действий.

1.2.4.2. Локальные переменные

Локальная переменная объявляется внутри тела функции. Ее видимость/доступность ограничивается заключенным в фигурные скобки `{}` блоком, в котором она объявлена. Точнее, область видимости переменной начинается с ее объявления и заканчивается закрывающей фигурной скобкой блока, в котором она объявлена.

Если мы определяем pi в функции main:

```
int main ()
{
    const double pi= 3.14159265358979323846264338327950288419716939;
    std::cout << "pi = " << pi << ".\n";
}
```

то переменная pi существует только в функции main. Мы можем определять блоки внутри функций и других блоков:

```
int main ()
{
  {
    const double pi= 3.14159265358979323846264338327950288419716939;
  }
  // Ошибка: pi вне области видимости:
  std::cout << "pi = " << pi << ".\n";
}
```

В этом примере определение pi ограничено блоком внутри функции, так что попытка вывода этого значения в оставшейся части функции приводит к ошибке времени компиляции, поскольку pi находится *вне области видимости*.

1.2.4.3. Сокрытие

Когда во вложенных областях видимости имеется переменная с тем же именем, видна только одна переменная. Переменная во внутренней области видимости скрывает одноименные переменные во внешних областях. Например:

```
int main ()
{
    int a= 5;       // Определение a №1
    {
        a = 3;      // Присваивание a №1, a №2 еще не определена
        int a;      // Определение a №2
        a = 8;      // Присваивание a №2, a №1 скрыта
        {
            a = 7; // Присваивание a №2
        }
    }               // Конец области видимости a №2
    a = 11;         // Присваивание a №1 (a №2 вне области видимости)
    return 0;
}
```

Из-за сокрытия мы должны различать время жизни и видимость переменных. Например, продолжительность жизни переменной a №1 — от ее объявления до конца функции main. Однако она видима только от ее объявления до объявления a №2 и вновь после закрытия блока, содержащего переменную a №2. Фактически видимость представляет собой время жизни минус время, когда переменная скрыта.

Определение одного и того же имени переменной дважды в одной области видимости является ошибкой.

Преимуществом областей видимости является то, что нам не нужно беспокоиться о том, не определено ли имя где-то за пределами области видимости. Оно будет просто скрыто, и не создаст конфликт имен[4]. К сожалению, сокрытие делает недоступными одноименные переменные из внешней области видимости. В некоторой степени справиться с этим можно с помощью разумного переименования. Лучшим решением для управления вложенностью и доступностью являются пространства имен (см. раздел 3.2.1).

Статические (`static`) переменные являются исключением, которое подтверждает правило. Они живут до конца выполнения программы, но видны только в своей области видимости. Опасаясь, что их подробное описание на этом этапе знакомства с языком программирования будет более отвлекающим, чем полезным, мы отложили их обсуждение до раздела A.2.2.

1.3. Операторы

C++ имеет множество встроенных операторов. Имеются различные виды операторов.

- Вычислительные
  - Арифметические. `++`, `+`, `*`, `%`, …
  - Булевы
    - Сравнения. `<=`, `!=`, …
    - Логические. `&&` и `||`
  - Побитовые. `~`, `<<` и `>>`, `&`, `^` и `|`
- Присваивания. `=`, `+=`, …
- Потока управления. Вызов функции, `?:` и `,`
- Работы с памятью. `new` и `delete`
- Доступа. `.`, `->`, `[ ]`, `*`, …
- Работы с типами. `dynamic_cast`, `typeid`, `sizeof`, `alignof`, …
- Обработки ошибок. `throw`

В этом разделе приведен обзор операторов. Некоторые операторы лучше описывать в контексте соответствующих возможностей языка; например, разрешение области видимости лучше всего объяснять вместе с пространствами имен.

[4] В противоположность макросам, устаревшей и безответственной возможности, унаследованной от C, которой следует избегать любой ценой, поскольку она подрывает всю структуру и надежность языка.

Большинство операторов могут быть перегружены для пользовательских типов, т.е. мы можем решать, какие вычисления выполняются, когда один или несколько аргументов в выражении имеют созданные нами типы.

В конце этого раздела вы найдете краткую таблицу (табл. 1.8) приоритетов операторов. Может оказаться полезным распечатать ее и хранить рядом с монитором — так поступают многие программисты, и почти никто не знает весь список приоритетов наизусть. Если вы не уверены в приоритетах или если вы считаете, что так код будет более понятен программистам, работающим с вашими исходными текстами, без колебаний используйте скобки вокруг подвыражений. В разделе B.2 имеется полный список всех операторов с краткими описаниями и ссылками.

1.3.1. Арифметические операторы

В табл. 1.2 перечислены все арифметические операторы, доступные в C++. Мы рассортировали их согласно приоритетам, но давайте рассмотрим их по одному.

Таблица 1.2. Арифметические операторы

Операция	Выражение
Пост-инкремент	`x++`
Пост-декремент	`x--`
Пре-инкремент	`++x`
Пре-декремент	`--x`
Унарный плюс	`+x`
Унарный минус	`-x`
Умножение	`x * y`
Деление	`x / y`
Остаток от деления (деление по модулю)	`x % y`
Сложение	`x + y`
Вычитание	`x - y`

Первой разновидностью операций являются инкремент и декремент. Эти операции могут использоваться для увеличения или уменьшения числа на 1. Так как они изменяют значение числа, они имеют смысл только для переменных, но не для временных результатов, например:

```
int i = 3;
i++;        // Теперь i равно 4
const int j= 5;
j++;        // Ошибка: j является константой
(3+5)++;    // Ошибка: 3 + 5 является временным результатом
```

Короче говоря, операциям инкремента и декремента нужно что-то, что изменяемо и адресуемо. Техническим термином для адресуемого элемента данных является *lvalue* (см. определение B.1 в приложении B). В приведенном выше

фрагменте кода это верно только для переменной i. В противоположность ему j является константой, а значение 3+5 не адресуемо.

Обе записи — префиксная и постфиксная — одинаково добавляют 1 к значению переменной или вычитают 1 из него. Однако смысл выражения инкремента и декремента различается для префиксных и постфиксных операторов. Префиксные операторы возвращают измененное значение, а постфиксные — старое значение, например:

```
int i = 3, j= 3;
int k = ++i + 4;  // i = 4, k = 8
int l = j++ + 4;  // j = 4, l = 7
```

В конечном итоге и i, и j равны 4. Однако при вычислении l используется старое значение j, в то время как в первом сложении используется уже увеличенное значение i.

В общем случае лучше воздерживаться от использования инкремента и декремента в математических выражениях и заменять их выражениями j+1 и тому подобными или выполнять инкремент и декремент отдельно. Так исходный текст легче читается и понимается человеком, а компилятору легче оптимизировать код, когда математические выражения не имеют побочных эффектов. Вскоре мы увидим, с чем это связано (раздел 1.3.12).

Унарный минус изменяет знак числового значения:

```
int i= 3;
int j= -i; // j = -3
```

Унарный плюс не выполняет никакого арифметического действия над стандартными типами. Для пользовательских типов мы можем определить свое поведение для унарного плюса и минуса. Как показано в табл.1.2, эти унарные операторы имеют приоритет, совпадающий с приоритетом префиксных инкремента и декремента.

Операции * и / являются естественными умножением и делением, и обе они определяются для всех числовых типов. Когда оба аргумента деления являются целыми числами, дробная часть результата отбрасывается (округление по направлению к нулю). Оператор % возвращает остаток от целочисленного деления. Таким образом, оба аргумента этого оператора должны иметь целочисленный тип.

Последними по очереди, но не по важности идут операторы + и –, которые обозначают сложение и вычитание двух переменных или выражений.

Семантические сведения об операциях — как округляются результаты или как обрабатывается переполнение — в языке не определены. По соображениям производительности C++ оставляет окончательное решение за используемым аппаратным обеспечением.

В общем случае унарные операторы имеют более высокий приоритет, чем бинарные. В тех редких случаях, когда применяются и префиксный, и постфиксный унарные операторы, префиксный оператор имеет более высокий приоритет, чем постфиксный.

Бинарные операторы ведут себя так же, как и в математике. Умножение и деление имеют больший приоритет, чем сложение и вычитание, а сами операции являются левоассоциативными, т.е.

```
x - y + z
```

всегда трактуется как

```
(x - y) + z
```

Есть кое-что, что действительно важно запомнить: порядок вычисления аргументов не определен. Взгляните на этот код:

```
int i= 3, j= 7, k;
k= f(++i) + g(++i) + j;
```

В этом примере ассоциативность гарантирует, что первое сложение выполняется до второго. Но какое выражение вычисляется первым — `f(++i)` или `g(++i)`, — зависит от реализации компилятора. Таким образом, `k` может принимать любое значение — `f(4)+g(5)+7`, `f(5)+g(4)+7` или даже `f(5)+g(5)+7`. Кроме того, нельзя полагать, что результат будет тем же самым на другой платформе. В общем случае изменять значения в выражениях — опасная практика. При определенных условиях это работает, но мы всегда должны обращать особое внимание на такой код и тщательно его тестировать. Словом, лучше потратить время на ввод дополнительных символов и выполнять изменения отдельно. Подробнее об этом мы поговорим в разделе 1.3.12.

⇨ c++03/num_1.cpp

С помощью этих операторов мы можем написать нашу первую (завершенную) числовую программу:

```
#include <iostream>
int main ()
{
    const float r1= 3.5, r2 = 7.3, pi = 3.14159;

    float area1 = pi*r1*r1;
    std::cout << "Круг с радиусом " << r1 << " имеет площадь "
              << area1 << "." << std::endl;

    std::cout << "Среднее " << r1 << " и " << r2 << " равно "
              << (r1 + r2)/2 << "." << std::endl;
}
```

Если аргументы бинарной операции имеют различные типы, один или несколько аргументов автоматически приводятся к общему типу в соответствии с правилами из раздела B.3.

Преобразование может приводить к потере точности. Числа с плавающей точкой предпочтительнее целых чисел, и очевидно, что преобразование 64-разрядного `long` в 32-разрядный `float` приводит к потере точности; даже 32-разрядные значения типа `int` не всегда могут быть представлены правильно в виде 32-разрядных

значений типа `float`, так как некоторые биты нужны для представления показателя степени. Бывают также ситуации, когда целевая переменная может хранить правильный результат, но точность уже потеряна при промежуточных расчетах. Чтобы проиллюстрировать это поведение преобразования, давайте рассмотрим следующий пример:

```
long l= 1234567890123;
long l2= l +  1.0f - 1.0;  // Неточно
long l3= l + (1.0f - 1.0); // Верно
```

На платформе автора это приводит к следующему результату:

```
l2 = 1234567954431
l3 = 1234567890123
```

В случае `l2` мы теряем точность из-за промежуточных преобразований, в то время как `l3` вычисляется правильно. Этот пример, правда, носит искусственный характер, но вы должны быть осведомлены о риске, связанном с неточными промежуточными результатами.

К счастью, вопрос неточности не будет беспокоить нас в следующем разделе.

1.3.2. Булевы операторы

Булевы операторы включают логические операторы и операторы сравнения. Все они, как и предполагается в названии, возвращают значения типа `bool`. Эти операторы и их смысл перечислены в табл. 1.3 (сгруппированы в соответствии с приоритетом).

Таблица 1.3. Булевы операторы

Операция	Выражение
Нет	`!b`
Больше	`x > y`
Больше или равно	`x >= y`
Меньше	`x < y`
Меньше или равно	`x <= y`
Равно	`x == y`
Не равно	`x != y`
Логическое И	`b && c`
Логическое ИЛИ	`b || c`

Бинарные операторы сравнения и логические операторы имеют приоритеты, меньшие, чем приоритеты всех арифметических операторов. Это означает, что выражение наподобие `4>=1+7` вычисляется так, как если бы оно было записано как `4>=(1+7)`. И наоборот, унарный оператор `!` для логического отрицания имеет более высокий приоритет, чем приоритет любого бинарного оператора.

В старом (или старомодном) коде вы можете увидеть логические операции, выполняемые над значениями типа `int`. Воздержитесь от этого в своем коде — это менее удобочитаемо и может вести к неожиданному поведению программы.

Совет

Для булевых выражений всегда используйте тип `bool`.

Пожалуйста, обратите внимание, что сравнения нельзя выстраивать цепочками наподобие следующей:

```
bool in_bound = min <= x <= y <= max; // Ошибка
```

Вместо этого требуется более длинное логическое выражение:

```
bool in_bound = min <= x && x <= y && y <= max;
```

В следующем разделе вы увидите операторы, очень похожие на рассмотренные.

1.3.3. Побитовые операторы

Эти операторы позволяют работать с отдельными битами целочисленных типов. Они очень важны для системного программирования и не так важны при разработке современного программного обеспечения. В табл. 1.4 эти операторы сгруппированы в соответствии с приоритетом.

Таблица 1.4. Побитовые операторы

Операция	Выражение
Дополнение до единицы	`~x`
Левый сдвиг	`x << y`
Правый сдвиг	`x >> y`
Побитовое И	`x & y`
Побитовое исключающее ИЛИ	`x ^ y`
Побитовое включающее ИЛИ	`x | y`

Операция `x << y` сдвигает биты `x` влево на `y` позиций. И наоборот, `x >> y` сдвигает биты `x` на `y` позиций вправо. В большинстве случаев на свободные места добавляются нулевые биты, за исключением отрицательных значений типов `signed` при сдвиге вправо — такой сдвиг зависит от реализации. Побитовое И можно использовать для проверки значения определенного бита значения. Побитовое включающее ИЛИ может установить бит, а исключающее — изменить его значение на противоположное. Эти операции более важны для системного программирования, чем для научного. В качестве алгоритмического развлечения мы используем их в разделе 3.6.1.

1.3.4. Присваивание

Значение объекта (модифицируемого lvalue) может быть установлено с помощью присваивания:

```
object = expr;
```

Если типы не совпадают, значение выражения `expr` преобразуется в тип объекта `object`, если это возможно. Присваивание является правоассоциативной операцией, так что можно последовательно присвоить значение нескольким объектам в одном выражении:

```
o3 = o2 = o1 = expr;
```

Составные операторы присваивания применяют арифметическую или побитовую операцию к объекту слева от оператора с аргументом справа от него. Например, следующие две операции эквивалентны:

```
a += b;   // Эквивалентно выражению в следующей строке
a = a + b;
```

Все операторы присваивания имеют более низкий приоритет, чем все арифметические и побитовые операции, поэтому перед выполнением составного присваивания всегда вычисляется выражение справа от него:

```
a *= b + c; // Эквивалентно выражению в следующей строке
a = a * (b + c);
```

Операторы присваивания перечислены в табл. 1.5. Все они правоассоциативны и имеют один и тот же приоритет.

Таблица 1.5. Операторы присваивания

Операция	Выражение
Простое присваивание	`x = y`
Умножение и присваивание	`x *= y`
Деление и присваивание	`x /= y`
Деление по модулю и присваивание	`x %= y`
Сложение и присваивание	`x += y`
Вычитание и присваивание	`x -= y`
Сдвиг влево и присваивание	`x <<= y`
Сдвиг вправо и присваивание	`x >>= y`
И и присваивание	`x &= y`
Включающее ИЛИ и присваивание	`x |= y`
Исключающее ИЛИ и присваивание	`x ^= y`

1.3.5. Поток выполнения

Имеется три оператора управления потоком выполнения программы. Вызов функции в C++ обрабатывается как оператор. Подробное описание функций и их вызовов приведено в разделе 1.5.

Условный оператор `c ? x : y` вычисляет условие `c`, и, когда оно истинно, выражение имеет значение `x`, и `y` — в противном случае. Он может использоваться как альтернатива ветвлению с помощью конструкции `if`, особенно в тех местах, где разрешается только выражение, но не инструкция (см. раздел 1.4.3.1).

Очень специфичным оператором в C++ является *оператор запятой*, который обеспечивает последовательное вычисление. Его смысл просто в том, что сначала вычисляется подвыражение слева от запятой, а затем — справа от нее. Значением всего выражения является значение правого подвыражения:

```
3+4, 7*9.3
```

Результатом выражения является 65.1, а вычисление первого подвыражения совершенно не играет роли. Подвыражения также могут содержать оператор запятой, поэтому могут быть определены произвольно длинные последовательности выражений. С помощью оператора запятой можно вычислить несколько выражений в тех местах программы, где допускается только одно выражение. Типичным примером является увеличение нескольких индексов в цикле `for` (раздел 1.4.4.2):

```
++i, ++j
```

При использовании в качестве аргумента функции выражение с запятой нуждается в окружающих скобках; в противном случае запятая интерпретируется как разделитель аргументов функции.

1.3.6. Работа с памятью

Операторы `new` и `delete` соответственно выделяют и освобождают память (см. раздел 1.8.2).

1.3.7. Операторы доступа

C++ предоставляет несколько операторов для доступа к подструктурам, получения адреса переменной и разыменования (обращения к памяти по указанному адресу). Обсуждать эти операторы до того, как мы рассмотрим указатели и классы, не имеет никакого смысла. Так что мы просто отложим их описание до разделов, указанных в табл. 1.6.

Таблица 1.6. Операторы доступа

Операция	Выражение	Раздел
Выбор члена	`x.m`	2.2.3
Разыменующий выбор члена	`p->m`	2.2.3
Индексация	`x[i]`	1.8.1
Разыменование	`*x`	1.8.2
Разыменование члена	`x.*q`	2.2.3
Разыменование разыменованного члена	`p->*q`	2.2.3

1.3.8. Работа с типами

Операторы для работы с типами будут представлены в главе 5, “Метапрограммирование”, когда мы будем писать программы времени компиляции, работающие с типами. Доступные операторы перечислены в табл. 1.7.

Обратите внимание, что оператор `sizeof` при использовании с выражением является единственным, применимым без скобок. Оператор `alignof` появился в C++11; все прочие операторы имеются в языке по крайней мере начиная с C++98.

Таблица 1.7. Операторы для работы с типами

Операция	Выражение
Идентификация типа времени выполнения	`typeid(x)`
Идентификация типа	`typeid(t)`
Размер объекта	`sizeof(x)` или `sizeof x`
Размер типа	`sizeof(t)`
Количество аргументов	`sizeof...(p)`
Количество аргументов типа	`sizeof...(P)`
Выравнивание	`alignof(x)`
Выравнивание типа	`alignof(t)`

1.3.9. Обработка ошибок

Оператор `throw` используется для указания исключения во время выполнения (например, нехватки памяти) (см. раздел 1.6.2).

1.3.10. Перегрузка

Очень важным аспектом C++ является то, что программист может определить операторы для новых типов. Этот вопрос будет поясняться в разделе 2.7. Операторы встроенных типов изменить нельзя. Однако мы можем определить, как встроенные типы взаимодействуют с новыми типами, т.е. мы можем перегрузить смешанные операции наподобие удвоения матрицы.

Большинство операторов могут быть перегружены. Исключениями являются следующие:

`::` разрешение области видимости;
`.` выбор члена (может быть добавлен в C++17);
`.*` выбор члена через указатель;
`?:` условный оператор;
`sizeof` размер типа или объекта;
`sizeof...` количество аргументов;
`alignof` выравнивание памяти типа или объекта;
`typeid` идентификатор типа.

Перегрузка операторов в C++ дает нам очень большую свободу, и мы должны мудро ее использовать. Мы вернемся к этой теме в следующей главе, когда действительно будем перегружать операторы (подождите до раздела 2.7).

1.3.11. Приоритеты операторов

В табл. 1.8 дан краткий обзор приоритетов операторов. Для компактности мы объединяем обозначения для типов и выражений (например, `typeid`) и различные записи для `new` и `delete`. Символ `@=` представляет все вычисляющие присваивания, такие как `+=`, `*=` и т.д. Более подробно операторы и их семантика представлены в приложении B, "Определения языка", в табл. B.1.

Таблица 1.8. Приоритеты операторов

Приоритеты операторов			
class`::`*member*	*nspace*`::`*member*	`::`*name*	`::`*qualified-name*
object`.`*member* *type*(*expr_list*) `*_cast<`*type*`>(`*expr*`)`	*pointer*`->`*member* *lvalue*`++`	*expr*`[`*expr*`]` *lvalue*`--`	*expr*(*expr_list*) `typeid(`*type*`/`*expr*`)`
`sizeof` *expr* `++`*lvalue* `-`*expr* `new`... *type*...	`sizeof(`*type*`)` `--`*lvalue* `+`*expr* `delete []`$_{opt}$ *pointer*	`sizeof...(`*pack*`)` `~`*expr* `&`*lvalue* (*type*) *expr*	`alignof(`*type*`/`*expr*`)` `!`*expr* `*`*expr*
object`.*`*member_ptr*	*pointer*`->*`*member_ptr*		
expr `*` *expr*	*expr* `/` *expr*	*expr* `%` *expr*	
expr `+` *expr*	*expr* `-` *expr*		
expr `<<` *expr*	*expr* `>>` *expr*		
expr `<` *expr*	*expr* `<=` *expr*	*expr* `>` *expr*	*expr* `>=` *expr*
expr `==` *expr*	*expr* `!=` *expr*		
expr `&` *expr*			
expr `^` *expr*			
expr `|` *expr*			
expr `&&` *expr*			
expr `||` *expr*			
expr `?` *expr* `:` *expr*			
lvalue `=` *expr*	*lvalue* `@=` *expr*		
`throw` *expr*			
expr`,` *expr*			

1.3.12. Избегайте побочных эффектов!

Безумие, делая одно и то же снова и снова,
ожидать увидеть разные результаты.
— Неизвестный[5]

Ожидать разных результатов для одних и тех же входных данных в приложениях с побочными эффектами — вовсе не безумие. Напротив, очень трудно предсказать поведение программы, компоненты которой мешают одни другим. Кроме того, пожалуй, лучше уж иметь детерминированную программу с неправильным результатом, чем программу, которая дает правильный результат только иногда, поскольку последнюю обычно намного сложнее исправить.

В стандартной библиотеке C имеется функция для копирования строки (`strcpy`). Эта функция принимает указатели на первые символы источника и целевого объекта и копирует последующие символы до тех пор, пока не встретит нулевой символ. Ее можно реализовать с помощью единственного цикла, который имеет пустое тело (!) и выполняет копирование и инкремент как побочные эффекты теста продолжения выполнения цикла:

```
while(*tgt++ = *src++);
```

Выглядит страшно? Ну, самую малость. Однако это абсолютно законный код C++, хотя некоторые компиляторы могут и пожаловаться на него в педантичном режиме работы. Это неплохая разминка для мозгов — потратить некоторое время на размышления о приоритетах операторов, типах подвыражений и порядке вычислений.

Давайте рассмотрим кое-что попроще. Присвоим `i`-му элементу массива значение `i` и увеличим значение `i` для следующей итерации:

```
v[i]= i++;
```

Выглядит так, как будто нет никаких проблем. Но они есть — поведение этого выражения не определено. Почему? Постфиксный инкремент `i` гарантирует, что мы присвоим старое значение `i`, а затем увеличим эту переменную. Однако этот шаг может быть выполнен до того, как будет вычислено выражение `v[i]`, так что может быть выполнено присваивание значения `i` элементу `v[i+1]`.

Последний пример должен показать вам, что побочные эффекты, на первый взгляд, не всегда очевидны. Некоторые довольно сложные трюки могут сработать, а гораздо более простые — нет. Еще хуже, что что-то может работать некоторое время, до тех пор, пока кто-то не скомпилирует этот код на другом компиляторе или на новой версии компилятора, в котором окажутся измененными некоторые детали реализации.

[5] Ложно приписывалось Альберту Эйнштейну, Бенджамину Франклину и Марку Твену.

Первый фрагмент является примером отличных навыков программирования и доказательства того, что приоритет операторов имеет смысл — скобки в данном случае не нужны. Тем не менее такой стиль программирования не годится для современного C++. Стремление как можно больше сократить код восходит к временам раннего C, когда ввод был более сложным, с использованием механических (даже не электрических) клавиатур и перфораторов, да еще и без мониторов. При сегодняшних технологиях ввод немного большего количества букв проблемой быть не должен.

Еще одним неблагоприятным аспектом лаконичной реализации копирования является смешение различных задач: тестирования, модификации и обхода. В разработке программного обеспечения весьма важной концепцией является *разделение проблем*. Оно способствует повышению гибкости и снижению сложности. В данном случае мы хотим упростить понимание реализации копирования. Применение этого принципа к однострочной реализации копирования дает следующий код:

```
for (; *src; tgt++, src++)
    *tgt= *src;
*tgt= *src; // Копирование завершающего нулевого символа
```

Так мы четко разделяем три проблемы:

- тестирование: `*src`;
- изменение: `*tgt= *src`;
- обход: `tgt++, src++`.

Становится также более очевидным, что приращение выполняется над указателями, а тестирование и присваивание — над содержимым, на которое они указывают. Реализация оказывается не столь компактной, как раньше, но зато стало гораздо проще проверять правильность ввода. Целесообразно также сделать более очевидной проверку на равенство нулю (`*src != 0`).

Существует класс языков программирования, которые называются *функциональными языками*. Значения в таких языках нельзя изменить после того, как они были установлены. Очевидно, что C++ к таким языкам программирования не относится. Но мы получим большое преимущество при программировании в функциональном стиле, там, где это имеет смысл. Например, когда мы записываем присваивание, единственное, что должно измениться, — это переменная слева от знака присваивания. Поэтому следует заменить изменения константными выражениями, например `++i` на `i+1`. Правая сторона выражения без побочных эффектов облегчает как понимание поведения программы, так и оптимизацию кода компилятором. Как правило, более понятные программы обладают лучшим потенциалом для оптимизации.

1.4. Выражения и инструкции

C++ различает выражения и инструкции. Можно было бы вскользь заметить, что каждое выражение становится инструкцией после добавления точки с запятой, однако мы хотели бы обсудить эту тему немного подробнее.

1.4.1. Выражения

Давайте строить их рекурсивно снизу вверх. Любое имя переменной (`x`, `y`, `z`, …), константа или литерал — это выражение. Одно или несколько выражений, объединенных оператором, также представляют собой выражения, например `x + y` или `x * y + z`. В некоторых языках, таких как Pascal, присваивание является инструкцией. Но в C++ это выражение, например `x = y + z`. Поэтому оно может использоваться внутри другого присваивания: `x2 = x = y + z`. Присваивания вычисляются справа налево. Также являются выражениями операции ввода-вывода, такие как

```
std::cout << "x = " << x << "\n"
```

Вызов функции с аргументами, которые представляют собой выражения, также является выражением, например `abs(x)` или `abs(x * y + z)`. Таким образом, вызовы функций могут быть вложенными: `pow(abs(x),y)`. Обратите внимание, что вложенность была бы невозможна, если бы вызовы функций были инструкциями.

Поскольку присваивание является выражением, его можно использовать в качестве аргумента функции: `abs(x=y)`. Аргументами функции могут быть и операции ввода-вывода, например

```
print(std::cout << "x = " << x << "\n", "Я такой приколист!");
```

Нет нужды говорить, что это не особо удобочитаемо и вызовет больше путаницы, чем принесет пользы. Выражение в скобках также является выражением, как, например, `(x+y)`. Поскольку приоритет скобок выше приоритета любого оператора, мы всегда можем изменить порядок вычисления для удовлетворения наших потребностей. Так, в `x*(y+z)` сначала вычисляется сумма.

1.4.2. Инструкции

Инструкцией является любое из представленных выше выражений, за которым следует точка с запятой, например

```
x= y + z;
y= f(x + z) * 3.5;
```

Инструкция наподобие

```
y + z;
```

разрешена, несмотря на то что она (скорее всего) совершенно бесполезна. Во время выполнения программы вычисляется сумма `y` и `z`, которая затем игнорируется.

Современные компиляторы оптимизируют код и выбрасывают такие бесполезные вычисления. Однако не гарантируется, что такая инструкция всегда будет опущена. Если y или z является объектом пользовательского типа, то операция сложения также определена пользователем и может изменять, например, y, z или что-то еще. Очевидно, что это плохой (хотя и вполне законный в C++) стиль программирования (наличие скрытого побочного эффекта).

Отдельная точка с запятой является пустой инструкцией, так что после выражения мы можем ставить столько точек с запятой, сколько захотим. Некоторые инструкции не заканчиваются точкой с запятой, например определения функций. Если добавить к таким инструкциям точку с запятой, это не будет ошибкой — просто будет добавлена дополнительная пустая инструкция. Тем не менее некоторые компиляторы в педантичном режиме могут вывести соответствующее предупреждение. Любая последовательность инструкций, окруженная фигурными скобками, является *составной инструкцией*.

Объявления переменных и констант, которые мы видели раньше, также являются инструкциями. В качестве начального значения переменной или константы мы можем использовать любое выражение (за исключением другого присваивания или оператора запятой). Другие инструкции, которые мы рассмотрим, включают определения функций и классов, а также управляющие инструкции, с которыми вы познакомитесь в следующем разделе.

За исключением условного оператора поток выполнения программы управляется инструкциями. Здесь мы различаем ветвления и циклы.

1.4.3. Ветвление

В этом разделе рассмотрим различные возможности, которые позволяют нам выбрать ветвь выполнения программы.

1.4.3.1. Инструкция `if`

Это простейшая форма управления, смысл которой интуитивно очевиден, например

```
if (weight > 100.0)
    cout << " Достаточно тяжело.\n";
else
    cout << "Такой груз я донесу.\n";
```

Зачастую ветвь `else` не нужна и может быть опущена. Скажем, у нас есть значение в переменной x и нам необходимо его абсолютное значение:

```
if (x < 0.0)
    x= -x;
    // Теперь мы знаем, что x >= 0.0 (постусловие)
```

Ветви инструкции `if` являются областями видимости, что делает следующие инструкции неверными:

```
if (x < 0.0)
    int absx = -x;
else
    int absx = x;
cout << "|x| = " << absx << "\n"; // absx уже за областью видимости
```

Выше мы ввели две новые переменные, обе имеющие имя `absx`. Они не конфликтуют, поскольку находятся в разных областях видимости. Ни одна из них не существует после инструкции `if`, и обращение к `absx` в последней строке является ошибкой. На самом деле переменные, объявленные в ветвях, могут использоваться только внутри этих ветвей.

Каждая ветвь `if` состоит из одной инструкции. Для выполнения нескольких операций мы можем использовать фигурные скобки, как в приведенной реализации метода Кардано:

```
double D= q*q /4.0 + p*p*p /27.0;
if (D > 0.0) {
    double z1= ...;
    complex<double> z2 = ..., z3 = ...;
    ...
} else if (D == 0.0) {
    double z1 = ..., z2 = ..., z3 = ...;
    ...
} else { // D < 0.0
    complex<double> z1 = ..., z2 = ..., z3 = ...;
    ...
}
```

Всегда полезно вначале написать фигурные скобки. Многие руководства по стилю программирования требуют применять фигурные скобки даже для одной инструкции, тогда как автор предпочитает в этом случае обходиться без них. В любом случае настоятельно рекомендуется использовать отступ ветви для лучшей удобочитаемости.

Инструкции `if` могут быть вложенными; при этом каждое `else` связано с последним открытым `if`. Если вас интересуют конкретные примеры, обратитесь к разделу A.2.3. И вот еще один совет.

Совет

Хотя пробелы никак не влияют на компиляцию программ C++, отступы должны отражать структуру программы. Редакторы, понимающие C++ (наподобие интегрированной среды разработки Visual Studio или редактора emacs в режиме C++) и автоматически добавляющие отступы, очень облегчают структурное программирование. Всякий раз, когда строка имеет не тот отступ, который ожидается, скорее всего, имеет место не та вложенность, которая предполагалась.

1.4.3.2. Условное выражение

Хотя в этом разделе описываются инструкции, мы бы хотели поговорить здесь об условном выражении из-за его близости к инструкции `if`. Результатом выполнения

```
condition ? result_for_true : result_for_false
```

является второе подвыражение (т.е. `result_for_true`), если вычисление `condition` дает `true`, и `result_for_false` — в противном случае. Например,

```
min = x <= y ? x : y;
```

соответствует следующей инструкции `if`:

```
if (x <= y)
    min = x;
else
    min = y;
```

Для начинающих вторая версия может быть более удобочитаемой, но опытные программисты часто предпочитают первую версию из-за ее краткости.

`?:` является выражением и, следовательно, может использоваться для инициализации переменных:

```
int x = f(a),
    y = x < 0 ? -x : 2*x;
```

С помощью этого оператора легко вызвать функцию с выбором нескольких аргументов:

```
f(a, (x < 0 ? b : c), (y < 0 ? d : e));
```

Это будет выглядеть очень неуклюже при использовании инструкции `if`. Если вы этому не верите, попробуйте сами.

В большинстве случаев не важно, что используется: `if` или условное выражение. Так что используйте то, что удобнее для вас.

Забавно. Примером, в котором существенен выбор между `if` и `?:`, является операция `replace_copy` из стандартной библиотеки шаблонов (STL). Она обычно реализуется с помощью условного оператора, в то время как `if` было бы более обобщенным решением. Эта "ошибка" оставалась ненайденной около 10 лет и была обнаружена только с помощью автоматического анализа в кандидатской диссертации Джереми Сика (Jeremy Siek) [38].

1.4.3.3. Инструкция `switch`

Инструкция `switch` представляет собой особую разновидность инструкции `if`. Она обеспечивает краткую запись ситуации, когда для различных целочисленных значений должны выполняться разные действия:

```
switch ( op_code ) {
    case 0:  z = x + y; break;
    case 1:  z = x - y; cout << " разность\n"; break;
    case 2:
    case 3:  z = x * y; break;
    default: z = x / y;
}
```

Несколько неожиданным поведением является продолжение выполнения кода следующих вариантов, если только мы не прекратим выполнение с помощью инструкции `break`. Таким образом, для случаев 2 и 3 в нашем примере выполняются одни и те же операции. Расширенное использование `switch` можно найти в приложении А.2.4.

1.4.4. Циклы

1.4.4.1. Циклы `while` и `do-while`

Как предполагается в названии, цикл `while` повторяется до тех пор, пока выполняется некоторое условие. Давайте реализуем пример с рядом Коллатца, определяемым следующим образом.

Алгоритм 1.1. Ряд Коллатца

Вход. x_0

1 **while** $x_i \neq 1$ **do**

2 $$x_i = \begin{cases} 3x_{i-1} + 1, & \text{если } x_{i-1} \text{ нечетно;} \\ x_{i-1} / 2, & \text{если } x_{i-1} \text{ четно.} \end{cases}$$

Если не беспокоиться о переполнении, реализовать этот алгоритм очень просто с помощью цикла `while`:

```
int x= 19;
while (x != 1) {
    cout << x << '\n';
    if (x % 2 == 1) // Нечетно
        x= 3 * x + 1;
    else            // Четно
        x= x / 2;
}
```

Как и инструкцию `if`, цикл можно записывать без фигурных скобок, если в нем только одна инструкция.

C++ предлагает также цикл `do-while`. В этом случае условие продолжения выполнения цикла проверяется в конце итерации:

```
double eps= 0.001;
do {
    cout << "eps= " << eps << '\n';
```

```
    eps /= 2.0;
} while ( eps > 0.0001 );
```

Такой цикл выполняется как минимум один раз — даже для очень малого значения `eps` в нашем примере.

1.4.4.2. Цикл `for`

Наиболее распространенным циклом в C++ является цикл `for`. В качестве простого примера сложим два вектора[6] и выведем получившийся результат:

```
double v[3],
       w[] = {2., 4., 6.},
       x[] = {6., 5., 4};
for(int i = 0; i < 3; ++i)
    v[i]= w[i] + x[i];
for(int i = 0; i < 3; ++i)
    cout << "v[" << i << "]= " << v[i] << '\n';
```

Заголовок этого цикла состоит из трех компонентов:

- инициализация;
- условие *продолжения*;
- операция продвижения.

В приведенном выше примере мы видим типичный цикл `for`. В инициализации обычно объявляется и инициализируется (как правило, нулем — это начальный индекс большинства индексируемых структур данных) новая переменная. В условии обычно проверяется, меньше ли индекс цикла определенного значения, а последняя операция обычно увеличивает индексную переменную цикла. В этом примере мы выполняем преинкремент переменной цикла `i`. Для встроенных типов, таких как `int`, не имеет значения, пишем ли мы `++i` или `i++`. Однако это имеет значение для пользовательских типов, для которых постфиксный инкремент выполняет излишнее копирование (ср. с разделом 3.3.2.5). Чтобы быть последовательными, в этой книге мы всегда используем префиксный инкремент для индексов цикла.

Очень популярной ошибкой начинающих является запись условия как `i <= size(..)`. Поскольку индексы в C++ начинаются с нуля, индекс `i == size(..)` выходит за границы диапазона. Людям с опытом работы в Fortran или MATLAB необходимо некоторое время, чтобы привыкнуть к индексации “с нуля”. Для многих индексация, начинающаяся с единицы, кажется более естественной, а кроме того, она используется в математической литературе. Однако расчеты индексов и адресов почти всегда проще вести при нулевом начальном индексе.

[6] Позже мы рассмотрим истинные классы векторов, а пока что возьмем простые массивы.

В качестве еще одного примера вычислим ряд Тейлора для экспоненциальной функции:

$$e^x = \sum_{i=0}^{\infty} \frac{x^n}{n!}$$

до десятого члена:

```
double x= 2.0, xn= 1.0, exp_x = 1.0;
unsigned long fac= 1;
for(unsigned long i = 1; i <= 10; ++i) {
    xn  *= x;
    fac *= i;
    exp_x += xn / fac;
    cout << "e^x = " << exp_x << '\n';
}
```

Здесь оказывается гораздо проще вычислить нулевой член отдельно и начать цикл с первого члена. Мы также использовали в условии сравнение "меньше или равно" для того, чтобы гарантировать вычисление члена $x^{10}/10!$.

Цикл `for` в C++ очень гибкий. Инициализирующая часть может быть любым выражением, объявлением переменной или пустой. Можно вводить в этой части несколько новых переменных одного и того же типа. Это может использоваться для того, чтобы избежать повторения вычислений в условии одной и той же операции, например

```
for(int i = xyz.begin(), end = xyz.end(); i < end; ++i) ...
```

Переменные, объявленные в части инициализации, видимы только в цикле и скрывают переменные с теми же именами вне цикла.

Условие может быть любым выражением, преобразуемым в `bool`. Пустое условие всегда истинно и цикл в этом случае повторяется бесконечно. Он может быть прекращен внутри тела цикла — этот способ мы рассмотрим в следующем разделе. Мы уже упоминали, что индекс цикла обычно увеличивается в третье подвыражении `for`. В принципе, мы можем изменять его и в теле цикла. Однако код станет гораздо понятнее, если делать это только в заголовке цикла. С другой стороны, нет никаких ограничений, требующих использования только одной переменной, и увеличения ее значения только на 1. Мы можем изменять столько переменных, сколько захотим, используя оператор запятой (раздел 1.3.5), причем изменять любым способом, например

```
for(int i = 0, j = 0, p = 1; ...; ++i, j+= 4, p*= 2) ...
```

Это, конечно, сложнее, чем простое увеличение индекса цикла, но все равно более удобочитаемо, чем объявление/изменение индексных переменных перед циклом или внутри его тела.

1.4.4.3. Цикл `for` для диапазона C++11

Очень компактная запись получается при использовании новой возможности C++, которая именуется *циклом `for` для диапазона*. Мы поговорим о нем более подробно при рассмотрении концепции итераторов (раздел 4.1.2).

Пока что мы будем рассматривать его как сжатую форму записи для выполнения итерации над всеми записями массива или другого контейнера:

```
int primes []= {2, 3, 5, 7, 11, 13, 17, 19};
for(int i : primes )
    std::cout << i << " ";
```

Этот код выводит все числа массива, разделенные пробелами.

1.4.4.4. Управление циклом

Имеются две инструкции, которые изменяют обычную работу цикла:

- `break`;
- `continue`.

Инструкция `break` полностью завершает цикл, а `continue` завершает только текущую итерацию и заставляет цикл перейти к следующей итерации, например

```
for (...; ...; ...) {
    ...
    if (dx == 0.0) continue;
    x+= dx;
    ...
    if (r < eps) break;
    ...
}
```

В приведенном выше примере мы считаем, что оставшаяся часть итерации не нужна, если `dx == 0.0`. В некоторых итеративных вычислениях в средине итерации может стать понятно, что работа уже выполнена (здесь при `r < eps`).

1.4.5. `goto`

Все ветвления и циклы внутренне реализуются с помощью переходов. C++ предоставляет инструкцию безусловного перехода `goto`. Однако учтите следующий совет.

Совет

Не используйте `goto`! Нигде и никогда!

Применение goto в C++ более ограничено, чем в C (например, мы не можем выполнять переход через инициализации), но по-прежнему может разрушить структуру нашей программы.

Написание программ без использования goto называется *структурным программированием*. Однако в настоящее время этот термин используется редко, так как применение этого стиля в высококачественном программном обеспечении подразумевается само собой.

1.5. Функции

Функции являются важными строительными блоками программ на C++. Первый пример, который мы видели, — это функция main в первой же рассмотренной программе. Об этой функции мы поговорим подробнее в разделе 1.5.5.

Общий вид функции в C++ выглядит как

```
[ inline ] возвращаемый_тип имя_функции( список_аргументов )
{
    Тело функции
}
```

В этом разделе мы рассмотрим эти компоненты более подробно.

1.5.1. Аргументы

C++ различает два способа передачи аргументов в функции — по значению и по ссылке.

1.5.1.1. Передача аргументов по значению

Когда мы передаем аргумент в функцию, по умолчанию создается его копия. Например, следующая инструкция увеличивает x, но внешний по отношению к функции код этого не видит:

```
void increment(int x)
{
    x++;
}

int main ()
{
    int i = 4;
    increment(i); // Не приводит к увеличению i
    cout << "i = " << i << '\n';
}
```

Эта программа выводит на экран значение 4. Операция x++ в функции increment увеличивает только локальную копию i, но не саму переменную i. Такая передача аргументов в функции называется *передачей по значению*.

1.5.1.2. Передача аргументов по ссылке

Чтобы иметь возможность модифицировать параметры функций, аргументы в функцию должны *передаваться по ссылке*:

```
void increment(int & x)
{
    x++;
}
```

Теперь увеличивается сама переменная, так что будет выведено значение 5, как и ожидалось. Мы будем обсуждать ссылки более подробно в разделе 1.8.4.

Временные переменные — такие, как результаты операций — не могут быть переданы по ссылке:

```
increment(i + 9); // Ошибка: временное значение
```

поскольку мы в любом случае не в состоянии вычислить `(i + 9)++`. Для того чтобы вызвать такую функцию с некоторым временным значением, его следует сначала сохранить в переменной, и уже ее передать в функцию.

Большие структуры данных, такие как векторы или матрицы, почти всегда передаются по ссылке, чтобы избежать дорогостоящей операции копирования:

```
double two_norm(vector & v) { ... }
```

Такая операция, как вычисление нормы, не должна изменять свой аргумент. Но передача вектора по ссылке несет риск случайной его перезаписи. Чтобы гарантировать, что наш вектор не меняется (и не копируется), мы передаем его как константную ссылку:

```
double two_norm(const vector & v) { ... }
```

Если мы попытаемся изменить `v` в этой функции, компилятор сообщит об ошибке.

И передача аргумента по значению, и передача как константной ссылки обеспечивают неизменность аргумента, но различными средствами.

- Аргументы, переданные по значению, могут изменяться в функции, поскольку функция работает с копией[7].
- При передаче константных ссылок мы работаем непосредственно с передаваемым аргументом, но все операции, которые могут его изменить, при этом запрещены. В частности, такие аргументы не могут находиться в левой части присваивания или быть переданы другим функциям через неконстантные ссылки (фактически левая часть присваивания также является неконстантной ссылкой).

[7] В предположении корректного копирования. Пользовательские типы с некорректными реализациями копирования могут подрывать целостность переданных данных.

В отличие от изменяемых[8] ссылок константные ссылки позволяют передавать временные значения:

```
alpha = two_norm(v + w);
```

Это, правда, не совсем согласуется с дизайном языка, но зато намного облегчает жизнь программистам.

1.5.1.3. Аргументы по умолчанию

Если аргумент обычно имеет одно и то же значение, его можно объявить со значением по умолчанию. Скажем, если мы реализуем функцию, которая вычисляет корень *n*-й степени, но в основном применяется для вычисления квадратного корня, то мы можем написать

```
double root(double x, int degree = 2) { ... }
```

Эта функция может быть вызвана как с двумя, так и с одним аргументом:

```
x = root(3.5, 3);
y = root(7.0);    // То же, что и root(7.0,2)
```

Можно объявить несколько значений по умолчанию, но только в конце списка аргументов. Другими словами, после аргумента со значением по умолчанию мы не можем указывать аргумент без такового.

Значения по умолчанию полезны при добавлении дополнительных параметров. Давайте предположим, что у нас есть функция, которая рисует круги:

```
draw_circle(int x, int y, float radius);
```

Все эти круги черные. Позже мы добавляем возможность указывать цвет кругов:

```
draw_circle(int x, int y, float radius, color c= black);
```

Благодаря аргументу по умолчанию нам не нужно переделывать наше приложение, поскольку вызовы `draw_circle` с тремя аргументами по-прежнему будут корректно работать.

1.5.2. Возврат результатов

В приведенных ранее примерах мы возвращали только `double` или `int`. Это хорошо ведущие себя типы возвращаемых значений. Теперь мы рассмотрим крайности — очень большие возвращаемые данные или их отсутствие.

1.5.2.1. Возврат большого количества данных

Функции, вычисляющие новые значения больших структур данных, оказываются более трудными. Детали мы рассмотрим позже, а пока только вскользь рассмотрим эту тему. Хорошая новость заключается в том, что компиляторы достаточно умны, чтобы во многих случаях не создавать копию возвращаемого

[8] Слово *изменяемый* (mutable) в этой книге используется как синоним слова “неконстантный”. В C++ имеется также ключевое слово `mutable` (раздел 2.6.3), которое мы практически не используем.

значения (см. раздел 2.3.5.3). Кроме того, копирование позволяет избежать семантика перемещения (раздел 2.3.5), когда происходит непосредственный захват временных данных. Современные библиотеки вообще избегают возвращения больших структур данных, используя методы, именуемые шаблонами выражений, и откладывают вычисления до тех пор, пока не станет известно, где будет храниться результат (раздел 5.3.2). В любом случае мы не должны возвращать ссылки на локальные переменные функции (раздел 1.8.6).

1.5.2.2. Отсутствие возвращаемого значения

Синтаксически каждая функция должна возвращать что-то, даже если возвращать нечего. Эта дилемма решается с помощью имени пустого типа — `void`. Например, функция, которая просто выводит значение `x`, не должна возвращать что-либо:

```
void print_x (int x)
{
    std::cout << "Значение x = " << x << '\n';
}
```

`void` не является реальным типом и используется как заполнитель, позволяющий нам обойтись без возвращаемого значения. Мы не можем определить объект типа `void`:

```
void nothing; // Ошибка: объектов void не бывает
```

Функция `void` может быть завершена раньше с помощью инструкции `return` без аргумента:

```
void heavy_compute ( const vector & x, double eps, vector & y)
{
    for (...) {
        ...
        if (two_norm(y) < eps)
            return;
    }
}
```

1.5.3. Встраивание

Вызов функции является относительно дорогой операцией: нужно сохранить регистры, скопировать аргументы в стек и т.д. Чтобы избежать этого лишнего труда, компилятор может встраивать вызовы функций. В этом случае вызов функции заменяется операциями, содержащимися в функции. Программист может попросить компилятор сделать это с помощью соответствующего ключевого слова:

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

Однако компилятор не обязан выполнять встраивание. Наоборот, он может встраивать функции без ключевого слова `inline`, если это представляется

перспективным с точки зрения производительности. Тем не менее объявление `inline` имеет свое применение для включения функции в несколько единиц компиляции, которые мы будем обсуждать в разделе 7.2.3.2.

1.5.4. Перегрузка

Функции в C++ могут совместно использовать одно и то же имя, если объявления их параметров достаточно различны. Это называется *перегрузкой функций*. Давайте сначала рассмотрим пример:

```
#include <iostream>
#include <cmath>

int divide(int a, int b) {
    return a / b;
}

float divide(float a, float b) {
    return std::floor(a / b);
}

int main () {
    int x= 5, y= 2;
    float n= 5.0, m= 2.0;
    std::cout << divide(x,y) << std::endl;
    std::cout << divide(n,m) << std::endl;
    std::cout << divide(x,m) << std::endl; // Ошибка: неоднозначность
}
```

Здесь мы определили функцию `divide` дважды: с параметрами `int` и с параметрами `double`. Когда мы вызываем `divide`, компилятор выполняет *разрешение перегрузки*.

1. Имеется ли перегрузка, в которой типы аргументов в точности соответствуют переданным значениям? Если да, использовать ее, в противном случае:
2. Имеются ли перегрузки с соответствием типов после выполнения преобразований типов? Сколько?
   - 0. Ошибка — подходящая функция не найдена.
   - 1. Использовать эту функцию.
   - > 1. Ошибка — неоднозначный вызов.

Как это относится к нашему примеру? Вызовы `divide(x,y)` и `divide(n,m)` имеют точные соответствия. Для `divide(x,m)` нет точно соответствующей перегрузки и есть две перегрузки, соответствующие после *неявного преобразования*, так что мы сталкиваемся неоднозначностью.

Термин “неявное преобразование” требует пояснений. Мы уже видели, что числовые типы могут преобразовываться один в другой. Это неявные преобразования, как демонстрируется в примере. Когда позже мы определим собственные типы, мы сможем реализовать преобразования из других типов в наши, и обратно, из наших новых типов в существующие. Такие преобразования могут быть объявлены как явные (`explicit`), и тогда они применимы только когда преобразование запрошено явно, но не для соответствия аргументов функций.

⇨ `c++11/overload_testing.cpp`

Формулируя более официально, перегрузки функций должны различаться *сигнатурами*. Сигнатура в C++ состоит из

- имени функции;
- количества аргументов, именуемого *арностью*;
- типов аргументов (в указанном в объявлении порядке).

Перегрузки, различающиеся только возвращаемым типом или именами аргументов, имеют одинаковые сигнатуры и рассматриваются как (запрещенные) переопределения:

```
void f(int x) {}
void f(int y) {} // Переопределение: отличие только в имени аргумента
long f(int x) {} // Переопределение: отличие только в типе возврата
```

Функции с разными именами или арностью безоговорочно являются различными. Наличие символа ссылки превращает тип аргумента в другой тип аргумента (таким образом, `f(int)` и `f(int&)` могут сосуществовать). Следующие три перегрузки имеют разные сигнатуры:

```
void f(int x) {}
void f(int & x) {}
void f(const int & x) {}
```

Этот фрагмент кода компилируется без ошибок. Однако при вызове `f` начинаются проблемы:

```
int i= 3;
const int ci= 4;
f(3);
f(i);
f(ci);
```

Все три вызова функции неоднозначны, потому что в каждом случае лучшими соответствиями являются первая перегрузка с аргументом, передаваемым по значению, и одна из перегрузок с передачей аргумента по ссылке. Смешивание перегрузок с передачей по значению и по ссылке почти всегда ведет к проблемам. Таким образом, когда одна перегрузка имеет ссылочный аргумент, то и другие

перегрузки должны иметь ссылочные аргументы. В нашем примере мы можем навести порядок, удалив первую перегрузку с передачей аргумента по значению. Тогда вызовы `f(3)` и `f(ci)` будут разрешаться в перегрузки с константной ссылкой, а `f(i)` — с изменяемой.

1.5.5. Функция `main`

Функция `main` принципиально не отличается от любой другой функции. В стандарте имеются две разрешенные сигнатуры:

```
int main()
```

и

```
int main(int argc, char * argv[])
```

Последняя запись эквивалентна следующей:

```
int main( int argc, char ** argv)
```

Параметр `argv` содержит список аргументов, а `argc` — его длину. Первый аргумент (`argv[0]`) в большинстве систем содержит имя выполняемого файла (которое может отличаться от имени исходного файла). Чтобы поработать с аргументами, напишем короткую программу под названием `argc_argv_test`:

```
int main (int argc, char * argv [])
{
    for(int i= 0; i < argc; ++i)
        cout << argv[i] << '\n';
    return 0;
}
```

Вызов этой программы со следующими параметрами командной строки

```
argc_argv_test first second third fourth
```

дает вывод на экран

```
argc_argv_test
first
second
third
fourth
```

Как видите, каждый пробел в командной строке разделяет аргументы. Функция `main` возвращает целое число в качестве кода завершения, который указывает, закончилось ли выполнение программы успешно. Значение 0 (или макрос `EXIT_SUCCESS` из заголовочного файла `<cstdlib>`) указывает на успешное завершение, а любое другое значение — на сбой. Стандарт разрешает опустить оператор `return` в функции `main`. В этом случае компилятором автоматически вставляется `return 0;`. Некоторые дополнительные сведения можно найти в разделе A.2.5.

1.6. Обработка ошибок

Оплошность не становится ошибкой,
пока вы не отказываетесь ее исправить.
— Джон Ф. Кеннеди

Два главных способа обработки неожиданного поведения программы в C++ — утверждения и исключения. Первые предназначены для обнаружения ошибок в программировании, а вторые — для исключительных ситуаций, которые мешают продолжению правильной работы программы. Честно говоря, это различие далеко не всегда очевидно.

1.6.1. Утверждения

Макрос `assert` из заголовочного файла `<cassert>` унаследован от языка программирования C, но он полезен и в C++. Он вычисляет переданное ему выражение, и если результат равен `false`, то выполнение программы немедленно завершается. Этот макрос должен использоваться для обнаружения ошибок программирования. Допустим, мы реализуем крутой алгоритм вычисления квадратного корня из неотрицательных действительных чисел. Из математики мы знаем, что результат является неотрицательным числом. В противном случае в нашем расчете что-то сделано неверно:

```
#include <cassert>

double square_root ( double x)
{
    check_somehow(x >= 0);
    ...
    assert(result >= 0.0);
    return result;
}
```

Как реализовать первоначальную проверку — оставим этот вопрос пока что открытым. Если полученный нами результат отрицательный, выполнение программы прекратится с выводом на экран сообщения наподобие следующего:

```
assert_test: assert_test.cpp:10: double square_root(double):
Утверждение 'result >= 0.0' не выполнено.
```

Тот факт, что полученный результат оказался меньше нуля, говорит о том, что наша реализация содержит ошибку, и мы должны исправить ее, прежде чем использовать разработанную функцию в серьезных приложениях.

После того как мы исправили ошибку, может возникнуть соблазн удалить `assert`. Не стоит этого делать. Может быть, в один прекрасный день мы изменим реализацию; после этого желательно провести все тесты заново. Утверждения для постусловий, по сути, представляют собой модульные мини-тесты.

Большое преимущество `assert` заключается в том, что все такие проверки можно отключить единственным объявлением макроса. Перед включением `<cassert>` в исходный текст можно просто определить `NDEBUG`:

```
#define NDEBUG
#include <cassert>
```

Все утверждения при этом будут отключены, т.е. они не будут вносить в выполнимый файл никакого кода. Вместо изменения исходного текста программы каждый раз, когда мы переключаемся между отладочной и производственной версиями, лучше и проще объявить макрос `NDEBUG` с помощью флагов компилятора (обычно `-D` в Linux и `/D` в Windows):

```
g++ my_app .cpp -o my_app -O3 -DNDEBUG
```

Программное обеспечение с утверждениями в критических местах кода может оказаться замедленным в два или более раз, если не отключить их в режиме выпуска. Хорошие системы построения, такие как `CMake`, автоматически включают флаг компиляции `-DNDEBUG` в режиме выпуска.

Поскольку утверждения можно так легко отключить, прислушайтесь к следующей рекомендации.

Оборонительное программирование

Тестируйте столько свойств, сколько вы в состоянии протестировать.

Даже если вы уверены, что некоторое свойство явно выполняется в вашей реализации, напишите утверждение. Иногда система не ведет себя в точности так, как предполагалось, компилятор может содержать ошибку (это очень редкое, но возможное событие) или мы сделали что-то немного отличающееся от того, что намеревались сделать первоначально. Словом, независимо от того, насколько мы тщательно и обдуманно пишем код, рано или поздно какое-то из утверждений может сработать. В случае, когда таких свойств так много, что функциональность начинает запутываться и затормаживаться тестами, их можно перенести в другую функцию.

Ответственные программисты всегда реализуют большие наборы тестов. Тем не менее они не гарантируют, что программа будет работать при любых обстоятельствах. Приложение может работать в течение многих лет, как часы, но в один далеко не прекрасный день выйти из строя. В этой ситуации мы можем запустить приложение в режиме отладки, когда будут включены все утверждения, и в большинстве случаев они окажутся большим подспорьем в поиске причины аварийной ситуации. Однако это требует, чтобы аварийная ситуация была воспроизводимой и чтобы программа в более медленном отладочном режиме могла достичь критического раздела за разумное время.

1.6.2. Исключения

В предыдущем разделе мы рассмотрели, как утверждения помогают обнаружить ошибки программирования. Однако есть много критических ситуаций, которые мы не можем предотвратить каким бы то ни было разумным программированием, например файлы, которые должна читать наша программа, могут оказаться удаленными. Или, например, наша программа потребует больше памяти, чем доступно на данном компьютере. Некоторые проблемы теоретически могут быть предотвращены, но практические усилия оказываются непропорционально высоки. Например, проверить, является ли матрица регулярной, можно, но для этого придется выполнить работу, которая может оказаться большей, чем работа над фактической задачей. В таких случаях обычно эффективнее попытаться решить задачу и убедиться в отсутствии *исключений* во время вычислений.

1.6.2.1. Мотивация

Прежде чем проиллюстрировать обработку ошибок в старом стиле, мы представим вам нашего антигероя Герберта[9], который является гениальным математиком и рассматривает программирование как необходимое зло для демонстрации того, как великолепно работают его алгоритмы. Он научился программировать, как настоящий мужчина, и невосприимчив к новомодным бессмыслицам современного программирования.

Его любимый подход к решению вычислительных задач — возвращать код ошибки (как это делает функция `main`). Скажем, мы хотим прочитать матрицу из файла и проверяем, на месте ли интересующий нас файл. Если его нет, то мы возвращаем код ошибки 1:

```
int read_matrix_file ( const char * fname, ...)
{
    fstream f( fname );
    if (!f.is_open())
        return 1;
    ...
    return 0;
}
```

Таким образом мы проверяем все, что может пойти не так, и сообщаем об этом вызывающей функции с помощью соответствующего кода ошибки. Это нормальное решение, когда вызывающая функция проверяет возвращенное значение и реагирует соответствующим образом. Но что происходит, если вызывающая функция просто игнорирует код возврата? Да ничего! Программа продолжает работать и позже может столкнуться с аварийной ситуацией из-за абсурдных данных или, что еще хуже, получить бессмысленные результаты, которые небрежные люди могут использовать для построения автомобилей или самолетов. Конечно, создатели автомобилей и самолетов не столь небрежны, но в более реалистичном

[9] Автор приносит извинения всем Гербертам, читающим книгу, за использование их имени.

программном обеспечении даже аккуратные программисты не в состоянии отследить каждую крошечную деталь.

Тем не менее эти соображения не в состоянии убедить динозавров от программирования, таких как Герберт. “Вы не только глупы настолько, что передаете моей прекрасно реализованной функции несуществующий файл, но еще и не проверяете код возврата! Все неправильно делаете именно вы, а не я”.

Другой недостаток кодов ошибок заключается в том, что мы не можем просто вернуть результаты наших вычислений и должны передавать их через ссылочные аргументы. Это не позволяет нам просто создавать выражения с участием результатов наших вычислений. Другой способ — вернуть из функции результат вычислений, а код ошибки передать через ссылочный аргумент — оказывается ничуть не менее громоздким.

1.6.2.2. Генерация исключения

Наилучший подход — сгенерировать исключение с помощью оператора `throw`:

```
matrix read_matrix_file ( const char * fname, ...)
{
    fstream f( fname );
    if (!f.is_open())
        throw "Невозможно открыть файл.";
    ...
}
```

В этой версии мы генерируем исключение. Вызывающее приложение теперь обязано отреагировать на него — в противном случае программа аварийно завершит работу.

Преимущество исключений над кодами ошибок заключается в том, что мы имеем дело с проблемой только там, где можем ее обработать. Например, функция `read_matrix_file` может не иметь возможности обработать отсутствие файла. В этой ситуации код просто генерирует исключение. Таким образом, нам не нужно запутывать нашу программу, возвращая коды ошибок. В случае исключения оно просто будет передано соответствующему обработчику исключений. В нашем случае такая обработка может выполняться в пользовательском интерфейсе, где у пользователя запросят новый файл. Таким образом, исключения позволяют сделать исходный текст более удобочитаемым и при этом обеспечить более надежную обработку ошибок.

C++ позволяет сгенерировать исключение любого вида: строки, числа, пользовательские типы и т.д. Однако лучше всего определить специальные типы исключений или использовать таковые из стандартной библиотеки:

```
struct cannot_open_file {};

void read_matrix_file( const char * fname, ...)
{
    fstream f( fname );
    if (!f.is_open())
```

```
        throw cannot_open_file {};
    ...
}
```

Здесь мы ввели наш собственный тип исключения. В главе 2, "Классы", мы подробно рассмотрим, как могут быть определены классы. В приведенном выше примере мы определили пустой класс, для которого необходимы только открывающие и закрывающие скобки и точка с запятой. Крупные проекты обычно создают целую иерархию типов исключений, которые часто являются производными (глава 6, "Объектно-ориентированное программирование") от `std::exception`.

1.6.2.3. Перехват исключений

Чтобы отреагировать на исключение, его следует перехватить. Это делается с помощью блока `try-catch`:

```
try {
    ...
}
catch (e1_type & e1) { ... }
catch (e2_type & e2) { ... }
```

Там, где мы ожидаем проблему, которую мы в состоянии решить (или по крайней мере что-то сделать), мы открываем блок `try`. После закрывающей скобки мы можем перехватить исключение и начать "спасательную операцию" в зависимости от типа перехваченного исключения и, возможно, его значения. Рекомендуется перехватывать исключения по ссылке [45, совет 73], в особенности когда речь идет о полиморфных типах (определение 6.1 из раздела 6.1.3). Когда в блоке сгенерировано исключение, выполняется первый блок `catch` с типом, соответствующим типу исключения. Прочие блоки `catch` того же типа (или подтипов, раздел 6.1.1) игнорируются. Блок `catch` с троеточием перехватывает все исключения:

```
try {
    ...
}
catch ( e1_type & e1) { ... }
catch ( e2_type & e2) { ... }
catch (... ) {   // Перехват всех остальных исключений
}
```

Очевидно, что такой обработчик любых исключений должен быть последним.

Если мы не в состоянии сделать ничего иного, можно перехватить исключение хотя бы для того, чтобы вывести информативное сообщение об ошибке перед тем, как завершить выполнение программы:

```
try {
    A = read_matrix_file("does_not_exist.dat");
} catch(cannot_open_file & e) {
    cerr << "Ваш файл не существует! Закрываем лавочку.\n";
    exit( EXIT_FAILURE );
}
```

После того как исключение перехватывается, считается, что проблема решена и продолжается выполнение инструкций, находящихся после блока `catch`. Выше для прекращения выполнения мы использовали функцию `exit` из заголовочного файла <cstdlib>. Эта функция завершает выполнение программы, даже если мы находимся не в функции `main`. Она должна использоваться только тогда, когда дальнейшее выполнение программы слишком опасно и нет никакой надежды, что вызывающая функция сможет справиться с этим исключением.

В качестве альтернативы можно продолжить — после вывода сообщения или каких-то частичных действий — обработку исключения в охватывающих блоках, генерируя его заново:

```
try {
    A= read_matrix_file("does_not_exist.dat");
} catch(cannot_open_file & e) {
    cerr << "Этого файла здесь нет! Спасайте.\n";
    throw e;
}
```

В нашем случае мы уже находимся в функции `main`, так что в стеке вызовов нет ничего, что могло бы перехватить такое регенерированное исключение. Для повторной генерации того же исключения, которое было перехвачено, можно воспользоваться сокращенной записью:

```
} catch(cannot_open_file & e) {
    ...
    throw;
}
```

Предпочтительнее использовать эту сокращенную запись, так как она меньше подвержена ошибкам и более ясно показывает, что мы хотим заново сгенерировать исходное исключение. Проигнорировать исключение легко с помощью пустого блока:

```
} catch(cannot_open_file &) {} // Файл не нужен, продолжаем
```

В действительности пока что наша обработка исключений не решает проблему отсутствующего файла. Решать ее можно по-разному. Например, если имя файла указывает пользователь, мы можем докучать ему до тех пор, пока не получим требуемый файл:

```
bool keep_trying = true;
do {
    char fname [80]; // Лучше использовать std::string
    cout << "Введите имя файла: ";
    cin >> fname;
    try {
        A= read_matrix_file(fname);
        ...
        keep_trying = false;
```

```
    } catch(cannot_open_file & e) {
        cout << "Невозможно открыть файл. Попробуйте еще раз!\n";
    } catch (...)
        cout << "Что-то пошло не так. Попробуйте еще раз!\n";
    }
} while(keep_trying);
```

Когда мы добрались до конца `try`-блока, мы знаем, что исключение не было сгенерировано, так что работа успешно выполнена. В противном случае окажемся в одном из `catch`-блоков, и значение переменной `keep_trying` останется равным `true`.

Большим преимуществом исключений является то, что проблемы, которые не могут быть решены в контексте, в котором они обнаружены, можно отложить на более поздний срок. Пример из практики автора касается LU-разложения. Оно не может быть выполнено для сингулярной матрицы. С этим ничего нельзя поделать. Однако в случае, когда факторизация является частью итеративных вычислений, мы могли бы продолжать итерации и без факторизации. Хотя такая обработка ошибок могла быть реализована и традиционными методами, исключения позволяют нам осуществить ее гораздо более удобочитаемо и элегантно. Мы можем запрограммировать факторизацию для регулярной матрицы, а при обнаружении сингулярности генерировать исключение. Вызывающая функция, перехватив исключение, может принять соответствующие данному контексту меры — насколько это возможно.

1.6.2.4. Кто может генерировать исключения

C++11

В C++03 разрешено указывать, какие типы исключений может генерировать некоторая функция. Не вдаваясь в подробности, скажем, что эти спецификации оказались не очень полезными, и в настоящее время считаются устаревшими.

В C++11 в язык добавлен новый квалификатор для указания того факта, что данная функция не должна генерировать исключения, например:

```
double square_root( double x) noexcept { ... }
```

Преимущество такого квалификатора в том, что вызывающий код не обязан проверять после вызова `square_root`, не было ли сгенерировано исключение. Если исключение, несмотря на наличие квалификатора, все же сгенерировано, программа завершается.

Генерируется ли исключение в шаблонных функциях, может зависеть от того, генерируют ли исключения аргументы типов. Для корректной обработки этой ситуации `noexcept` может зависеть от условий времени компиляции (см. раздел 5.2.2).

Что более предпочтительно — использование утверждений или исключений — вопрос непростой, и у нас нет короткого и однозначного ответа на него. Пока что этот вопрос, скорее всего, не будет вас беспокоить. Поэтому мы отложим обсуждение до раздела A.2.6 и оставим окончательное решение за вами, когда вы прочтете этот раздел.

1.6.3. Статические утверждения C++11

Программные ошибки, которые могут быть обнаружены уже во время компиляции, можно проверять с помощью статического утверждения `static_assert`. В этом случае компилятор выдает сообщение об ошибке и прекращает компиляцию. Детальное описание и рассмотрение конкретного примера пока что не имеет смысла; мы отложим его до раздела 5.2.5.

1.7. Ввод-вывод

Для выполнения операций ввода-вывода в последовательные устройства наподобие терминала или клавиатуры C++ использует удобную абстракцию, именуемую потоками. Поток — это объект, в который программа может вставлять символы или извлекать их оттуда. Стандартная библиотека C++ содержит заголовочный файл `<iostream>`, в котором объявляются стандартные потоки ввода и вывода.

1.7.1. Стандартный вывод

По умолчанию стандартный вывод из программы записывается на экран, и в программе мы можем получить доступ к потоку с именем `cout`. Он используется с оператором вставки << (такой же, как и сдвиг влево). Мы уже видели, что этот оператор может использоваться в одной инструкции более одного раза. Это особенно полезно, когда мы хотим выводить комбинацию из текста, переменных и констант, например:

```
cout << "Квадратный корень из " << x << " = " << sqrt (x) << endl;
```

Вывод при этом имеет вид наподобие

```
Квадратный корень из 5 = 2.23607
```

`endl` генерирует символ новой строки. Альтернативой применения `endl` является использование символа `\n`. Для эффективности вывод может быть буферизован, и в этом отношении `endl` и `\n` различаются. Первый из них приводит к сбросу буфера, а последний — нет. Сброс буфера может помочь нам при отладке (без отладчика), чтобы определить, между какими двумя выводами происходит аварийное завершение программы. Но при записи большого количества исходного текста в файл очистка буфера после каждой строки может существенно замедлить работу.

К счастью, оператор вставки имеет относительно низкий приоритет, так что арифметические операции могут быть записаны непосредственно, без использования скобок:

```
std::cout << "11 * 19 = " << 11 * 19 << std::endl;
```

Все сравнения, логические и побитовые операции должны быть сгруппированы с помощью скобок. То же самое относится и к условному оператору:

```
std::cout << (age > 65 ? "Он мудр\n" : "Да он просто мальчишка!\n");
```

Если вы забудете скобки, компилятор напомнит вам о них (предлагая расшифровать загадочные сообщения об ошибках).

1.7.2. Стандартный ввод

Стандартным устройством ввода обычно является клавиатура. Обработка стандартного ввода в C++ осуществляется с помощью перегруженного оператора извлечения >> из потока `cin`:

```
int age;
std::cin >> age;
```

`std::cin` считывает символы из устройства ввода и интерпретирует их как значение типа переменной (здесь — `int`), в которую его сохраняет (здесь — `age`). Ввод с клавиатуры обрабатывается после нажатия клавиши <Enter>.

Мы также можем запросить у `cin` несколько данных от пользователя:

```
std::cin >> width >> length;
```

Это эквивалентно записи

```
std::cin >> width;
std::cin >> length;
```

В обоих случаях пользователь должен предоставить два значения: одно — для ширины и другое — для длины. Они могут быть разделены любым корректным пробельным разделителем — пробелом, знаком табуляции или символом новой строки.

1.7.3. Ввод-вывод в файлы

C++ предоставляет следующие классы для выполнения файлового ввода и вывода.

`ofstream`	Запись в файлы
`ifstream`	Чтение из файлов
`fstream`	Чтение из файлов и запись в них

Мы можем использовать файловые потоки таким же образом, как `cin` и `cout`, с той лишь разницей, что мы должны связать эти потоки с физическими файлами. Вот пример такого использования:

```
#include <fstream>
int main()
{
```

```
    std::ofstream square_file;
    square_file.open("squares.txt");
    for (int i= 0; i < 10; ++i)
        square_file << i << "^2 = " << i*i << std::endl;
    square_file.close();
}
```

Этот код создает файл с именем `squares.txt` (или перезаписывает его, если таковой уже существует) и выполняет запись в него — так же, как выполняется запись в `cout`. C++ устанавливает общую концепцию потока, которой удовлетворяют как файл вывода, так и поток `std::cout`. Это означает, что мы можем писать в файл все, что можно писать в `std::cout`, и наоборот. Определяя `operator <<` для нового типа, мы делаем это один раз для типа `ostream` (раздел 2.7.3), и он может работать с консолью, с файлами и с любым другим потоком вывода.

В качестве альтернативы для неявного открытия файла можно передать его имя как аргумент конструктора потока. Этот файл так же неявно закрывается, когда `square_file` выходит из области видимости[10], в данном случае — в конце функции `main`. Краткая версия предыдущей программы имеет следующий вид:

```
#include <fstream>

int main()
{
    std::ofstream square_file("squares.txt");
    for (int i= 0; i < 10; ++i)
        square_file << i << "^2 = " << i*i << std::endl;
}
```

Как обычно, мы предпочитаем короткую запись. Явное открытие и закрытие необходимо только в случае, когда файл сначала объявлен, а открывается по какой-либо причине позже. Аналогично явный вызов `close` требуется только тогда, когда файл должен быть закрыт, прежде чем он выйдет из области видимости.

1.7.4. Обобщенная концепция потоков

Потоки не ограничиваются экранами, клавиатурами и файлами; любой класс может использоваться как поток, если он является производным[11] от `istream`, `ostream` или `iostream` и предоставляет реализации функций этих классов. Например, Boost.Asio предоставляет потоки TCP/IP, а Boost.IOStream — альтернативу описанному выше вводу-выводу. Стандартная библиотека содержит `stringstream`, который может использоваться для создания строки из любых типов, которые могут быть выведены в поток. Метод `str()` класса `stringstream` возвращает внутреннюю строку потока типа `string`.

[10] Благодаря мощной технологии под названием “RAII”, о которой мы поговорим в разделе 2.4.2.1.

[11] О том, что это означает, вы узнаете из главы 6, “Объектно-ориентированное программирование”. Здесь мы просто отметим, что выходные потоки технически порождены из `std::ostream`.

Мы можем написать функции вывода, которые принимают поток любого вида, используя изменяемую ссылку на ostream в качестве аргумента:

```
#include <iostream>
#include <fstream>
#include <sstream>

void write_something(std::ostream & os)
{
    os << "Знаете ли вы, что 3 * 3 = " << 3 * 3 << std::endl;
}

int main (int argc, char * argv [])
{
    std::ofstream myfile("example.txt");
    std::stringstream mysstream;
    write_something(std::cout);
    write_something(myfile);
    write_something(mysstream);
    std::cout << "mysstream = "   // В этой строке есть
              << mysstream.str(); // символ новой строки
}
```

Аналогично универсальный ввод может быть реализован с помощью istream.

1.7.5. Форматирование

➪ c++03/formatting.cpp

Потоки ввода-вывода форматируются с помощью так называемых манипуляторов ввода-вывода, которые описаны в заголовочном файле <iomanip>. По умолчанию C++ выводит только несколько цифр чисел с плавающей точкой. Мы можем повысить точность:

```
double pi= M_PI;
cout << "pi = " << pi << '\n';
cout << "pi = " << setprecision(16) << pi << '\n';
```

и получить более точное значение числа:

```
pi = 3.14159
pi = 3.141592653589793
```

В разделе 4.3.1 мы покажем, как можно корректировать точность до количества цифр, представимых типом.

Когда мы выводим таблицы, векторы или матрицы, желательно выравнивать значения для удобочитаемости. Для этого мы можем задать ширину выходных данных:

```
cout << "pi = " << setw(30) << pi << '\n';
```

Результат имеет следующий вид:

```
pi =              3.141592653589793
```

setw влияет только на следующий вывод данных, в то время как setprecision влияет на все последующие выводы (числовых данных), подобно прочим манипуляторам. Предоставленная ширина рассматривается как минимальная, и, если выводимому значению требуется больше знакомест, таблицы превращаются в нечто уродливое.

Мы можем запросить выравнивание по левому краю и заполнение пустого пространства выбранным нами символом, например –:

```
cout << "pi = "  << setfill('-') << left
     << setw(30) << pi << '\n';
```

Результат имеет следующий вид:

```
pi = 3.141592653589793-------------
```

Еще один способ форматирования — непосредственная установка флагов. Некоторые реже используемые настройки форматирования можно применять только таким образом, как, например, нужно ли показывать знак для положительных значений. Кроме того, можно заставить выводить значения в "научной" записи — нормализованном представлении с показателем степени:

```
cout.setf(ios_base::showpos);
cout << "pi = " << scientific << pi << '\n';
```

Результат имеет следующий вид:

```
pi = +3.1415926535897931e+00
```

Целые числа могут быть выведены в восьмеричной и шестнадцатеричной системах счисления:

```
cout << "63 восьмеричное      = " << oct << 63 << ".\n";
cout << "63 шестнадцатеричное = " << hex << 63 << ".\n";
cout << "63 десятичное        = " << dec << 63 << ".\n";
```

Результат имеет следующий вид:

```
63 восьмеричное      = 77.
63 шестнадцатеричное = 3f.
63 десятичное        = 63.
```

Логические значения по умолчанию выводятся как целые числа 0 и 1. При необходимости можно потребовать выводить их как true и false:

```
cout << "pi < 3 = " << (pi < 3) << '\n';
cout << "pi < 3 = " << boolalpha << (pi < 3) << '\n';
```

Наконец можно сбросить все измененные флаги форматирования:

```
int old_precision = cout.precision();
cout << setprecision(16)
...
cout.unsetf(ios_base::adjustfield | ios_base::basefield
          | ios_base::floatfield  | ios_base::showpos
          | ios_base::boolalpha);
cout.precision(old_precision);
```

Каждый флаг представляет бит в переменной состояния. Чтобы включить несколько флагов, их можно объединить с помощью операции побитового ИЛИ.

1.7.6. Обработка ошибок ввода-вывода

Определимся сразу: ввод-вывод в C++ не защищен от ошибок (не говоря уж о дураках). Об ошибках может быть сообщено различными способами, и наша обработка ошибок должна соответствовать этим способам. Давайте рассмотрим следующий программный пример:

```
int main ()
{
    std::ifstream infile("some_missing_file.xyz");
    int i;
    double d;
    infile >> i >> d;
    std::cout << "i = " << i << ", d = " << d << '\n';
    infile.close();
}
```

Хотя файл не существует, операция открытия не приводит к аварийному завершению, и выполнение продолжается. Мы даже можем читать из несуществующего файла. Излишне говорить, что "прочитанные" значения `i` и `d` не имеют смысла:

```
i = 1, d = 2.3452e-310
```

По умолчанию потоки не генерируют исключений. Причины такого поведения исторические: потоки старше исключений, поэтому такое поведение оставлено для обратной совместимости, чтобы не нарушать работоспособность старых программ.

Чтобы быть уверенным, что все операции успешны, мы должны проверять флаги ошибок, в принципе, после каждой операции ввода-вывода. Следующая программа запрашивает у пользователя новое имя файла до тех пор, пока файл не будет открыт. После прочтения его содержимого мы вновь проверяем успешность выполненной операции:

```
int main()
{
    std::ifstream infile;
    std::string filename{"some_missing_file.xyz"};
    bool opened = false;
    while(!opened) {
```

```
            infile.open(filename);
            if (infile.good()) {
                opened = true;
            } else {
                std::cout << "Файл '" << filename
                          << "' не существует, введите новое имя: ";
                std::cin >> filename;
            }
        }
        int i;
        double d;
        infile >> i >> d;
        if (infile.good())
            std::cout << "i = " << i << ", d = " << d << '\n';
        else
            std::cout << "Ошибка чтения содержимого файла.\n";
        infile.close();
    }
```

Из этого простого примера видно, что написание надежных приложений с использованием файлового ввода-вывода может потребовать определенных трудов.

Если мы хотим использовать исключения, можно разрешить их во время выполнения каждого потока:

```
cin.exceptions(ios_base::badbit|ios_base::failbit);
cout.exceptions(ios_base::badbit|ios_base::failbit);
std::ifstream infile("f.txt");
infile.exceptions(ios_base::badbit|ios_base::failbit);
```

Потоки генерируют исключения каждый раз, когда происходит сбой операции или когда они находятся в "плохом" (bad) состоянии. Исключения могут также генерироваться по достижении (непредвиденного) конца файла. Однако в конец файла более удобно определять с помощью проверки (например, `while(!f.eof())`).

В приведенном выше примере исключения для `infile` включаются только после открытия файла (или попытки открытия). Для проверки результата операции открытия с помощью исключений необходимо сначала создать поток, затем включить исключения и только после этого явно открыть файл. Включение исключений дает нам как минимум гарантию, что если программа корректно завешает работу, то все операции ввода-вывода выполняются успешно. Мы можем сделать нашу программу более надежной, перехватывая исключения, которые могут быть сгенерированы.

Исключения файлового ввода-вывода только частично защищают нас от ошибок. Например, приведенная далее программа, очевидно, неверна (не совпадают типы и не разделены числа):

```
void with_io_exceptions(ios & io)
{
    io.exceptions(ios_base::badbit|ios_base::failbit);
}
```

```
int main ()
{
    std::ofstream outfile;
    with_io_exceptions(outfile);
    outfile.open("f.txt");

    double o1= 5.2, o2= 6.2;
    outfile << o1 << o2 << std::endl; // Нет разделителя
    outfile.close();

    std::ifstream infile;
    with_io_exceptions(infile);
    infile.open("f.txt");
    int i1, i2;
    char c;
    infile >> i1 >> c >> i2;          // Несоответствие типов
    std::cout << "i1 = " << i1 << ", i2 = " << i2 << "\n";
}
```

Тем не менее генерации исключений не происходит, а программа выводит на экран следующий результат:

```
i1 = 5, i2 = 26
```

Как мы все знаем, тестирование не доказывает правильность программы. Это еще более очевидно для ввода-вывода. Входной поток считывает входящие символы и передает их в качестве значения переменной соответствующего типа, например `int` для `i1`. Он останавливается на первом же символе, который не может быть частью значения: сначала — на точке для значения `i1` типа `int`. Если мы читаем после этого еще один `int`, происходит ошибка, поскольку пустая строка не может рассматриваться как значение типа `int`. Но ведь мы считываем `char`, которому вполне соответствует точка. В ходе анализа ввода для `i2` мы считываем сначала дробную часть от `o1`, а затем — целую часть от `o2`, прежде чем получим символ, который не может принадлежать значению типа `int`.

К сожалению, не каждое нарушение грамматических правил на практике вызывает исключение. `0.3` в ходе анализа в качестве `int` дает нуль (в то время как при считывании следующего значения, вероятно, произойдет ошибка); `-5` в ходе анализа в качестве 32-битового беззнакового целого дает 4 294 967 291. Похоже, в потоках ввода-вывода пока не применяется принцип сужения (если он вообще когда-либо будет применяться — из-за обеспечения обратной совместимости).

В любом случае часть приложения, связанная с вводом-выводом, требует особого внимания. Числа должны быть корректно разделены, например, пробелами и считываться в переменные тех же типов, из которых были записаны. Когда вывод осуществляется с ветвлением, так что формат файла может варьироваться, код чтения входных данных оказывается значительно более сложным и даже может быть неоднозначным.

Имеются еще две разновидности ввода-вывода, которые мы хотим отметить: бинарный и ввод-вывод в стиле C. Заинтересовавшийся читатель найдет их в разделах A.2.7 и A.2.8 соответственно. Вы можете обратиться к ним позже, когда у вас возникнет такая необходимость.

1.8. Массивы, указатели и ссылки

1.8.1. Массивы

Поддержка встроенных массивов C++ имеет определенные ограничения и кое в чем странное поведение. Тем не менее мы считаем, что каждый программист C++ должен из знать и быть осведомлен о возможных проблемах.

Массив объявляется следующим образом:

```
int x[10];
```

Переменная `x` представляет собой массив с десятью элементами типа `int`. В стандарте C++ размер массива должен быть константой и известен во время компиляции. Некоторые компиляторы (например, `gcc`) поддерживают размеры времени выполнения.

Обращение к элементам массива осуществляется с помощью квадратных скобок. `x[i]` является ссылкой на *i*-й элемент массива `x`. Первым элементом массива является `x[0]`; последним — `x[9]`. Массивы могут быть инициализированы при определении:

```
float v[] = {1.0, 2.0, 3.0}, w[] = {7.0, 8.0, 9.0};
```

В этом случае размер массива выводится компилятором.

Список инициализации в C++11 не может быть подвергнут сужающему преобразованию. На практике это редко вызывает проблемы. Например, код

```
int v[] = {1.0, 2.0, 3.0}; // Ошибка в C++11: сужение
```

корректен в C++03, но не в C++11, поскольку преобразование литерала с плавающей точкой в `int` потенциально ведет к потере точности. Впрочем, мы в любом случае не будем писать такой уродливый код.

Операции над массивами обычно выполняются в циклах; например, вычисление $x = v - 3w$ как векторная операция выполняется с помощью следующего кода:

```
float x[3];
for(int i= 0; i < 3; ++i)
    x[i] = v[i] - 3.0*w[i];
```

Можно определять массивы и более высоких размерностей:

```
float A[7][9];  // Матрица 7×9
int q[3][2][3]; // Массив 3×2×3
```

Язык не предоставляет операции линейной алгебры над массивами. Реализации, основанные на массивах, безвкусны и подвержены ошибкам. Например, функция для векторного сложения будет выглядеть следующим образом:

```
void vector_add(unsigned size, const double v1[],
                const double v2[], double s[])
{
    for(unsigned i = 0; i < size; ++i)
        s[i]= v1[i] + v2[i];
}
```

Обратите внимание, что мы передаем размер массивов в качестве первого параметра функции, так как параметры-массивы не содержат информации о размере[12]. В этом случае вызывающая функция отвечает за передачу правильного размера массивов:

```
int main ()
{
    double x[] = {2, 3, 4}, y[] = {4, 2, 0}, sum[3];
    vector_add(3, x, y, sum);
    ...
}
```

Поскольку размер массива известен во время компиляции, мы можем вычислить его путем деления размера массива в байтах на размер одного элемента:

```
vector_add(sizeof x / sizeof x[0], x, y, sum);
```

При таком старом интерфейсе мы также не в состоянии проверить соответствие размеров наших массивов. К сожалению, библиотеки C и Fortran с такими интерфейсами, в которых сведения о размере передаются как аргументы функции, по-прежнему используются и сегодня. Малейшая ошибка пользователя приводит к большим неприятностям, и могут потребоваться огромные усилия для того, чтобы отследить причину сбоев. Поэтому в этой книге мы покажем, как можно реализовать собственное математическое программное обеспечение, более простое в использовании и менее подверженное ошибкам. Хочется верить, что в будущие стандарты C++ будет включено больше высшей математики, в особенности — библиотека для решения задач линейной алгебры.

Массивы обладают двумя основными недостатками.

- При обращении к массиву не выполняется проверка индексов, так что можно легко выйти за его пределы, и программа завершит работу с ошибкой нарушения сегментации памяти. Это далеко не худший случай; по крайней мере, при этом мы видим, что что-то пошло не так. Неверное обращение к массиву может также испортить наши данные. Программа при этом продолжает работать и давать совершенно неправильные результаты — с последствиями,

[12] При передаче массивов более высоких размерностей только первое измерение может быть открытым, в то время как все остальные должны быть известны во время компиляции. Однако такие программы попросту опасны (и уродливы), и C++ предлагает куда лучшие решения.

которые вы можете себе представить сами. Мы даже могли бы перезаписать сам программный код. Тогда данные интерпретируются компьютером как машинные команды, что может привести к любой бессмыслице.

- Размер массива должен быть известен во время компиляции[13]. Например, пусть у нас есть массив, сохраненный в файл, и нам надо считать его обратно в память:

```
ifstream ifs("some_array.dat");
ifs >> size;
float v[size]; // Ошибка: размер не известен во время компиляции
```

Этот код не будет работать, так как размер массива должен быть известен во время компиляции.

Первая проблема может быть решена только с помощью массивов нового типа, а вторая — с помощью динамического выделения памяти. Это приводит нас к указателям.

1.8.2. Указатели

Указатель представляет собой переменную, которая содержит адрес памяти. Этот адрес может быть адресом другой переменной, который мы получаем с помощью оператора получения адреса (например, `&x`) или динамически выделенной памяти. Давайте начнем с последнего случая и рассмотрим создание массива динамического размера.

```
int* y = new int[10];
```

Здесь выделена память для массива из десяти элементов типа `int`. Размер массива может быть выбран во время выполнения. Мы можем также реализовать пример с чтением вектора из предыдущего раздела:

```
ifstream ifs("some_array.dat");
int size;
ifs >> size;
float * v = new float[size];
for(int i = 0; i < size; ++i)
    ifs >> v[i];
```

Указатели так же опасны, как и массивы: доступ к данным за пределами диапазона может вызвать сбой программы или тихую порчу данных. При работе с динамически выделенными массивами за хранение размера массива отвечает программист.

Кроме того, программист несет ответственность за освобождение памяти, когда она больше не нужна. Это делается следующим образом:

[13] Некоторые компиляторы поддерживают значения времени выполнения в качестве размеров массивов. Поскольку такое поведение другими компиляторами не гарантируется, в переносимом программном обеспечении его следует избегать. Рассматривалось включение этой возможности в стандарт C++14, но оно было отложено, так как не удалось полностью прояснить все тонкие моменты.

```
delete[] v;
```

В ряде современных языков явное освобождение памяти не требуется, так как при применяемой в них технологии *сборки мусора* для освобождения памяти достаточно просто сбросить переменную-указатель, присвоив ей нулевое значение (память освобождается и когда переменная выходит из области видимости и уничтожается). В разделе А.2.9 мы даем некоторые комментарии по сборке мусора, которые сводятся к тому, что можно неплохо работать и без нее.

Поскольку массивы как параметры функции внутренне рассматриваются как указатели, функция `vector_add` из раздела 1.8.1 может работать и с указателями:

```
int main (int argc, char * argv [])
{
    double *x   = new double[3],
           *y   = new double[3],
           *sum = new double[3];
    for(unsigned i = 0; i < 3; ++i)
        x[i] = i+2, y[i] = 4-2*i;
    vector_add(3,x,y,sum);
    ...
}
```

С указателями мы не можем использовать трюк с `sizeof`; он даст нам лишь размер в байтах самого указателя, который, конечно, не зависит от количества записей. Во всех остальных отношениях указатели и массивы являются взаимозаменяемыми в большинстве ситуаций: как указатель может быть передан в качестве аргумента-массива (как в предыдущем листинге), так и массив может быть передан в качестве аргумента-указателя. Единственное место, где они действительно различаются — это определение. В то время как определение массива размера `n` резервирует память для `n` записей, определение указателя оставляет за собой место только для хранения адреса.

Так как мы начинали с массивов, выполним еще один шаг перед тем как перейти к использованию указателей. Простейшее применение указателя — выделение памяти для одного элемента данных:

```
int* ip = new int;
```

Освобождение памяти в этом случае имеет следующий вид:

```
delete ip;
```

Обратите внимание на двойственность выделения и освобождения памяти. Выделение одного объекта требует освобождения одного объекта, а выделение массива требует освобождения массива. В противном случае система времени выполнения будет обрабатывать освобождение памяти неправильно, и, скорее всего, это приведет к аварийному завершению. Указатели также могут ссылаться на другие переменные:

```
int  i   = 3;
int* ip2 = &i;
```

Оператор & принимает объект и возвращает его адрес. Обратный оператор * принимает адрес и возвращает объект:

```
int j = *ip2;
```

Эта операция называется *разыменованием*. Учитывая приоритеты операторов и правила грамматики, смысл символа * как разыменования или умножения невозможно спутать — по крайней мере, компилятору.

C++11 Неинициализированные указатели содержат случайные значения (набор битов, содержащийся в соответствующем месте памяти). Применение неинициализированных указателей может привести к самым разнообразным ошибкам. Чтобы явно показать, что указатель ни на что не указывает, ему надо присвоить значение nullptr:

```
int* ip3 = nullptr; // >= C++11
int* ip4 {};        // >= C++11
```

или (в старых компиляторах)

```
int* ip3 = 0;       // В C++11 и более поздних лучше не использовать
int* ip4 = NULL;    // То же самое
```

C++11 Гарантируется, что адрес 0 никогда не будет использован в приложениях, так что его можно безопасно использовать для обозначения того, что этот указатель является пустым (не указывает ни на что). Тем не менее литерал 0 не совсем ясно выражает это намерение и может привести к неоднозначности при перегрузке функций. Макрос NULL ничуть не лучше. Он просто возвращает значение 0. В C++11 вводится новое ключевое слово nullptr, являющееся литералом указателя. Это значение может присваиваться или сравниваться с указателями любого типа. Поскольку его нельзя спутать с другими типами, а его имя говорит само за себя, его применение предпочтительнее, чем использование любых других обозначений. Инициализация с помощью пустого списка в фигурных скобках также устанавливает указатель равным nullptr.

Наибольшая опасность указателей — *утечки памяти*. Например, наш массив y оказывается слишком маленьким, и мы хотим присвоить этой переменной новый массив:

```
int* y = new int[15];
```

Теперь мы можем использовать больше памяти с помощью y. Отлично! Но что происходит с памятью, выделенной ранее? Она все еще выделена, но у нас больше нет к ней доступа. Мы не можем даже освободить ее, потому что для этого нужно знать ее адрес. Эта память оказывается потерянной для нашей программы. Только тогда, когда программа завершится, операционная система сможет ее освободить. В нашем примере мы потеряли только 40 байт из нескольких гигабайтов, которые мы можем использовать. Но если такая потеря происходит в итеративном

процессе, объем неиспользуемой памяти непрерывно увеличивается до тех пор, пока в какой-то момент не будет исчерпана вся (виртуальная) память.

Даже если трата памяти впустую не является критической для разрабатываемого приложения, когда мы пишем научное программное обеспечение высокого качества, утечки памяти совершенно неприемлемы. Если ваше программное обеспечение предназначено для использования многими пользователями, рано или поздно утечка будет обнаружена, и даже если она не приведет к сбоям программы, она приведет к потере доверия к ее качеству, а в конечном счете — к отказу пользователей от вашего программного обеспечения. К счастью, имеются инструменты, которые могут помочь найти утечки памяти (см. раздел Б.3).

Описанные проблемы, возникающие при работе с указателями, показаны не для того, чтобы вас запугать. И мы вовсе не рекомендуем отказываться от указателей. Многое можно сделать только с помощью указателей — списки, очереди, деревья, графы и т.д. Однако указатели должны использоваться с осторожностью, чтобы полностью избежать всех серьезных проблем, упомянутых выше.

Минимизировать ошибки, связанные с указателями, можно с помощью трех стратегий.

Использование стандартных контейнеров из стандартной библиотеки или других проверенных библиотек. `std::vector` из стандартной библиотеки предоставляет нам всю функциональность динамических массивов, включая изменение размеров и проверку выхода за границы диапазона, а также автоматическое освобождение памяти.

Инкапсуляция управления динамической памятью в классах. В таком случае мы должны справиться со всеми проблемами только один раз — в классе[14]. Если вся память, выделенная объектом, освобождается при уничтожении объекта, то не важно, как часто мы выделяем память. Если у нас есть 738 объектов с динамической памятью, то она будет освобождена 738 раз. Память должна выделяться при создании объекта и освобождаться при его уничтожении. Этот принцип называется *Захват ресурса есть инициализация* (Resource acquisition is initialization — RAII). И наоборот, если мы вызвали `new` 738 раз, частично в цикле и ветвях, то как мы можем быть уверены, что мы вызовем `delete` тоже точно 738 раз? Мы знаем, что имеется соответствующий инструментарий, но ошибки легче не допускать, чем исправлять[15]. Конечно, идея инкапсуляции тоже не "защищена от дурака", но она требует гораздо меньших усилий для корректной работы, чем разбрасывание обычных указателей по всей программе. Идиому RAII мы обсудим более подробно в разделе 2.4.2.1.

Использование интеллектуальных указателей, о которых мы поговорим в следующем разделе, 1.8.3.

[14] Обычно объектов в программе гораздо больше, чем классов; в противном случае что-то не так в самом дизайне программы.

[15] Кроме того, соответствующий инструментарий может указать на отсутствие утечки только для данного конкретного выполнения программы, но она может проявиться для других входных данных.

Указатели служат двум целям:

- указание на объекты,
- управление динамической памятью.

Проблема с обычными указателями заключается в том, что мы не знаем, указывает ли указатель на некоторые данные или он также отвечает за освобождение памяти, когда та больше не нужна. Чтобы иметь возможность такого явного различия на уровне типов, мы можем использовать *интеллектуальные указатели.*

1.8.3. Интеллектуальные указатели

C++11

В C++11 введены три новых типа интеллектуальных указателей: `unique_ptr`, `shared_ptr` и `weak_ptr`. Имеющийся в C++03 интеллектуальный указатель `auto_ptr` обычно считается неудачной попыткой на пути к `unique_ptr`, поскольку язык в то время еще не был к этому готов. Использовать интеллектуальный указатель `auto_ptr` больше не следует. Все интеллектуальные указатели определяются в заголовочном файле `<memory>`. Если вы не можете использовать на своей платформе возможности C++11, примите к сведению, что достойной заменой являются интеллектуальные указатели из библиотеки Boost.

1.8.3.1. `unique_ptr`

C++11

Имя этого указателя говорит о том, что он является *единственным указателем* на свои данные. Он может быть использован, по сути, так же, как и обычный указатель:

```
#include <memory>

int main ()
{
    unique_ptr<double> dp{new double};
    *dp= 7;
    ...
}
```

Основным отличием от обычных указателей является то, что память автоматически освобождается при уничтожении указателя. Таким образом, ему нельзя присваивать адреса, которые не выделены динамически:

```
double d;
unique_ptr<double> dd{&d}; // Ошибка: неверное удаление
```

Деструктор указателя `dd` попытается удалить `d`.

Уникальные указатели нельзя присваивать другим типам указателей или неявно преобразовывать в другие типы. Для получения хранящегося в интеллектуальном указателе обычного указателя можно использовать функцию-член `get`:

```
double * raw_dp = dp.get();
```

Его нельзя даже присваивать другому уникальному указателю:

```
unique_ptr<double> dp2{dp}; // Ошибка: копирование запрещено
dp2 = dp;                   // То же самое
```

Его можно только перемещать:

```
unique_ptr<double> dp2{move(dp)}, dp3;
dp3 = move(dp2);
```

Семантику перемещения мы рассмотрим в разделе 2.3.5. Пока же просто скажем, что в то время как копирование дублирует данные, *перемещение* передает данные от источника к целевому объекту. В нашем примере владение памятью, на которую указывает интеллектуальный указатель, сначала передается от dp к dp2, а затем к dp3. После этих передач dp и dp2 имеют значения nullptr, а выделенная память будет освобождена деструктором dp3. Таким же образом владение памятью передается и когда unique_ptr возвращается из функции. В следующем примере dp3 получает во владение память, выделенную в f():

```
std::unique_ptr<double> f()
{
    return std::unique_ptr<double>{new double};
}

int main ()
{
    unique_ptr<double> dp3;
    dp3 = f();
}
```

В этом случае move() не требуется, поскольку результат функции является временным значением, которое будет перемещено (детальнее, опять же, — в разделе 2.3.5).

Уникальный указатель имеет специальную реализацию[16] для массивов. Это необходимо для правильного освобождения памяти (с помощью delete[]). Кроме того, специализация обеспечивает доступ к элементам массива:

```
unique_ptr<double[]> da{new double[3]};
for(unsigned i = 0; i < 3; ++i)
    da[i] = i+2;
```

В то же время operator* для массивов недоступен.

Важным преимуществом unique_ptr является то, что он не имеет абсолютно никаких накладных расходов по сравнению с обычными указателями — ни в смысле производительности, ни в смысле расходуемой памяти.

Дополнительные материалы. Важной возможностью уникальных указателей является возможность предоставления собственной функции освобождения памяти (*удалителя*); подробности приведены в [26, §5.2.5f], [43, §34.3.1] и в онлайн-руководствах (например, cppreference.com).

[16] Специализации будут рассмотрены в разделах 3.6.1 и 3.6.3.

1.8.3.2. shared_ptr

C++11

Как становится понятно из названия, интеллектуальный указатель shared_ptr управляет памятью, которая совместно используется несколькими объектами (и каждый хранит указатель на нее). Такая память автоматически освобождается, как только не остается ни одного shared_ptr, указывающего на нее. Это может значительно упростить программу, особенно при использовании сложных структур данных. Чрезвычайно важной областью применения этих интеллектуальных указателей является параллелизм: память освобождается автоматически, когда все потоки завершают свой доступ к ней.

В отличие от unique_ptr, shared_ptr может быть скопирован сколько угодно раз, например

```
shared_ptr<double> f()
{
    shared_ptr<double> p1{new double};
    shared_ptr<double> p2{new double}, p3 = p2;
    cout << "p3.use_count () = " << p3.use_count() << endl;
    return p3;
}

int main ()
{
    shared_ptr<double> p = f();
    cout << "p.use_count() = " << p.use_count() << endl;
}
```

В этом примере мы выделяем память для двух значений типа double — для p1 и p2. Указатель p2 копируется в p3, так что оба они указывают на одну и ту же память, как показано на рис. 1.1.

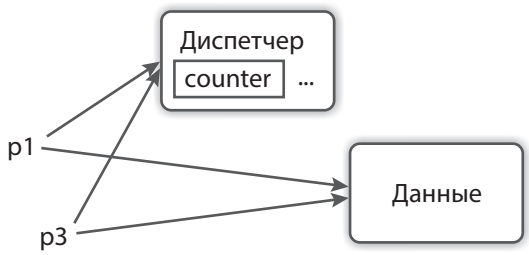

Рис. 1.1. shared_ptr в памяти

Мы можем увидеть это, выводя значение функции use_count, которая дает значение счетчика counter (рис. 1.1):

```
p3.use_count () = 2
p.use_count () = 1
```

При завершении функции `f()` указатели уничтожаются, и память, на которую указывает `p1`, освобождается (она так и не была использована). Второй выделенный блок памяти продолжает существовать, поскольку `p` из функции `main` продолжает на нее указывать.

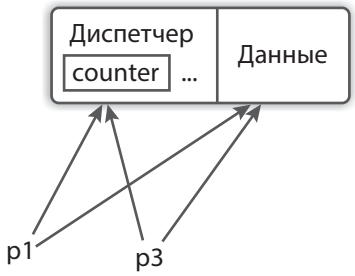

Рис. 1.2. `shared_ptr` в памяти после вызова `make_shared`

По возможности интеллектуальный указатель `shared_ptr` следует создавать с помощью вызова `make_shared`:

```
shared_ptr<double> p1= make_shared<double>();
```

Тогда данные диспетчера и данные, на которые указывает интеллектуальный указатель, сохраняются в памяти вместе, как показано на рис. 1.2, так что кеширование памяти работает эффективнее. Поскольку `make_shared` возвращает `shared_ptr`, для простоты можно использовать автоматический вывод типа (раздел 3.4.1):

```
auto p1 = make_shared<double>();
```

Мы должны признать, что интеллектуальный указатель `shared_ptr` имеет некоторые накладные расходы памяти и производительности. С другой стороны, упрощение наших программ благодаря применению `shared_ptr` в большинстве случаев стоит некоторых небольших накладных расходов.

Дополнительные материалы. Об удалителях и прочих деталях реализации `shared_ptr` читайте в [26, §5.2], [43, §34.3.2] и в онлайн-руководствах (например, `cppreference.com`).

1.8.3.3. `weak_ptr` C++11

Применение `shared_ptr` не лишено проблем — так, наличие *циклических ссылок* препятствует освобождению памяти. "Разбить" такие циклы и решить тем самым проблему может интеллектуальный указатель `weak_ptr`. Он не претендует ни на право владения памятью, ни даже на совместное использование. Пока что мы просто упоминаем о них — для полноты изложения, но если вам потребуются эти интеллектуальные указатели, информацию и них вы найдете в [26, §5.2.2], [43, §34.3.3] и на сайте `cppreference.com`.

Для динамического управления памятью указателям нет альтернативы. Если требуется только ссылаться на другие объекты, можно воспользоваться другой возможностью языка, именуемой *ссылками* (сюрприз, сюрприз!), о которой мы и поговорим в следующем разделе.

1.8.4. Ссылки

Следующий код объявляет ссылку:

```
int  i = 5;
int& j = i;
j = 4;
std::cout << "i = " << i << '\n';
```

Переменная `j` ссылается на `i`. Изменение `j` приведет к изменению `i` (и наоборот), как показано в примере. `i` и `j` всегда будут иметь одно и то же значение. Ссылку можно рассматривать как псевдоним, который вводит новое имя для существующего объекта или подобъекта. Всякий раз, когда мы определяем ссылку, мы тут же должны явно указать, на что она ссылается (в отличие от указателей, которые могут получить свое значение позже). После объявления ссылка не может измениться и начать указывать на другую переменную.

Пока что не прозвучало ничего особо полезного. Но ссылки являются чрезвычайно полезными в качестве аргументов функций (раздел 1.5), для обращения к частям других объектов (например, к седьмому элементу вектора) и для построения представлений (см., например, раздел 5.2.3).

C++11 В качестве компромисса между указателями и ссылками новый стандарт предлагает класс `reference_wrapper`, который ведет себя аналогично ссылкам, но позволяет избежать некоторых из их ограничений. Например, он может использоваться внутри контейнеров (см. раздел 4.4.2).

1.8.5. Сравнение указателей и ссылок

Основным преимуществом указателей перед ссылками является возможность динамического управления памятью и вычисления адресов. С другой стороны, ссылки могут ссылаться только на корректные местоположения в памяти[17]. Таким образом, они не грозят утечками памяти (если только вы не будете ими злоупотреблять), а кроме того, используют такую же запись, как и объект, на который они ссылаются. К сожалению, практически невозможно создать контейнеры ссылок.

Словом, ссылки не являются безопасной панацеей, но они, тем не менее, намного менее подвержены ошибкам, чем указатели. Указатели следует использовать только при работе с динамической памятью, например когда мы динамически создаем такие структуры данных, как списки или деревья. Даже тогда мы должны делать это с помощью тщательно протестированных типов или инкапсулировать

[17] Ссылки могут также ссылаться на произвольные адреса, но добиться этого несколько сложнее. Для вашей же безопасности мы не покажем, как этого добиться (к тому времени, как вы сможете сделать это самостоятельно, вы будете понимать, что стоит делать, а что — нет).

указатели в классах, насколько это возможно. Поскольку интеллектуальные указатели заботятся о выделении и освобождении памяти, им следует отдавать предпочтение над обычными указателями даже внутри классов. Сравнение указателей и ссылок приводится в табл. 1.9.

Таблица 1.9. Сравнение указателей и ссылок

Свойство	Указатели	Ссылки
Ссылка на определенное местоположение		√
Обязательная инициализация		√
Отсутствие утечек памяти		√
Обозначения, применяемые для объектов		√
Управление памятью	√	
Адресная арифметика	√	
Использование в контейнерах	√	

1.8.6. Не ссылайтесь на устаревшие данные!

Локальные переменные функций доступны только в области видимости функции, например

```
double& square_ref(double d) // НЕ ДЕЛАЙТЕ ТАК!!!
{
    double s = d*d;
    return s;
}
```

Здесь результат нашей функции ссылается на локальную переменную s, которой больше не существует. Память, где она хранилась до сих пор, никуда не исчезла, так что нам может повезти (ложное везение!) и она не будет перезаписана к тому моменту, когда мы ею воспользуемся. Но рассчитывать на это категорически нельзя. На самом деле такие скрытые ошибки даже хуже, чем очевидные, приводящие к аварийному завершению, — потому что они могут разрушить программу только при определенных условиях, и потому их очень трудно найти.

Такие ссылки называются *устаревшими ссылками*. Хороший компилятор предупредит вас, когда вы захотите получить ссылку на локальную переменную. К величайшему сожалению, такие примеры приходилось видеть даже в учебниках по программированию в Интернете!

То же самое справедливо и в отношении указателей:

```
double* square_ptr(double d) // НЕ ДЕЛАЙТЕ ЭТОГО!!!
{
    double s = d*d;
    return &s;
}
```

Этот указатель хранит локальный адрес, который вышел из области видимости. Такой указатель называется *висячим указателем*.

Возврат ссылок или указателей может быть корректен в случае функций-членов, когда они указывают на члены-данные (см. раздел 2.6).

Совет

Возвращайте указатели и ссылки только на данные в динамически выделенной памяти, на данные, существовавшие до вызова функции, или на статические данные.

1.8.7. Контейнеры в качестве массивов

В качестве альтернативы традиционным массивам C++ мы хотим представить вам два типа контейнеров, которые могут быть использованы аналогичным образом.

1.8.7.1. Стандартный вектор

Массивы и указатели являются частью языка C++. В отличие от них `std::vector` принадлежит стандартной библиотеке и реализован в виде шаблона класса. Тем не менее он может использоваться практически так же, как и массивы. Например, в примере из раздела 1.8.1 создание двух массивов, `v` и `w`, выглядит для векторов следующим образом:

```
#include <vector>

int main ()
{
    std::vector <float> v(3), w(3);
    v[0] = 1;  v[1] = 2;  v[2] = 3;
    w[0] = 7;  w[1] = 8;  w[2] = 9;
}
```

Размер вектора не обязан быть известен во время компиляции. Размеры векторов могут изменяться во время их жизни, как будет показано в разделе 4.1.3.1.

C++11 Поэлементная инициализация вектора не очень-то компактна. Поэтому C++11 допускает инициализацию с помощью списков в фигурных скобках:

```
std::vector<float> v = {1,2,3}, w = {7,8,9};
```

В этом случае размер вектора следует из длины списка инициализации. Сложение векторов, рассматривавшееся ранее, также может быть реализовано более надежно:

```
void vector_add(const vector<float>& v1, const vector<float>& v2,
                vector<float>& s)
{
    assert(v1.size() == v2.size());
    assert(v1.size() ==  s.size());
    for(unsigned i = 0; i < v1.size(); ++i)
        s[i] = v1[i] + v2[i];
}
```

В отличие от массивов и указателей C аргументы типа `vector` знают свои размеры, и теперь мы можем проверить, совпадают ли они. *Примечание*: размер массива можно вывести с помощью шаблонов; этот вопрос мы оставим в качестве упражнения (см. раздел 3.11.9).

Векторы могут копироваться и возвращаться из функций. Это позволяет нам использовать более естественную запись:

```
vector<float> add(const vector<float>& v1,
                  const vector<float>& v2)
{
    assert(v1.size() == v2.size());
    vector<float> s(v1.size());
    for(unsigned i = 0; i < v1.size(); ++i)
        s[i] = v1[i] + v2[i];
    return s;
}

int main ()
{
    std::vector<float> v = {1,2,3}, w = {7,8,9}, s = add(v,w);
}
```

Эта реализация потенциально дороже предыдущей, в которой целевой вектор передавался с помощью ссылки. Позже мы обсудим возможности оптимизации — осуществляемой как компилятором, так и пользователем. По нашему опыту более важно начинать работу с создания производительного интерфейса, а вопросами производительности озадачиваться несколько позже. Легче сделать правильную программу быстрой, чем сделать быструю программу правильной. Таким образом, первоначальная цель заключается в хорошем дизайне программы. Почти во всех случаях хороший интерфейс может быть реализован с достаточной производительностью.

Контейнер `std::vector` не является вектором в математическом смысле. В нем нет арифметических операций. Тем не менее этот контейнер оказывается весьма полезным в научных приложениях для обработки нескалярных промежуточных результатов.

1.8.7.2. `valarray`

`valarray` представляет собой одномерный массив с поэлементными операциями; даже умножение выполняется поэлементно. Операции со скалярным значением выполняются с каждым из элементов `valarray`. Таким образом, `valarray` чисел с плавающей точкой можно рассматривать как векторное пространство.

В следующем примере демонстрируются некоторые операции:

```
#include <iostream>
#include <valarray>
int main()
{
```

```
    std::valarray<float>
      v = {1, 2, 3},
      w = {7, 8, 9},
      s = v + 2.0f*w;
    v = sin(s);
    for(float x : v)
        std::cout << x << ' ';
    std::cout << '\n';
}
```

Обратите внимание, что `valarray<float>` может работать только с самим собой или с `float`. Так, запись `2*w` является ошибочной, поскольку перемножение `int` и `valarray<float>` не поддерживается.

Основная привлекательность `valarray` заключается в способности доступа к его срезам. Это позволяет *эмулировать* матрицы и тензоры высоких порядков, включая соответствующие их операции. Тем не менее из-за отсутствия непосредственной поддержки большинства операций линейной алгебры `valarray` используется в числовых приложениях не столь широко. Мы со своей стороны рекомендуем использовать для решения задач линейной алгебры авторитетные и проверенные библиотеки C++. Хочется верить, что в будущих стандартах язык программирования C++ будет включать такую библиотеку.

1.9. Структурирование программных проектов

Большой проблемой крупных проектов являются конфликты имен. По этой причине мы рассмотрим, как данная проблема усугубляется макросами. С другой стороны, позже, в разделе 3.2.1, мы покажем, как пространства имен помогают нам бороться с конфликтами имен.

Чтобы понять, как в программном проекте C++ взаимодействуют файлы, необходимо разобраться в процессе построения, т.е. в том, как выполнимый файл генерируется из исходных файлов. Это и будет предметом нашего первого подраздела. В этом свете мы представим механизм макросов и другие возможности языка.

Прежде всего мы хотим кратко обсудить возможность языка, которая способствует структурированию программы, — комментарии.

1.9.1. Комментарии

Очевидно, что основная цель комментария — описание на понятном языке того, что в исходном тексте программы не является очевидным для всех, например

```
// Трансмутация антибрахия за время O(n log n)
while(trans(mutation) < end_of(anti_brachius)) {
    ....
```

Часто комментарий представляет собой псевдокод, поясняющий запутанную реализацию:

```
// A = B * C
for(...) {
    int x78zy97 = yo6954fq, y89haf = q6843, ...
    for(...) {
        y89haf += ab6899(fa69f) + omygosh(fdab); ...
        for(...) {
            A(dyoa929,oa9978 ) += ...
```

В таком случае мы должны спросить себя, нельзя ли реструктуризировать наше программное обеспечение таким образом, чтобы такие непонятные реализации осуществлялись разово, в каком-нибудь темном углу библиотеки, а везде мы писали бы ясные и простые инструкции, например

```
A= B * C;
```

как программный, а не псевдокод. Это и есть одна из главных целей нашей книги — показать вам, как написать именно то выражение, которое вы хотите, в то время как его реализация "под капотом" выжимает из него максимальную производительность.

Еще одно частое использование комментариев — временно скрыть от компилятора код, который должен исчезнуть на время эксперимента с альтернативными реализациями, например

```
for(...) {
    // int x78zy97 = yo6954fq, y89haf = q6843, ...
    int x78zy98 = yo6953fq, y89haf = q6842, ...
    for(...) {
        ...
```

Как и C, C++ предоставляет возможность блочных комментариев, окруженных `/*` и `*/`. Они могут использоваться для превращения произвольной части кода или нескольких строк в комментарий. К сожалению, они не могут быть вложенными: независимо от того, сколько уровней комментариев открыты с помощью `/*`, первые встреченные символы `*/` завершают все комментарии блока. Почти все программисты сталкиваются с этой ловушкой. Они пытаются закомментировать бо́льшую часть кода, которая уже содержит блок комментариев, а в результате комментарий заканчивается раньше, чем планировалось, например

```
for(...) {
    /* int x78zy97 = yo6954fq; // Начало нового комментария
    int x78zy98 = yo6953fq;
    /* int x78zy99 = yo6952fq; // Начало старого комментария
    int x78zy9a = yo6951fq; */ // Конец старого комментария
    int x78zy9b = yo6950fq; */ // Конец нового комментария (увы, нет!)
    int x78zy9c = yo6949fq;
    for(...) {
```

Здесь строка с присваиванием `x78zy9b` должна быть закомментирована, но предшествующие символы `*/` преждевременно прерывают новый комментарий.

Вложенные комментарии можно корректно реализовать с помощью директивы препроцессора `#if`, как будет показано в разделе 1.9.2.4. Еще одной возможностью отключить несколько строк является использование соответствующей функции интегрированной среды разработки или редактора, предназначенного для работы с исходными текстами данного языка программирования.

1.9.2. Директивы препроцессора

В этом разделе мы представим команды (директивы), которые могут использоваться в предварительной обработке (препроцессинге) исходных текстов. Поскольку они практически не зависят от языка программирования, мы рекомендуем минимизировать их применение, в особенности это касается макросов.

1.9.2.1. Макросы

Почти любой макрос является демонстрацией недостатка языка программирования, программы или программиста.

— Бьярне Страуструп (Bjarne Stroustrup)

Это очень старый метод повторного использования кода — путем расширения имен макросов до текста их определения, потенциально с аргументами. Макросы дают много возможностей украсить свои программы, но еще больше возможностей их погубить. Макросы не связаны пространствами имен, областями видимости или любыми иными возможностями языка потому, что они являются простой заменой текста без какого-либо понятия о типах. К сожалению, некоторые библиотеки определяют макросы с такими распространенными именами, как, например, `major`. Мы должны бескомпромиссно отменить такие макросы, например, используя `#undef major`, без малейшей пощады по отношению к тем, у кого достанет ума их использовать. Visual Studio определяет — даже сегодня!!! — макросы `min` и `max`, и мы настоятельно рекомендуем вам отключить их при компиляции с помощью опции командной строки `/DNOMINMAX`. Почти все макросы могут быть заменены чем-то иным (константами, шаблонами, встроенными функциями). Но если вы действительно не в состоянии найти другой способ реализации чего-то, прислушайтесь к следующему совету.

Имена макросов

Используйте для макросов длинные и уродливые имена, состоящие из прописных букв, — наподобие `LONG_AND_UGLY_NAMES_IN_CAPITALS`!

Макросы могут создавать странные проблемы почти всеми мыслимыми и немыслимыми способами. Чтобы дать вам общее представление, мы рассмотрим

несколько примеров в разделе A.2.10, с некоторыми советами о том, как с ними бороться. Вы можете спокойно отложить чтение этого раздела до тех пор, пока не столкнетесь с некоторыми из этих проблем на практике.

Как вы узнаете из этой книги, C++ предоставляет куда лучшие альтернативы, такие как как константы, встраиваемые функции и `constexpr`.

1.9.2.2. Включение

Чтобы упростить язык C, многие возможности языка, такие как операции ввода-вывода, были исключены из ядра языка и вместо этого реализованы библиотекой. C++ следует этому дизайну и реализует новые возможности там, где это возможно, в стандартной библиотеке, но пока что еще никто не назвал C++ простым языком.

Как следствие почти каждая программа должна включать один или несколько заголовочных файлов. Наиболее часто включается заголовочный файл, отвечающий за ввод-вывод:

```
#include <iostream>
```

Препроцессор ищет этот файл в стандартных каталогах заголовочных файлов, таких как `/usr/include`, `/usr/local/include` и т.д. Мы можем добавить в этот путь поиска дополнительные каталоги с помощью флагов командной строки компилятора — обычно `-I` в мире Unix/Linux/Mac OS и `/I` в Windows.

Когда мы записываем имя файла в двойных кавычках, например

```
#include "herberts_math_functions.hpp"
```

компилятор обычно начинает поиск в текущем каталоге, а только затем — в стандартных путях[18]. Это эквивалентно использованию угловых скобок и добавлению текущего каталога в пути поиска. Некоторые программисты утверждают, что угловые скобки должны использоваться только для системных заголовочных файлов, а двойные кавычки — для пользовательских заголовочных файлов.

Чтобы избежать коллизий имен, зачастую к пути поиска добавляется родительский каталог, а в директиве используется относительный путь:

```
#include "herberts_includes/math_functions.hpp"
#include <another_project/more_functions.h>
```

Косая черта является переносимым символом разделения каталогов и работает и в Windows, несмотря на тот факт, что в этой операционной системе для вложенных каталогов используется символ обратной косой черты.

Защита включения. Часто используемые заголовочные файлы могут оказаться включенными несколько раз в одной единице трансляции из-за косвенного включения заголовочного файла. Чтобы избежать запрещенных повторений и ограничить расширение текста, используется так называемая *защита включений*,

[18] Какие именно каталоги будут просматриваться в поисках файла в двойных кавычках, зависит от реализации и в стандарте не оговаривается.

гарантирующая, что будет выполнено только первое включение заголовочного файла. Эта защита представляет собой простые макросы, которые указывают состояние включения определенного файла. Типичный включаемый файл выглядит следующим образом:

```
// Автор: я
// Лицензия: $100 только за каждое чтение этого файла

#ifndef HERBERTS_MATH_FUNCTIONS_INCLUDE
#define HERBERTS_MATH_FUNCTIONS_INCLUDE

#include <cmath>

double sine(double x);
...

#endif // HERBERTS_MATH_FUNCTIONS_INCLUDE
```

Таким образом, содержимое файла включается только тогда, когда защищающий макрос не определен. В тексте заголовочного файла мы определяем защищающий макрос для предотвращения повторных включений.

Как и в случае любых макросов, мы должны уделять повышенное внимание уникальности определяемого имени, причем не только в нашем проекте, но и во всех других заголовочных файлах, которые мы включаем прямо или косвенно. В идеале имя должно представлять проект и имя файла. Оно также может содержать относительный путь проекта или пространство имен (§3.2.1). Завершение имени защитного макроса суффиксом `_INCLUDE` или `_HEADER` — весьма распространенная практика. Случайное повторное использование имени защитного макроса может породить множество различных ошибок. Исходя из нашего опыта обнаружение причины этих неприятностей оказывается не самым простым делом. Опытные разработчики генерируют их автоматически с помощью упомянутых сведений или даже с помощью генераторов случайных чисел.

Удобной альтернативой является строка `#pragma once`. С ее использованием предыдущий пример сокращается до

```
// Автор: я
// Лицензия: $100 только за каждое чтение этого файла

#pragma once

#include <cmath>

double sine(double x);
...
```

Эта прагма не является частью стандарта, но все нынешние крупные компиляторы ее поддерживают. С помощью этой директивы ответственность за однократное включение заголовочного файла перекладывается на компилятор.

1.9.2.3. Условная компиляция

Важным и необходимым применением директив препроцессора является управление условной компиляцией. Препроцессор предоставляет директивы `#if`, `#else`, `#elif` и `#endif` для возможности ветвления. Условия могут быть сравнениями, проверкой определений имен макросов или логическими выражениями с ними. Директивы `#ifdef` и `#ifndef` являются сокращениями соответственно для:

```
#if defined  ( MACRO_NAME )
#if !defined ( MACRO_NAME )
```

Длинная форма должна использоваться при проверке определения в сочетании с другими условиями. Аналогично `#elif` является сокращением для `#else` и `#if`.

В идеальном мире мы могли бы писать только переносимые, соответствующие стандарту C++ программы. В действительности иногда мы вынуждены использовать непереносимые библиотеки. Скажем, у нас есть библиотека, доступная только в Windows, точнее — только при использовании Visual Studio. Для всех других компиляторов у нас есть альтернативная библиотека. Самый простой способ реализации, зависящей от платформы, — предоставить альтернативные фрагменты кода для различных компиляторов:

```
# ifdef _MSC_VER
... Код для Windows
# else
... Код для Linux/Unix
# endif
```

Аналогично нам нужна условная компиляция, когда мы хотим использовать новую возможность языка, которая доступна не на всех целевых платформах, скажем, семантику перемещения (§2.3.5):

```
# ifdef MY_LIBRARY_WITH_MOVE_SEMANTICS
... Что-то эффективное с использованием семантики перемещения
# else
... Менее эффективное, но более переносимое решение
# endif
```

Так мы можем использовать некоторую возможность при ее наличии и при этом по-прежнему поддерживать переносимость для компиляторов без этой возможности. Конечно, нам нужны надежные инструменты, которые определяют макрос только тогда, когда эта возможность действительно доступна. Условная компиляция является довольно мощным средством, но она имеет свою цену: поддержка исходного текста и тестирование становятся более трудоемкими и подверженными ошибкам. Эти недостатки могут быть уменьшены с помощью хорошо продуманной инкапсуляции, так что различные реализации используются более общими интерфейсами.

1.9.2.4. Вкладываемые комментарии

Директива `#if` может использоваться для того, чтобы закомментировать целые блоки кода:

```
#if 0
... Это код с ошибками. Мы исправим его. Потом. Может быть. Честно!
#endif
```

Преимущество такого подхода перед применением `/* ... */` состоит в том, что обеспечивается возможность вложенности таких закомментированных блоков:

```
#if 0
... Начало просто бессмыслицы.
#if 0
... Бессмыслица в квадрате внутри просто бессмыслицы.
#endif
... Окончание просто бессмыслицы.
... (К счастью, игнорируемое компилятором.)
#endif
```

Тем не менее следует умеренно использовать этот метод. Если три четверти программы представляют собой комментарии, ее следует серьезно пересмотреть.

Изложив материал этой главы конспективно, мы иллюстрируем основные возможности C++ в разделе A.3 приложения. Мы не включили его в основное содержание книги, чтобы поддержать высокий темп для нетерпеливых читателей. Тем, кто не любит такой спешки, мы рекомендуем найти время, чтобы прочитать упомянутый раздел и увидеть, какой путь в реальности проходят постепенно развивающиеся нетривиальные программы.

1.10. Упражнения

1.10.1. Возраст

Напишите программу, которая запрашивает входные данные с клавиатуры и получает их, а также выводит результат на экран и записывает его в файл. Вопрос, который задает программа, — “Сколько вам лет?”

1.10.2. Массивы и указатели

1. Напишите следующие объявления: указатель на символ, массив из 10 целых чисел, указатель на массив из 10 целых чисел, указатель на массив символьных строк, указатель на указатель на символ, целочисленная константа, указатель на целочисленную константу, константный указатель на целое число. Инициализируйте все эти объекты.

2. Напишите небольшую программу, которая создает массивы в стеке (массивы фиксированного размера) и массивы в куче (с помощью распределения памяти). Воспользуйтесь `valgrind`, чтобы посмотреть, что происходит, когда вы не удаляете их корректно.

1.10.3. Чтение заголовка файла Matrix Market

Формат данных Matrix Market используется для хранения плотных и разреженных матриц в формате ASCII. Заголовок содержит некоторую информацию о типе и размер матрицы. Для разреженных матриц данные хранятся в трех столбцах. Первый столбец содержит номер строки, второй — номер столбца, а третий — числовое значение на их пересечении. В случае матрицы с комплексными значениями добавляется четвертый столбец для мнимой части.

Вот пример файла Matrix Market:

```
%%MatrixMarket Действительная матрица с координатами
%
% Матрица рассчитанных значений
%
        2025         2025           100015
           1            1           .9273558001498543E-01
           1            2           .3545880644900583E-01
.....................
```

Первая строка, которая начинается не с символа `%`, содержит количество строк, количество столбцов и количество ненулевых элементов разреженной матрицы.

Используйте `fstream` для чтения заголовка файла Matrix Market и вывода на экран размера матрицы и количества ее ненулевых элементов.