Основы языка C++

брошюра в стадии написания

Андрей Строганов

13 сентября 2016 г.

Оглавление

I Основы **2**

1 Первая программа **5**
1.1 Что такое компилятор . . . 5
1.2 Самая маленькая программа . . . 6
1.3 Комментарии . . . 6
1.4 Вывод на экран . . . 6
1.5 Переход на новую строку . . . 7
1.6 Арифметические выражения . . . 8

2 Типы данных, переменные, константы **10**
2.1 Переменные . . . 10
2.2 Целочисленные типы данных . . . 11
2.3 Типы данных с плавающей точкой . . . 12
2.4 Объявление, инициализация, присваивание . . . 13
2.5 Сокращенная запись некоторых выражений . . . 13
2.6 Константы . . . 14

3 Инструкции ветвления **15**
3.1 Инструкция `if` . . . 15
3.2 Операторы сравнения . . . 16
3.3 Инструкция `if-else` . . . 16
3.4 Вложенные условия . . . 17
3.5 Логическое *И* и *ИЛИ* . . . 18
3.6 Инструкция switch . . . 19
3.7 Логические выражения и тип `bool` . . . 20

4 Инструкции циклов **22**
4.1 Инструкция while . . . 22
4.2 Инструкция do-while . . . 23
4.3 Инструкция `for` . . . 23
4.4 Оператор `break` . . . 24
4.5 Оператор `continue` . . . 25
4.6 Пошаговая прокрутка циклов . . . 26

5 Функции **28**
5.1 Определение функции . . . 29
5.2 Возвращаемое значение . . . 31
5.3 Объявление и определение функций . . . 32
5.4 Перегрузка функций . . . 33
5.5 Параметры по умолчанию . . . 33
5.6 Некоторые полезные функции . . . 33

6 Одномерные массивы **34**
6.1 Класс `std::vector` . . . 35

7 Введение в строки **38**

8 Работа с текстовыми файлами **41**
8.1 Запись файлов . . . 41
8.2 Чтение файлов . . . 43

II Углублённый курс **45**

9 Ссылки и указатели **46**
9.1 Ссылки . . . 46
9.2 Указатели . . . 46
9.2.1 Операторы `new` и `delete` . . . 48
9.3 Динамическое создание массивов . . . 49

10 Исключения **50**
10.1 Пример без исключений . . . 50
10.2 Добавим исключения . . . 52
10.3 Стандартные исключения . . . 54

11 Структуры **58**

Часть I

Основы

Введение

Существует множество языков программирования для решения самых разных задач. Эта брошюра посвящена одному из самых мощных и распространённых — C++. Несомненно, у любого языка наряду с достоинствами, есть и недостатки. Сейчас мы их не будем обсуждать, а лишь коротко представим язык, который читатель предполагает изучить.

Еще до появления C++ был язык C, который до сих пор развивается, и является одним из мощнейших инструментов профессиональных программистов. C++ унаследовал синтаксис C и уже много лет развивается параллельно. Некоторые люди ошибочно полагают, что прежде чем учить C++, нужно освоить C, и что C++ — это C с некоторыми дополнительными возможностями. Оба утверждения не совсем верны. Несмотря на схожесть синтаксиса, базовые средства в C и C++ достаточно сильно различаются, поэтому разумнее начинать изучать тот язык программирования, который вы и хотите освоить. Еще одно распространенное заблуждение заключается в том, что изучать программирование нужно с языка «полегче», например с Basic или Pascal. С этим можно было бы согласиться, если бы Basic и Pascal были действительно проще C++, но это не так. Небольшие учебные программы на C++ столь же просты и понятны, как и их аналоги на языках типа Pascal. Тем более, после изучения Basic или Pascal всё равно придётся переучиваться, так как промышленно важные программы на этих языках практически не разрабатываются.

Большим достоинством C++ является то, что одну и ту же задачу в нем можно решить не просто разными способами, а в корне различающимися подходами в программировании. Но заметим, что несмотря ни на что, C++ и C имеют много общего, поэтому программист C++ может без труда использовать и C.

В завершение сказанного, приведем несколько примеров известных программных продуктов, написанных на языках C и C++. Заметим, что большие программы часто включают в себя модули, написанные на разных языках, например ядро и вся механика игры может быть написана на одном языке, а искусственный интеллект персонажей — на другом.

C++: Microsoft Office, Mozilla Firefox, Google Chrome, Opera, Adobe Photoshop, Warcraft, Starcraft, Assasin's Creed, The Elders Scrolls (Morrowind, Oblivion, Skyrim),

Fallout, World of warcraft, Lineage, Dota2.

C: Microsoft Windows, Linux, FreeBSD, MacOS, многие программы.

Данное учебное пособие создано на основе курса C++, читаемого автором в центре образования 1434.

Глава 1

Первая программа

В этой главе мы коротко рассмотрим чем отличается исходный код программы от бинарного кода, как устроена программа на C++ и как выводить текст на экран.

1.1 Что такое компилятор

Бендер: Мне приснился кошмар. Повсюду нули и единицы... и, кажется, я видел двойку.
Фрай: Это был просто сон. Двоек не бывает.
(Футурама)

Язык программирования — это всего лишь набор правил, согласно которым описывается последовательность действий. Чтобы создать программу нужно написать ее согласно этим правилам, а потом написанный текст перевести на «компьютерный язык» с помощью специальной программы, которая называется *компилятор*. В результате компиляции получается компьютерная программа (например: текстовый редактор, игра или вирус), которую можно запустить или передать другим пользователям.

Покажем результат компиляции одного из примеров ниже. Вместо понятного текста на языке программирования, получается следующая последовательность единиц и нулей:

```
01111111010001010100110001000110000000010000000 1
00000001000010010000000000000000000000000000000 0
00000000000000000000000000000000000000001000000 0
00000011000000000000001000000000000000000000000 0
10000000100001010000010000001000001101000000000 0
00000000000000000011001000000110000000000000000 0
00000000000000000000000000000000011010000000...
```

Всего здесь 34016 символов. Откомпилированная программа стала менее понятной нам, но более понятной компьютеру.

1.2 Самая маленькая программа

Одним из основных понятий в языке программирования является *функция*. Функцией мы будем называть подпрограмму, имеющую имя. Любая программа на языке C++ должна иметь функцию с именем *main*, а результатом работы программы должно быть целое число (на это указывает тип `int`). Приведем пример программы, имеющей функцию main типа `int` (помимо `int` существуют другие типы данных, о них речь пойдет ниже):

```
int main()
{
}
```

В круглых скобках указываются параметры функции, но в данном случае `main` не принимает параметров. Между фигурными скобками помещается тело функции, то есть набор действий, которые выполняет программа. Текст программы также называется *исходным кодом*, *исходником* или *сорцами* (от английского *source code*, *source*, *sources*).

1.3 Комментарии

Часто для большей ясности в исходный код программы добавляются *комментарии*, которые начинаются сразу после двух слешей ("//"), либо заключаются в блок "/* */":

```
int main() //начало моей программы
{
  /*
  Это -- пустая функция,
  она ничего не делает.
  */
}
```

Комментарии игнорируются компилятором и нужны исключительно для сопровождения исходных кодов.

1.4 Вывод на экран

Язык программирования предоставляет лишь простейшие (примитивные) средства для построения программ. Даже выввод на экран смайлика — вообще

говоря, довольно сложная задача, которая требует нескольких строк кода. Поэтому часто громоздкий код помещается в отдельные файлы, которые мы можем подключить (`#include`) к программе, и, не вдаваясь в детали, использовать уже готовые решения.

Для вывода текста на экран используется объект `cout`, который объявлен в библиотеке *iostream*, в пространстве имен `std`. В качестве примера, выведем на экран строку «Трям! =)»:

```
1 #include <iostream>
2 int main()
3 {
4   std::cout << "Трям! =)";
5 }
```

В строке 1 с помощью директивы `#include` мы включили файл `iostream` в программу, чтобы компилятор знал, что существует объект `cout` и как он работает. В строке 4 строка "Трям =)" записывается (<<) в объект `cout`, определенный в пространстве имен `std`.

Чтобы явно не указывать пространство имен `std` (а его нужно указывать для всех типов и объектов из стандартной библиотеки: `cout`, `cin`, `endl`, `vector`, `map`, `fstream`, `ifstream`, `ofstream` и так далее), можно в начало программы добавить строку `using namespace std;` Пример:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Трям! =)";
}
```

Не забывайте, что программа начинает выполняться с функции *main*. Все, что написано до нее нужно чтобы компилятор правильно понимал незнакомые слова.

1.5 Переход на новую строку

Рассмотрим пример:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Кто ходит в гости по утрам,";
  cout << "Тот поступает мудро!";
}
```

На экране появится надпись:

```
Кто ходит в гости по утрам,Тот поступает мудро!
```

Чтобы строки не склеивались, необходимо после каждой строки выводить символ-разделитель, который в C++ обозначается *endl*. Следующий пример выведет текст:

```
По синим волнам океана,
Лишь звезды блеснут в небесах,
Корабль одинокий несется,
Несется на всех парусах.
(М. Лермонтов)
```

Пример:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "По синим волнам океана," << endl;
  cout << "Лишь звезды блеснут в небесах," << endl;
  cout << "Корабль одинокий несется," << endl;
  cout << "Несется на всех парусах." << endl;
  cout << "(М. Лермонтов)" << endl;
}
```

Обратите внимание на то, что символы << обозначают *запись* строки в объект cout, а не являются литературным ковычками « ».

Иногда при изложении материала мы не будем приводить полные исходные коды программ, а лишь ту часть программы, которую мы обсуждаем. При этом подразумевается, что в программе подключены (#include) соответствующие библиотеки, написано using namespace std; и объявлена функция int main().

1.6 Арифметические выражения

При выводе арифметического выражения на экран без кавычек будет выведен результат вычисления:

```
cout << 3*(2+5) << endl;
```

На экране появится: 21. Можно строить и более сложные выражения:

```
cout << "В каждую из 8 корзинок положим по 5 яблок." << endl;
cout << "Всего получится " << 8 * 5 << " яблок." << endl;
```

При запуске программы компьютер выведет:

```
В каждую из 8 корзинок положим по 5 яблок.
Всего получится 40 яблок.
```

Для арифметических операций используются следующие обозначения:

- `+` сложение,
- `-` вычитание,
- `*` умножение,
- `/` деление,
- `%` остаток от деления,
- `()` скобки.

Как известно, деление на ноль неопределено. Если в программе происходит деление на ноль, то она сразу закрывается с ошибкой.

Обратите внимание, что в C++ **не существует** операции возведения в степень. Для этого используется функция `pow`, о которой мы расскажем позже.

Глава 2

Типы данных, переменные, константы

В этой главе мы узнаем об основных типах данных, научимся считывать данные с клавиатуры и поговорим об арифметических операциях в C++.

2.1 Переменные

Для того, чтобы пользователь нашей программы мог с ней взаимодействовать, прежде всего, нужно научить программу спрашивать какие-нибудь данные и их запоминать. Например, мы хотим написать программу, которая будет вычислять периметр квадрата. Для этого нужно знать всего одно число — длину стороны.

Объявим в программе целочисленную (`int`) переменную `storona`, в которую, с помощью команды `cin` пользователь введёт целое число (длину стороны).

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите длину стороны: ";
  int storona; // объявляем переменную storona
  cin >> storona; // считываем с клавиатуры целое число
  cout << "Периметр квадрата: " << 4*storona << endl;
}
```

Переменную можно было назвать совершенно произвольно, например, не `storona`, а `dlina`, или просто буквой `a`. Имена переменных могут содержать большие и маленькие латинские буквы, цифры и знак подчеркивания, но

первой в имени переменной не может быть цифра. Примеры правильных имен переменных: `dlina`, `age`, `user_name`, `d2_`. Примеры неправильных имен: `dlina puti` (пробелы вставлять нельзя), `2v` (имя не может начинаться с цифры).

Приведем еще один пример программы. Найдем периметр прямоугольника по двум сторонам.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите длины сторон прямоугольника: ";
  int a, b;
  cin >> a >> b;
  int p = 2*(a+b);
  cout << "Периметр прямоугольника: " << p << endl;
}
```

В этом примере демонстрируется, что можно объявить сразу несколько переменных:

```
int a, b;
```

а затем считать их:

```
cin >> a >> b;
```

Можно было обойтись без переменной `p`, подставив выражение `2*(a+b)` сразу в `cout`:

```
  cin >> a >> b;
  cout << "Периметр прямоугольника: " << 2*(a+b) << endl;
```

Мы уже познакомились с одним целочисленным типом данных — `int`. Далее мы рассмотрим остальные числовые типы. Про последний, логический, тип `bool` речь пойдет в следующей главе.

2.2 Целочисленные типы данных

Тип данных определяет множество допустимых значений данных и возможные операции над ними.

К целочисленным типам данных относятся: `char`, `short` (или `short int`), `int`, `long` (или `long int`). Они используются для хранения целых чисел, например: 70, -12942, 15. Различаются эти типы только размером. Например, переменная типа `char` всегда занимает один байт в памяти компьютера. Мы знаем, что в одном байте 8 бит, а каждый бит может принимать значения 0 или 1. Таким образом, можно составить 2^8 различных последовательностей из 8 бит.

Поэтому, переменная типа `char` может хранить $2^8 = 256$ различных чисел, это числа от -128 до 127.

Переменная типа `short int` занимает уже 2 байта памяти и поэтому может хранить числа от -32768 до 32767.

Размер переменной целочисленного типа на разных компьютерах может меняться. Например, когда-то переменная типа `int` занимала 2 байта и принимала значения как переменная типа `short int`, а на современных компьютерах `int` уже занимает 4 байта и позволяет хранить числа от $-2\,147\,483\,648$ до $2\,147\,483\,647$.

С помощью модификатора `unsigned` целочисленный тип можно беззнаковым, тем самым увеличив максимально возможное значение за счет отрицательной области. Так, переменная типа `unsigned char` будет принимать значения от 0 до 255 включительно.

Предполагая размеры типов `char`, `short`, `int`, `long` равными 1, 2, 4, 8 байт соответственно, приведем максимальные и минимальные значения для них:

Тип	**Min**	**Max**
`char`	-128	127
`unsigned char`	0	255
`short`	-32768	32767
`unsigned short`	0	65535
`int`	-2^{31}	$2^{31}-1$
`unsigned int`	0	$2^{32}-1$
`long`	-2^{63}	$2^{63}-1$
`unsigned long`	0	$2^{64}-1$

Чтобы определить размер типа можно воспользоваться оператором `sizeof`:

```
cout << "Размер переменной типа int: " << sizeof(int) << endl;
```

Гарантируется, что независимо от архитектуры всегда будет выполняться соотношение:

$$1 \equiv \texttt{sizeof(char)} \le \texttt{sizeof(short)} \le \texttt{sizeof(int)} \le \texttt{sizeof(long)}$$

2.3 Типы данных с плавающей точкой

Типы данных с плавающей точкой (floating point type) `float`, `double` и `long double` используются для хранения дробных чисел, например: 50.2109, -132.501.

Эти типы отличаются не только диапазоном значений, но и количеством цифр после запятой. Обычно, переменные типа `float` могут хранить до 6 цифр после запятой, а `double` — до 10. В зависимости от архитектуры и операционной системы размеры типов могут варьироваться.

В арифметике целых чисел дробь всегда отбрасывается. Поэтому, например, `7/2 = 3`, или `7/8 = 0`. Если хотя бы один из операндов — дробное число, то и результат будет дробным. Поэтому, чтобы результат деления 7 на 2 был точным, достаточно либо конвертировать один из операндов в тип данных с плавающей точкой: `double(7)/2`, либо «подсказать» компилятору, что у 7 есть дробная часть, но она равна нулю: `7.0/2`. Иногда можно встретить запись вида `2*.5`, что означает нулевую целую часть: `2*0.5`.

2.4 Объявление, инициализация, присваивание

Чтобы объявить переменную, достаточно указать ее тип:

```
int age; //Объявили целочисленную переменную с именем age
```

Для записи значений в переменные используется оператор присваивания «=»

```
age = 15; //Присвоили значение 15 в переменную age
```

Не путайте оператор присваивания «=» с математичеким символом, обозначающим равенство. В программировании `a=b` обозначает не *приравнять* переменные, а присвоить (то есть *записать*) в переменную `a` то, что находится в переменной `b`. Слева от оператора присваивания должен находиться объект, в который можно что-то присвоить, поэтому, например, запись `5=a` является некорректной и вызовет ошибку компиляции.

Значение можно присвоить сразу при объявлении переменной:

```
int age = 15;
```

Это называется *инициализация переменной age при объявлении.* Всегда инициализировать переменные (то есть присваивать им начальные значения при объявлении) считается хорошим тоном в программировании, это уменьшает количество ошибок в программе.

2.5 Сокращенная запись некоторых выражений

Часто оказываются полезными операторы инкремента «++» и декремента «−−»:

```
++a; // эквивалентно a = a + 1
a++; // эквивалентно a = a + 1
--a; // эквивалентно a = a - 1
a--; // эквивалентно a = a - 1
```

Между префиксным оператором инкремента (++a) и постфиксным (a++) есть небольшая разница, о которой мы поговорим позже, но результат операции один и тот-же: значение переменной a увеличивается на единицу.

Типичное выражение a=a@b, где @ – некоторый арифметический оператор, может быть записано более выразительно: a@=b. Примеры:

```
a += 7; // эквивалентно a = a + 7
a /= 3; // эквивалентно a = a / 3
```

2.6 Константы

Для хранения промежуточных значений или улучшения ясности кода применяются константы, то есть именованные величины, которые нельзя изменить:

```
const double pi = 3.14159265;
```

Дальше в программе можно будет использовать pi вместо 3.14159265, что заметно улучшит читаемость и упростит сопровождение программы. Любая попытка присвоить в pi какое либо значение повлечет за собой ошибку компиляции, тем самым предотвратив неправильную работу программы. Старайтесь всегда объявляйте константными величины, которые не предполагается изменять.

Глава 3

Инструкции ветвления

В любой программе требуется выполнять те или иные действия, в зависимости от обстоятельств. Допустим, что мы пишем игру. Тогда если пользователь выбра заклинание *молния*, то мы должны нарисовать молнию и нанести урон войскам, а если выбрано лечение, то необходимо восстановить часть здоровья. Такое поведение программы, когда приходится рассматривать различные варианты называется *ветвлением.*

3.1 Инструкция `if`

Инструкция `if` позволяет выполнить блок кода (то есть несколько строк кода программы, заключенные в фигурные скобки) если верно некоторое условие (логическое выражение):

```
if( /*условие*/ )
{
  // некоторые действия
}
```

Напишем программу, определяющую модуль введённого числа. Напомним, что модуль — это абсолютная величина числа:

$$|x| = \begin{cases} x, & \text{если } x \geqslant 0, \\ -x, & \text{если } x < 0. \end{cases}$$

Например, $|5| = 5, \quad |-3| = -(-3) = 3.$

Программа получает на вход некоторое целое число и должна вывести его с положительным знаком. Очевидно, если введено положительное число, то его менять не нужно, а если число отрицательное, то нужно его взять с обратным знаком.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите число: ";
  int a;
  cin >> a;

  if(a < 0)
  {
    a = -a;
  }

  cout << "Ответ: " << a << endl;
}
```

Так как в `if` выполняется только одна строка (`a = -a;`), можно не выделять эту строку фигурными скобками, поэтому `if` можно записать ещё короче:

```
if(a < 0) a = -a;
```

3.2 Операторы сравнения

Для сравнения величин используются следующие операторы:

`==`	равно,
`!=`	не равно,
`<`	меньше,
`<=`	меньше, либо равно,
`>`	больше,
`>=`	больше, либо равно.

Не путайте оператор сравнения (`==`) с оператором присваивания (`=`). При сравнении данные не меняются, а при присваивании в одну переменную записывается значение другой.

3.3 Инструкция `if-else`

В предыдущем примере требовалось выполнить действите `a = -a;` при определённом условии `a < 0`. Часто бывает необходимо в противном случае выполнить другое действие. Для этого в паре с `if` используется блок `else`:

```
if( /*условие*/ )
{
  // блок кода, выполняющийся, если условие верно
}
```

```
else
{
  // блок кода, выполняющийся, если условие неверно
}
```

Напишем программу, определяющую чётность введённого числа. Для проверки на чётность будем сравнивать с нулём остаток от деления числа на 2.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите число: ";
  int a;
  cin >> a;

  if(a % 2 == 0)
    cout << "Число чётно" << endl;
  else
    cout << "Число нечётно" << endl;
}
```

Блок `else` всегда используется в связке с `if` и обозначает действия, которые нужно выполнить, если `if` не выполнился.

3.4 Вложенные условия

В некоторых задачах приходится проверять одно условие в другом. Напишем программу, определяющую количество решений уравнения $ax + b = 0$, где a и b вводит пользователь. Если $a \neq 0$, то решение одно: $x = -\frac{b}{a}$. Если $a = 0$, то возможны два случая: если $b = 0$, то решений бесконечно много, а если $b \neq 0$, то решений нет вообще.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите коэффициенты a и b: ";
  int a, b;
  cin >> a >> b;

  if(a == 0)
  {
    if(b == 0)
```

```
      cout << "Корней бесконечно много" << endl;
    else
      cout << "Корней нет" << endl;
  }
  else
    cout << "Один корень" << endl;
}
```

3.5 Логическое *И* и *ИЛИ*

Допустим, что вводятся два числа и требуется определить, являются ли они оба положительными. Чтобы проверить, верны ли одновременно два или более выражений, используем оператор *И* (логическое умножение), который обозначается двумя амперсандами &&:

```
if(a > 0 && b > 0) // если a > 0 И b > 0
```

Если хотя бы одно из чисел меньше либо равно 0, то `if` не выполнится. Пусть теперь требуется определить, что хотя бы одно из двух чисел положительно. Для этого воспользуемся оператором *ИЛИ* (логическое сложение), корорый обозначается двумя вертикальными палочками ||:

```
if(a > 0 || b > 0) // если a > 0 или b > 0
```

Используя операторы *И* и *ИЛИ* можно создавать достаточно сложные выражения. Например, проверим, что оба числа положительные, причем $a < b$ и одно из них равно 5.

```
if( (a > 0 && b > 0) && (a < b) && (a == 5 || b == 5) )
```

Приведем пример еще одного решения предыдущей задачи (о количестве корней уравнения), без вложенных условий.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Введите коэффициенты a и b: ";
  int a, b;
  cin >> a >> b;

  if(a == 0 && b == 0)
    cout << "Корней бесконечно много" << endl;
  else if(a == 0 && b != 0)
    cout << "Корней нет" << endl;
  else
```

```
    cout << "Один корень" << endl;
}
```

Обратите внимание, что двойные неравенства вида $0 < x < 5$ в C++ имеют другой смысл и результат будет весьма неожиданным, поэтому в таких случаях используют оператор *И*: `0 < x && x < 5`.

3.6 Инструкция switch

В случаях, когда необходимо сравнить значение целочисленной переменной со списком констант удобно применять инструкцию switch:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Выберите рисунок: " << endl;
  cout << "1) Роза" << endl;
  cout << "2) Смайлик" << endl;
  cout << "3) Стрела" << endl;

  cout << "Ваш выбор: ";
  int choice;
  cin >> choice;

  switch(choice) // проверяем значения переменной choice
  {
    case 1: // если choice равна 1
    {
      cout << "@}-’-,---" << endl;
      break;
    }
    case 2: // если choice равна 2
    {
      cout << "^_^" << endl;
      break;
    }
    case 3: // если choice равна 3
    {
      cout << ">---->" << endl;
      break;
    }
    default:
      //choice не равен 1 2 или 3
```

```
      cout << "Такого пункта в меню нет." << endl;
  } //switch
} //main
```

Ключевое слово `break` означает выйти из `switch`. Без `break` получилось бы так, что если пользователь введет 2, то на экран выведутся смайлик и стрела. То есть, `switch` выполнится с первого подходящего `case` до конца.

3.7 Логические выражения и тип `bool`

Рассмотрим произвольное арифметическое выражение: $2+3\cdot(7-5)$. Оно содержит различные арифметические операции: сложение, вычитание, умножение. Результатом этого выражения является число 8. Похожим образом можно вычислять логические выражения, то есть некоторые утверждения, которые могут быть истинны или ложны. Результатом вычисления логического выражения является также истина или ложь.

Логическим (булевым) типом в C++ является тип `bool`, переменные данного типа могут принимать значения `true` или `false`, которые означают, что некоторое логическое выражение истинно или ложно соответственно.

Значения типа `bool` можно *логически* складывать и умножать с помощью операторов && и ||. Результатом логического умножения (&&) является `true` только если оба операнда равны `true`, в противном случае результатом будет `false`.

Результатом логического сложения (||) является `true` если хотя бы один операнд равен `true`, если оба операнда равны `false`, то и результат `false`.

Например:

```
bool b;
b = (7 < 3) || (5 < 6); // b = true, т.к. 5 < 6 = true
b = (0 == 6) || ( 6 < 3); // b = false, т.к. оба выражения ложны
```

Существует оператор отрицания, обозначающийся восклицательным знаком `!`. Он действует лишь на один операнд меняя его значение на противоположенное. Например:

```
bool b = true;
bool u = !b; // u = false
```

Продемонстрируем работу логических операторов на следующих примерах.

```
int x = 2, y = 5;
bool b;
b = (y == 5); // true
b = (x > y);  // false
b = (x != 2); // false
```

```
b = (x > 1 && y > 0); // true
b = !(y == x || y < x); //true
```

Все числовые значения могут быть приведены к `bool` по правилу: «все, что не ноль = `true`, ноль = `false`»:

```
bool b;
int a = 4;
b = a - 7; // true
b = a - 4; // false
```

Булевые переменные нам еще не раз пригодятся, поэтому оставим этот раздел без примера. Пока мы не обсудили циклы, думается, что любой пример использования `bool` окажется слишком искуственным.

Глава 4

Инструкции циклов

С помощью инструкций циклов часть алгоритма можно выполнить несколько раз, пока верно некоторое условие. Один проход по телу цикла называется *итерацией*. Если проверка условия выполняется перед итерацией, говорят, что данная инструкция цикла с *предусловием*. Если проверка после итерации, то с *постусловием*.

4.1 Инструкция while

В общем случае инструкция цикла `while` выглядит так:

```
while( /*условие*/ )
{
  //Тело цикла
}
```

Напишем программу, которая выводит на экран все целые числа от 0 до 50 включительно. Для этого объявим переменную `i`, в которую присвоим начальное значение — число 0. В *теле цикла* будем выводить на экран значение переменной `i`, а затем увеличивать её на единицу (`++i`). Так, на первой итерации `i` будет равно 0, потом 1 и так далее. *Условием* выполнения цикла будет `i <= 50`. Как только `i` станет равной 51, цикл завершится:

```
int i = 0;
while(i <= 50)
{
  cout << i << endl;
  ++i;
}
```

Выражение `i <= 50` вычисляется перед каждой итерацией и, если оно равно `true` (то есть `i` действительно меньше или равно 50), выполняется очередная итерация цикла. Когда `i <= 50` станет равным `false` программа выйдет из цикла.

4.2 Инструкция do-while

Инструкция `do-while` аналогичная `while`, но условие проверяется после итерации:

```
do
{
  // Тело цикла
}
while( /*условие*/ );
```

Циклы с постусловием гарантированно выполняют хотя бы одну итерацию. Допустим, что требуется считать число от 10 до 50 и гарантировать что пользователь не введет неправильное значение:

```
do
{
  cout << "Введите число от 10 до 50: ";
  int a;
  cin >> a;
}
while(a < 10 || a > 50);
```

Цикл выполняется пока `a < 10` или `a > 50`.

4.3 Инструкция `for`

Несколько расширенным вариантом `while` является инструкция `for`:

```
for(инициализация; условие; действие)
{
    тело цикла
}
```

Инициализация — это действие, выполняемое один раз перед циклом. Обычно при инициализации объявляются переменные и устанавливаются их начальные значения.

Условие — некоторое логическое выражение. Цикл продолжается пока условие верно.

Действие — обычно здесь выполняются операции, необходимые для обеспечения правильной работы цикла. Например, если требуется, чтобы некоторая переменная `i` принимала значения $\{0, 1, 2, ...\}$, то действие будет `i = i + 1` или `++i`.

Уже знакомый нам пример с выводом чисел от 0 до 50 будет выглядеть так:

```
for(int i = 0; i <= 50; ++i)
  cout << i << " ";
```

Перед входом в цикл мы объявляем и инициализируем переменную `i`, а в конце каждой итерации увеличиваем ее на 1 (`++i`). Перед началом итерации проверяется условие `i <= n`. Если оно окажется неверным, цикл завершится.

4.4 Оператор `break`

Оператор `break` прерывает выполнение цикла. Напишем программу, которая проверяет, является ли число `n` простым. Для этого достаточно проверить, что оно не имеет делителей среди чисел от 2 до $\sqrt{n}$ (докажите это). Введём переменную `bool prime` и присвоим в неё значение `true`, тем самым предполагая, что число простое. Затем переберем все целые числа от 2 до $\sqrt{n}$, и если хотя бы одно число является делителем n, то присвоим `false` и прервём цикл.

```
bool prime = true;
for(int i = 2; i*i <= n; ++i)
  if(n % i == 0) // если n делится на i
  {
    prime = false; // число - составное
    break; // выходим из цикла
  }

if(prime) cout << "Число простое" << endl;
else cout << "Число составное" << endl;
```

В данном случае `break` прерывает цикл как только мы нашли один делитель, ведь в таком случае другие делители искать уже не нужно.

Эту задачу можно было бы решить еще элегантнее без `break`, проверяя значение `prime` в условии цикла:

```
bool prime = true;
for(int i = 2; prime && i*i <= n; ++i)
  if(n % i == 0)
    prime = false;
```

Можно еще немного упростить решение, заменив `if` на прямое вычисление логического выражения:

```
bool prime = true;
for(int i = 2; prime && i*i <= n; ++i)
  prime = n % i;
```

Здесь мы воспользовались правилом: «все, что не ноль, то `true`».

На этом примере можно было бы остановиться, так как проще и понятнее его сделать уже нельзя, но мы приведем еще одну реализацию, демонстрирующую то, что результат оператора присваивания является значением.

```
bool prime = true;
for(int i = 2; (prime = n % i) && i*i <= n; ++i);
```

На каждой итерации в `prime` присваивается `true`, если `n` не делится на `i` и сразу проверяется, нужно ли продолжать цикл. Очевидно, если `n` разделилось на `i` без остатка, то в `prime` присвоится `false` и цикл прервется. Обратите внимание на точку с запятой сразу после `for`. Это означает, что тело цикла пустое.

4.5 Оператор `continue`

Оператор `continue` используется когда нужно перейти на следующую итерацию цикла, не выполняя до конца текущую. Пусть требуется считать с клавиатуры 10 чисел и вычислить сумму цифр всех чётных.

Сначала приведём решение без использования `continue`:

```
int s = 0;
for(int i = 1; i <= 10; ++i)
{
  int x;
  cin >> x;
  if(x % 2 == 0)
  {
    while(x)
    {
      s += x % 10;
      x /= 10;
    }
  }
}
cout << s << endl;
```

Обратите внимание, что в программе большая вложенность: цикл, в нём условие, внутри еще один цикл, в котором два действия. С помощью `continue` можно уменьшить вложенность, сразу проверив чётность `x`, и, если он нечётный, перейдя на следующую итерацию:

```
int s = 0;
for(int i = 1; i <= 10; ++i)
{
  int x;
  cin >> x;
  if(x % 2 != 0) continue;

  while(x)
  {
    s += x % 10;
    x /= 10;
  }
}
cout << s << endl;
```

Оператор `continue` всегда переводит управление на конец текущей итерации.

4.6 Пошаговая прокрутка циклов

Чтобы лучше понять тот или иной алгоритм полезно самостоятельно "смоделировать" работу цикла, то есть посмотреть, как меняются переменные на каждой итерации, как срабатывают те или иные условия и т.д. Значения переменных заносятся в таблицу. Рассмотрим следующий алгоритм:

```
int a, b;
cin >> a >> b;

int i = 0;
while(a > 0 && b > 0)
{
  if(a > b) a -= b;
  else b -= a;
  ++i;
}

cout << i << endl;
```

В данном примере вводятся два числа, после этого начинается цикл, который продолжается пока оба числа больше нуля. На каждой итерации цикла мы из большего числа вычитаем меньшее, а затем увеличиваем переменную `i` на 1. Пусть пользователь ввел `a = 78`, `b = 43`. Требуется определить, что выведет данная программа. По сути, требуется посчитать, сколько итераций выполнится.

Составим таблицу, в которую внесем значения всех переменных и напишем результаты сравнения в `while`.

i	a	b	a > 0	b > 0	a > 0 && b >0	a > b
0	78	43	true	true	true	true
1	35	43	true	true	true	false
2	35	8	true	true	true	true
3	27	8	true	true	true	true
4	19	8	true	true	true	true
5	11	8	true	true	true	true
6	3	8	true	true	true	false
7	3	5	true	true	true	false
8	3	2	true	true	true	true
9	1	2	true	true	true	false
10	1	1	true	true	true	false
11	1	0	true	false	false	!

Видно, что при `a = 1` и `b = 0` условие в `while` не сработает и цикл завершится. При этом последнее значение `i` равно 11.

Рассмотрим пример вычисления n-го числа Фибоначчи:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Введите n: ";
  int n;
  cin >> n;

  int a_prev = 1, a = 1;

  for(int i = 2; i < n; ++i)
  {
    const int t = a;
    a = a + a_prev;
    a_prev = t;
  }

  cout << "Ответ: " << a << endl;
}
```

Выполните прокрутку при `n = 8` и заполните таблицу:

a_prev	a	i	i < n	t
1	1	2	true	1
1	2	3	true	2

Глава 5

Функции

Решение любой задачи естестественно стараться свести к решению нескольких маленьких подзадач. Если в программе приходится в нескольких местах выполнять один и тот же алгоритм, тогда имеет смысл выделить этот алгоритм в подпрограмму и вызывать ее при необходимости. Такая подпрограмма будет называться *функцией*.

Пусть в программе требуется выводить числа не арабскими цифрами, а палочками (то есть число 5 будет записано как ||||| и т.д.). Код, который преобразует число в такую запись написать не сложно:

```
int x = 5;
for(int i = 0; i < x; ++i)
  cout << '|';
```

Допустим, что требуется вывести значения 3-х переменных на экран. Придется копировать один и тот же код:

```
#include <iostream>
using namespace std;
int main()
{
  int a = 5;
  int b = 20;
  int c = 3;

  cout << "a: ";
  for(int i = 0; i < a; ++i)
    cout << '|';
  cout << endl;

  cout << "b: ";
```

```
  for(int i = 0; i < b; ++i)
    cout << '|';
  cout << endl;

  cout << "c: ";
  for(int i = 0; i < c; ++i)
    cout << '|';
  cout << endl;
}
```

Программа выведет:

```
a: |||||
b: ||||||||||||||||||||
c: |||
```

В этом решении есть проблема: если возникнет необходимость изменить вывод, то придется вносить слишком много изменений в разные части программы.

5.1 Определение функции

Определить функцию — значит полностью описать алгоритм ее работы. Функция имеет следующий вид:

```
тип имя(список параметров)
{
  тело функции
}
```

Функция должна быть объявлена до того момента, где она будет вызвана.

Напишем функцию, которая бы принимала одну переменную типа `int` и выводила ее палочками на экран. Назовем эту функцию my_print. Она будет типа `void`, это означает, что функция не возвращает какое либо значение (об этом речь пойдет ниже), а параметр у функции – число, которое нужно вывести.

```
#include <iostream>
using namespace std;

// определяем функцию my_print до начала main()

void my_print(int number)
{
  for(int i = 0; i < number; ++i)
    cout << '|';
  cout << endl;
```

```
11  }
12
13  int main()
14  {
15    int a = 5;
16    int b = 20;
17    int c = 3;
18
19    cout << "a: ";
20    my_print(a);
21
22    cout << "b: ";
23    my_print(b);
24
25    cout << "c: ";
26    my_print(c);
27  }
```

Как нам известно, выполнение программы начинается с функции `main` независимо от того, что написано до нее. Поэтому выполнение программы начнется с 13-й строки, а на 20-й программа перейдет в функцию `my_print` и начнет выполнять ее. При этом значение переменной `a` будет скопировано во внутреннюю переменную `number`. Когда функция `my_print` выполнится (11-я строка), программа вернется на 20-ю строку и продолжит выполнение.

Можно добавить второй параметр в функцию — имя выводимой переменной:

```
#include <iostream>
#include <string>
using namespace std;

// определяем функцию my_print до начала main()

void my_print(string name, int number)
{
  cout << name << ": ";
  for(int i = 0; i < number; ++i)
    cout << '|';
  cout << endl;
}

int main()
{
  int a = 5;
  int b = 20;
```

```
  int c = 3;

  my_print("Значение переменной a", a);
  my_print("Значение переменной b", b);
  my_print("Значение переменной c", c);
}
```

Программа выведет:

```
Значение переменной a: |||||
Значение переменной b: ||||||||||||||||||||
Значение переменной c: |||
```

Таким образом мы упростили функцию `main` и создали удобное средство для вывода чисел. Вообще говоря, это все еще не самое удачное решение, но, оно несомненно лучше, чем копировать вывод много раз.

5.2 Возвращаемое значение

В предыдущем параграфе мы рассмотрели создание функции, которая выполняла некоторые действия. Результат работы этой функции пользователь видел на экране. Во многих случаях от функции требуется что-то вычислить или определить, в таком случае результатом ее работы является значение некоторого типа. Приведем пример функции, вычисляющей факториал числа. Результат ее работы — значение типа `int`.

```
#include <iostream>
using namespace std;

int fact(int number)
{
  int p = 1;

  for(int i = 1; i <= number; ++i)
    p *= i;

  return p; // возвращаем вычисленное значение
}

int main()
{
  cout << "Введите число: ";
  int a;
  cin >> a;
```

```
  cout << a << "! = " << fact(a) << endl;
}
```

При выводе на экран программа перейдет в функцию `fact`, будет вычислено значение факториала и за счет строки 12 это значение вернется на место вызова функции, то есть встанет в `cout` вместо `fact(a)`. Пример работы программы:

```
Введите число: 6
6! = 720
```

5.3 Объявление и определение функций

Как мы уже упомянули, функция должна быть объявлена до места вызова, хотя, мы так и не сказали, что занчит *объявление* функции. Обратимся к уже рассмотренной ранее функции, возвращающей факториал числа

```
int fact(int number)
{
  int p = 1;

  for(int i = 1; i <= number; ++i)
    p *= i;

  return p; // возвращаем вычисленное значение
}
```

Здесь приведено *определение* функции `fact`, то есть вся функция, включая тело. *Объявлением* или *прототипом* этой функции будет лишь первая строка без тела:

```
int fact(int number); // прототип функции fact
```

причем, указывать имена параметров необязательно, достаточно лишь перечислить типы:

```
int fact(int); // прототип функции fact
```

Объявление функции подсказывает компилятору, что такая функция есть, и она будет определена далее в тексте программы. Таким образом, объявив заранее все функции в программе, их определения можно помещать, не беспокоясь о местах вызова:

```
#include <iostream>

using namespace std;

int fact(int number); // объявляем функцию fact
```

```
int main()
{
  cout << "Введите число: ";
  int a;
  cin >> a;

  cout << a << "! = " << fact(a) << endl; // вызываем функцию fact
}

int fact(int number) // определяем функцию fact
{
  int p = 1;

  for(int i = 1; i <= number; ++i)
    p *= i;

  return p;
}
```

5.4 Перегрузка функций

5.5 Параметры по умолчанию

5.6 Некоторые полезные функции

Глава 6

Одномерные массивы

Массивом называется последовательность элементов одного типа, имеющих общее имя. Обращение к элементу массива осуществляется через его номер (индекс). В общем виде объявление массива можно записать так:

```
T v[N];
```

Здесь `T` — произвольный тип, `v` — имя массива, `N` — количество элементов. Важно помнить, что `N` должно быть целочисленной константой. То есть, это может быть некоторое число или предварительно объявленная константа, но не переменная. Рассмотрим несколько примеров.

```
char v[10]; //правильно

int N = 5;
float v[N]; //неправильно, N - переменная

const int N = 5;
float v[N]; //правильно
```

Если в массиве `N` элементов, тогда их номера будут иметь значения $\{0, 1, 2, \ldots, N-1\}$. При обращении к элементу массива, которого нет (например к `v[N]` или к `v[N + 10]`), произойдет выход за границы массива. В лучшем случае это приведет к некорректной работе программы, в худшем — к аварийному завершению программы.

Напишем программу, считывающую 5 целых чисел в массив и меняющую знак у всех отрицательных элементов.

```
#include <iostream>
using namespace std;
```

```
int main()
{
  const int N = 5;
  int v[N]; //объявляем массив

  for(int i = 0; i < N; ++i)
  {
    cout << "Введите элемент массива " << i << ": ";
    cin >> v[i]; //считываем в цикле числа
  }

  //перебираем массив и ищем отрицательные элементы
  for(int i = 0; i < N; ++i)
    if(v[i] < 0) //если элемент i отрицательный
      v[i] = -v[i]; //меняем знак

  //выводим элементы на экран
  for(int i = 0; i < N; ++i)
    cout << v[i] << " ";
  cout << endl;
}
```

```
Пример ввода: 3 -8 1 -2 -5
На экране появится: 3 8 1 2 5
```

6.1 Класс `std::vector`

Обычные массивы достаточно примитивны и часто оказываются неудобными. Обычно размер данных неизвестен, а создание массивов «с запасом» безосновательно увеличивает требуемый объем оперативной памяти.

Стандратный контейнер `vector` определен в библиотеке `<vector>` работает «как обычный массив», но предоставляет удобные функции для работы с данными. Более того, `vector` не требует предварительного задания количества элементов.

В следующем примере создается пустой вектор из элементов типа `int`.

```
vector<int> v;
```

В вектор можно добавлять элементы при помощи функции `push_back`:

```
vector<int> v;
v.push_back(6);
v.push_back(58);
v.push_back(-20);
```

Теперь в векторе 3 элемента. Их можно менять, например, вместо 6 присвоим в нулевой элемент значение 3:

```
vector<int> v;
v.push_back(6);
v.push_back(58);
v.push_back(-20);
v[0] = 3;
```

Заметим, что при создании вектор пустой, поэтому следующий код является неправильным:

```
vector<int> v;
v[0] = 6;
v[1] = 58;
v[2] = -20;
```

Обращаться по индексу можно лишь к существующим элементам. В данном случае в векторе нет элементов с номерами 0, 1 и 2. Их можно создать, используя функцию `push_back`, либо указать начальный размер вектора при объявлении. Следующий пример демонстрирует правильное решение:

```
vector<int> v(3); // объявили вектор и создали в нем 3 элемента
v[0] = 6;
v[1] = 58;
v[2] = -20;
```

Чтобы узнать, сколько в данный момент в векторе элементов, используется функция `size`:

```
vector<int> v;
v.push_back(6);
v.push_back(58);
v.push_back(-20);
cout << "Количество элементов в векторе v: " << v.size() << endl;
```

Рассмотрим простой пример: пользователь вводит некоторое количество чисел, а программа печатает эти числа в обратном порядке. Причем, число 0 обозначает конец ввода. Как видно, изначально неизвестно, сколько чисел введет пользователь.

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

using namespace std;

int main()
```

```
{
  vector<int> v; // вектор из элементов типа int

  // считываем пока не 0 и добавляем каждый элемент в вектор:
  int x;
  while((cin >> x) && x != 0)
    v.push_back(x);

  // выводим вектор с конца:
  for(size_t i = v.size() - 1; i >= 0; --i)
    cout << v[i] << " ";
  cout << endl;
}
```

```
Пример ввода: 4 9 2 2 3 0
На экране появится: 3 2 2 9 4
```

Глава 7

Введение в строки

Для представления строк в C++ используется класс `string` из заголовочного файла `<string>`.

Рассмотрим следующий пример: программа спрашивает имя пользователя и приветствует его:

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

using namespace std;

int main()
{
  cout << "Как вас зовут? ";
  string name;
  cin >> name;
  cout << "Привет, " << name << "! =)" << endl;
}
```

```
Пример ввода: Петр
Пример вывода: Привет, Петр! =)
```

Строки типа `string` имеют вспомогательные функции. Например, чтобы узнать длину строки (количество символов в строке) можно воспользоваться функцией `size`:

```
cin >> name;
cout << "Длина имени: " << name.size() << endl;
```

```
Пример ввода: Петр
Пример вывода: Длина имени: 4
```

Заметим, что если пользователь введет два слова, разделенных пробелом (или несколькими пробелами), считается только первое слово, а второе останется в буфере для дальнейшего чтения. Чтобы считать строку целиком воспользуемся функцией `getline`:

```
string s;
cout << "Введите строку: "
getline(cin, s);
```

Со строкой можно работать как с массивом символов, то есть обращаться к отельным символам по индексу. Например, пусть строка `s` начинается с символа `'a'`:

```
s[0] = 'a';
```

Символьные константы указываются именно в апострофах, а не кавычках, так как даже один символ в кавычках считается строкой, а не символом. То есть запись `s[0] = "a"` была бы неверна. С другой стороны, в апострофах указывается всегда один символ. Например: `'a'`, `'b'`. Некоторые символы имеют специальную форму записи: `'\n'` (переход на новую строку), `'\\'` (косая черта) и т.д.

Напишем программу, которая все символы подчеркивания (`'_'`) заменяет на пробелы (`' '`).

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

using namespace std;

int main()
{
  cout << "Введите строку: " << endl;
  string s;
  getline(cin, s);

  //перебираем строку посимвольно
  for(int i = 0; i < s.size(); ++i)
  {
    if(s[i] == '_')
      s[i] = ' ';
  }

  cout << "Результат: " << s << endl;
}
```

```
Пример ввода: Строка, разделенная_символами_подчеркивания.
Пример вывода: Строка, разделенная символами подчеркивания.
```

Глава 8

Работа с текстовыми файлами

В стандартной библиотеке C++ для работы с файлами используются *файловые потоки*. Вообще, потоком, часто называется объект, позволяющий считывать и записывать данные последовательно. Чтение и запись потоковых данных ничем не отличается от работы с `cin` и `cout`. Для использования файловых потоков необходимо подключить библиотеку `<fstream>`.

8.1 Запись файлов

Для записи файлов используем тип `ofstream`. Откроем новый файл `numbers.txt` и запишем в него 10 случайных чисел. Если такой файл уже есть, то он будет перезаписан (то есть прежднее его содержание будет удалено).

```
#include <fstream> // для ofstream
#include <cstdlib> // для rand()

using namespace std;

int main()
{
  ofstream f("numbers.txt");

  // проверим, удалось ли открыть файл
  if( !f )
  {
    cout << "Не удалось открыть файл" << endl;
    return 1; // выходим из main() с ненулевым кодом ошибки
```

```
  }

  // записываем в файл 10 случайных чисел:
  for(int i = 0; i < 10; ++i)
    f << rand() % 51 - 25 << " ";
}
```

Данные в файловый поток записываются аналогично cout:

```
f << "               *   " << endl;
f << "                *  " << endl;
f << "      ******    * " << endl;
f << "                * " << endl;
f << "      ******    * " << endl;
f << "                *  " << endl;
f << "               *   " << endl;
```

Чтобы открыть существующий файл для записи, не стирая его содержимое, используется флаг ios::app:

```
ofstream f("numbers.txt", ios::app);
```

Теперь данные будут дописываться в конец файла.

8.2 Чтение файлов

Для чтения из файлового потока используем тип ifstream, чтение из которго происходит аналогично cin. Напишем программу, которая считывает из файла все числа и вычисляет их сумму. В качестве входного файла можно использовать файл, который мы создали с помощью ofstream. При этом все числа должны разделяться пробелами, табуляциями или пустыми строками (не важно в каком количестве).

```
#include <fstream> // для ifstream
#include <cstdlib> // для rand()
using namespace std;

int main()
{
  ifstream f("numbers.txt");

  // проверим, удалось ли открыть файл
  if( !f )
  {
    cout << "Не удалось открыть файл" << endl;
    return 1; // выходим из main() с ненулевым кодом ошибки
  }

  int s = 0;
  int x;
  while(f >> x) // пока удается считываеть
    s += x;
```

```
  cout << "Сумма чисел в файле: " << s << endl;
}
```

Часть II

Углублённый курс

Глава 9

Ссылки и указатели

9.1 Ссылки

Ссылка является альтернативным именем объекта. Для типа T тип T& является ссылкой на T. Например:

```
int x = 5;
int& u = x;
```

Теперь u и x имена одного и того же объекта:

```
int x = 5;
int& u = x;
cout << x << endl; // на экране: 5
cout << u << endl; // на экране: 5
u = 10;
cout << x << endl; // на экране: 10
cout << u << endl; // на экране: 10
```

9.2 Указатели

Указатель — это адрес в памяти некоторого объекта. Рассмотрим пример:

```
int x = 15;
```

После объявления переменной x в памяти выделилось несколько байт (на современных компьютерах — 4 или 8) и в них было записано число 15. Обычно нас не интересовало, где именно выделена эта память.

Для некоторого типа T тип T* является указателем на T, например:

```
int* p; // указатель на int
```

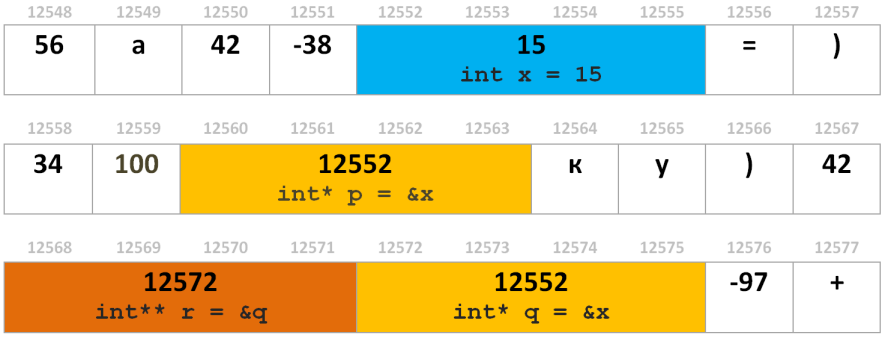
В данном примере p может хранить адрес некоторой переменной типа int. Пока она не хранит чего-либо осмысленного. Поскольку p — адрес, в нее нельзя присвоить что-то напрямую. Следующий пример вызовет ошибку компиляции:

```
int* p = 123; // ошибка
```

При этом в p можно присвоить адрес уже созданной переменной. Для получения адреса объекта используется оператор получения адреса, который обозначается символом &:

```
int x = 15;
int* p = &x; // указатель p хранит адрес x
```

Что при этом происходит в памяти компьютера? Будем считать, что размер типа int и размер указателя равны 4 байта (это верно для большинства 32-разрядных компьютеров). Пусть при объявлении переменной x операционная система выделила 4 байта: с 12552 по 12555 В этих 4-х байтах разместилось число 15. Когда мы создали указатель p и присвоили ему адрес x, где-то в оперативной памяти ОС выделила нам еще 4 байта и записала в них адрес переменной x, то есть число 12548. Это проиллюстрировано в первых двух строках на следующем рисунке. Обратите внимание, что в белых ячейках находятся какие-то произвольные данные, главное, что среди них мы видим переменную x и указатель p.



Возникает вопрос: как получить доступ к значению переменной, имея указатель на нее. Для этого используется оператор *разыменования* (*dereference*), который обозначается звездочкой *:

```
int x = 15;
cout << x << endl;
int* p = &x; // указатель p хранит адрес x
x = 8;
cout << *p << endl; // выводим значение по адресу p
*p = -20;
cout << x << endl;
```

На экране:

```
15
8
-20
```

Видно, что, аналогично ссылке, имея адрес, можно выполнять любые операции со значением по этому адресу.

Не путайте оператор получения адреса `&` с амперсандом, обозначающим ссылку. Амперсадны в выражениях `int& x = y;` и `int* p = &x;` имеют совершенно разный смысл.

Вернемся к иллюстрации. В последней строке показано, что мы можем иметь несколько указателей на один и тот же объект (указатели `p` и `q`), а также хранить адреса других указателей (то есть указатели на указатели). Приведем пример, соответствующий этому рисунку:

```
int x = 15;
int* p = &x; // p - указатель на x
int* q = &x; // q - указатель на x
// обратите внимание, что p != q,
// но *p и *q - один и тот же объект, то есть *p == *q
int** r = &q; // r - указатель на q или указатель на указатель на x
```

Как мы увидим в следующих главах, адрес объекта (то есть указатель на него) — мощнейший инструмент, который позволит создавать интересные и гибкие объекты.

9.2.1 Операторы `new` и `delete`

Чтобы создать объект и получить его адрес совсем не обязательно предварительно создавать переменну. Оператор `new` создает объект и возвращает его адрес:

```
int* p = new int;
*p = 2013;
```

Теперь в памяти по адресу `p` находится значение типа `int`, равное 2013. Важнейшее отличие объектов, созданных с помощью оператора `new` от обычных переменных заключается в том, объекты, созданные с помощью `new` не удаляются автоматически. Например:

```
void f()
{
  int x = 2013;
} // здесь переменная x будет уничтожена
```

Но если создать `int` динамически (`new`):

```
void f()
{
  int* p = new int;
  *p = 2013;
} // указатель p будет уничтожен, но объект, на который он указывает -- не
```

Мы сейчас не будем приводить реальный пример, где такое поведение полезно. Скажем лишь, что при работе с объектами бывает необходимо гарантировать, что они не будут разрушены до определенного момента, в этих случаях удобно использовать динамическое создание.

Чтобы уничтожить объект, созданный при помощи `new` используется оператор `delete`:

```
void f()
{
  int* p = new int;
  *p = 2013;
  cout << *p;
  delete p; // удаляем объект по адресу p
}
```

9.3 Динамическое создание массивов

Для выделения памяти под массивы элементов используются операторы `new[]` и `delete[]`. В квадратных скобках `new` указывается количество элементов, а у `delete` квадратные скобки всегда пустые. В отличие от обычных массивов, количество элементов массива может быть заданно не только константой, но и переменной:

```
int n;
cout << "Введите количество элементов: ";
cin >> n;

int* v = new int[n]; // создан массив из n элементов
```

Как мы уже сказали, объект, созданный `new` не удаляется из памяти автоматически, поэтому программист должен не забыть вызвать `delete[]`

```
delete[] v; // удалить массив v
```

Глава 10

Исключения

В любой программе приходится иметь дело с теми или иными внутренними ошибками. Например, при загрузке игры оказалось, что на диске нет необходимых графических файлов, или при вычислении некоторого выражение произошло деление на ноль. Обычно в таких случаях продолжать выполнение функции невозможно (точнее — бессмысленно) и требуется выйти в функцию, из которой была вызвана данная, передав в неё информацию о возникшей ошибке. Иногда требуется выйти сразу из нескольких вложенных функций.

10.1 Пример без исключений

Пусть некоторая функция `double eval(double a, double b, char op)` вычисляет сумму, разность, произведение или частное двух чисел `a, b` в зависимости от оператора `op` (`eval` — от слова evaluate):

```
#include <iostream>
using namespace std;

double eval(double a, double b, char op)
{
  switch(op)
  {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return a / b;
  }
}

int main()
```

```
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;

  const double result = eval(a, b, op);
  cout << "Результат: " << resutl;
}
```

В этом примере две проблемы. В первую очередь, если `b` равно нулю, то произойдет ошибка. Также непонятно, что будет если `op` не равен всем перечисленным символам? Тогда выполнение программы дойдет до конца функции и вернется какое-то произвольное значение. Наша задача:изменить `eval` так, чтобы в `main()` мы могли узнать результат вычисления, а если в `eval` произошла ошибка (то есть результат не имеет смысла), то узнать об этом и проинформировать пользователя.

Прежде чем обсуждать механизм исключений, рассмотрим один из возможных способов информирования об ошибках. Изменим `eval` так, что она возвращает не результат применения оператора к двум переменным, а значение типа `bool`. В случае, если выполнение прошло успешно, eval вернет `true`, а в случае ошибки — `false`. Но как мы тогда получим само значение? Мы добавим еще один параметр при вызове функции — ссылку на переменную, в которую `eval` должна записать результат, то есть:

```
bool eval(double a, double b, char op, double& result)
{
  switch(op)
  {
    case '+': result = a + b; break;
    case '-': result = a - b; break;
    case '*': result = a * b; break;
    case '/':
    {
      if(b == 0) // если деление на ноль
        return false;
      result a / b;
    }
  }
  default: return false; // если op не равен +,-,*,/

  return true; // успешно вычислено
}

int main()
```

```
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;

  double result;
  if(eval(a, b, op, result)
    cout << "Результат: " << result << endl;
  else
    cout << "Произошла ошибка" << endl;
}
```

Это плохое решение. По сути мы все так же возвращаем ответ, но теперь прикрепляем к нему «записку», в которой написано, верить ли этому ответу. Функция `eval` заметно усложнилась, теперь помимо работы с `result` мы должны не ошибиться с возвращаемым значением. Более того, такой способ информирования слишком пассивный. Если программист забудет проверить то, что возвращает `eval`, то ошибка просто будет проигнорирована и программа далее будет работать с данными, которые по сути не имеют смысла. И, напоследок, если `eval` вызывается в нескольких вложенных функциях, а при ошибке их дальнейшее выполнение не имеет смысла, то выход из этой вложенности окажется слишком громоздким.

10.2 Добавим исключения

Исключение (*exception*) является естественным механизмом для информирования разных частей программы о произошедшей исключительной ситуации (в частности — ошибки). Когда происходит ошибка, программа может сгенерировать (иногда говорят «бросить») исключение любого типа с помощью ключевого слова `throw`. В этот момент выполнение программы уходит на уровень выше, где оно может быть перехвачено и проанализировано с помощью ключевого слова `catch`.

Рассмотрим наш пример. Обратите внимание на то, что теперь нет необходимости возвращать `true` или `false`, поэтому будем возвращать сам результат `double`:

```
double eval(double a, double b, char op)
{
  switch(op)
  {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
```

```
    case '/':
      if(b == 0) throw "Деление на 0"; // выходим из функции
      return a / b;
    default: throw "Неизвестный оператор";
  }
}

int main()
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;

  const double result = eval(a, b, op);
  cout << "Результат: " << result;
}
```

Исключения поднимаются вверх по системе вложенных функций, словно пузырьки. Если во время не перехватить, то пузырек всплывет выше `main()` и программа завершится с ошибкой.

Так как при вызове `eval` в `main` мы не перехватываем (`catch`) исключение, при делении на ноль, функция eval завершит работу, исключение попадет в `main`, и, не будучи перехваченным, завершит программу:

```
Введите выражение: 4 / 0
terminate called after throwing an instance of 'char const*'
```

Добавим в `main()` перехват исключения типа `const char*`. При перехвате исключений в некоторой области программы, данная область заключается в блок `try`, за которым перечисляются типы исключений с помощью ключевого слова `catch`:

```
int main()
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;

  try // далее возможны исключения
  {
    const double result = eval(a, b, op);
    cout << "Результат: " << result;
  }
  catch (const char* s) // перехватываем исключение типа const char*
```

```
  {
    cout << "Произошла ошибка: " << s << endl;
  }
}
```

```
Введите выражение: 4 / 0
Произошла ошибка: Деление на 0
```

При этом программа не завершается сразу, а продолжает выполняться после блоков `catch`.

Такой способ информирования об ошибках имеет один существенный недостаток: каждый блок `catch` перехватывает определенный тип исключений. В нашем примере функция `eval` генерирует исключения в двух случаях: при делении на ноль и при неизвестном операторе, но в `main` мы не можем определить, какая именно ошибка произошла. Лучше для каждого исключения создать собственный тип данных. Как это сделать мы обсудим в разделе «Наследование», а пока лишь коротко рассмотрим уже готовые типы исключений из стандартной библиотеки.

10.3 Стандартные исключения

Стандартные исключения определены в файле `<stdexcept>`, они используются в различных ситуациях, и для создания собственных типов исключений. Здесь мы не будем давать обзор всех исключений. Скажем лишь, что все стандартные исключения наследуются от типа `std::exception` и разделяются на исключения для логических ошибок и ошибок времени выполнения (runtime errors).

Логические ошибки — ошибки, возникающие во внутренней логике программы. К ним относятся:

- `logic_error` — общие ошибки.
- `domain_error` — нарушение области определения.
- `invalid_argument` — неправильное значение аргумента (например у функции).
- `length_error` — ошибки, связанные с размером данных.
- `out_of_range` ошибки, возникающие когда некоторое значение выходит за границу допустипых значений.

Ошибки времени выполнения обычно возникают при не зависящих от программы обстоятельствах. Например при неправильном запросе или отсутствии на диске необходимого программе файла.

- `runtime_error` — общие ошибки при выполнении программы.
- `range_error` — выход за пределы отрезка, неправильно заданные граничные значения и т.д.
- `overflow_error` — переполнение. Часто такая ошибка возникает при попытке добавить данные в объект, в котором не осталось свободного места.
- `underflow_error` — антипереполнение. Обычно возникает при попытке обратиться к элементу в пустом контейнере (то есть когда нет ни одного элемента и обращаться не к чему).

Так как все стандартные исключения наследуются от `std::exception`, все они имеют функцию `what()`, возвращающую пояснение к ошибке. Покажем, как можно применить исключения `domain_error` и `invalid_argument` в нашей программе:

```
#include <iostream>
#include <stdexcept>

using namespace std;

double eval(double a, double b, char op)
{
  switch(op)
  {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/':
    {
      if(b == 0)
        throw domain_error("Деление на 0");
      return a / b;
    }
    default:
      throw invalid_argument("Неизвестный оператор");
  }
}

int main()
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;
```

```
  try
  {
    const double result = eval(a, b, op);
    cout << "Результат: " << resutl;
  }
  catch(domain_error& err)
  {
    cout << "Произошла ошибка в области определения: " << err.what();
  }
  catch(invalid_argument& err)
  {
    cout << "Неправильный аргумент: " << err.what();
  }
}
```

```
Введите выражение: 4 / 0
Произошла ошибка области в определения: Деление на 0
```

Обратите внимание, что исключение перехватывается по ссылке `catch(domain_erro err)`. Полностью смысл этого станет понятен когда мы познакомимся с классами и наследованием. Пока лишь заметим, что так как оба типа `domain_error` и `invalid_argument` наследуют `exception`. Благодаря перехвату исключений по ссылке, в случаях, когда нас не интересует конкретный тип ошибки, мы можем перехватывать лишь базовый тип:

```
double eval(double a, double b, char op)
{
  switch(op)
  {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/':
    {
      if(b == 0)
        throw domain_error("Деление на 0");
      return a / b;
    }
    default:
      throw invalid_argument("Неизвестный оператор");
  }
}

int main()
```

```
{
  double a, b;
  char op;
  cout << "Введите выражение: ";
  cin >> a >> op >> b;

  try
  {
    const double result = eval(a, b, op);
    cout << "Результат: " << resutl;
  }
  catch(exception& err)
  {
    cout << "Произошла ошибка: " << err.what();
  }
}
```

```
Введите выражение: 4 / 0
Произошла ошибка: Деление на 0
```

Глава 11

Структуры