Н. А. Литвиненко

Технология программирования на C++

Начальный курс

- **Основы программирования на C++** – описание синтаксиса языка C, рассмотрение приемов и методов программирования в стиле классического C
- **Классы** – введение понятия класса, рассмотрение шаблонных классов, вопросов наследования и построения сложных классов
- **Потоковый ввод/вывод** – организация современного подхода к организации ввода/вывода при помощи потоковых классов
- **Стандартная библиотека шаблонов STL** – описание стандартной библиотеки и многочисленные примеры ее использования
- **Организация оконного интерфейса** – структура Windows-программы, методы обработки стандартных сообщений. Программирование с использованием API-функций

УЧЕБНОЕ ПОСОБИЕ

bhv

Н. А. Литвиненко

Технология программирования на C++

Начальный курс

Допущено УМО вузов по университетскому политехническому образованию
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению 654600
«Информатика и вычислительная техника»

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800.92C++(075.8)
ББК 32.973.26-018.1я73
Л64

Литвиненко Н. А.

Л64 Технология программирования на C++. Начальный курс. — СПб.: БХВ-Петербург, 2005. — 288 с.: ил.

ISBN 5-94157-655-2

Рассмотрены основы программирования на C++, начиная с описания синтаксиса языка C, приемов и методов программирования в стиле классического C до введения понятий классов, шаблонов классов и вопросов наследования. Уделено особое внимание использованию стандартной библиотеки шаблонов STL. Представлен современный подход к организации ввода/вывода при помощи потоковых классов. Рассматривается техника создания простейших Windows-приложений с использованием API-функций. Материал иллюстрируется многочисленными примерами.

Для студентов и преподавателей технических вузов и самообразования

УДК 681.3.068+800.92C++(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Дарья Масленникова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.04.05.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 23,22.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-655-2

© Литвиненко Н. А., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Введение

Основной причиной, побудившей автора взяться за написание данной книги, послужило желание создать учебник для студентов, изучающих дисциплины "Технология программирования" и "Объектно-ориентированное программирование". Несмотря на обилие различной литературы по языку программирования C++, сложно порекомендовать какую-то одну книгу в качестве подобного учебника. Из недавно изданных можно отметить учебник и практикум Павловской [3, 4] — серьезные издания со строгим изложением материала, хорошо подходящие в качестве справочных пособий. Однако для первоначального изучения языка требуется менее формализованный подход, который автор и попытался реализовать в данном учебном пособии, построенном на основе курса лекций, читаемых студентам специальностей "Программное обеспечение вычислительной техники и автоматизированных систем" и "Информационные системы и технологии". В качестве дополнительной литературы можно рекомендовать очень неплохое издание Р. Лафоре [2] из серии "Классика Computer Science". Книга отличается хорошим стилем изложения, но приведенные в ней примеры не всегда удачны.

При построении курса возникает вопрос о его структуре и содержании. Язык программирования C++ является языком объектно-ориентированного программирования (ООП). Существует несколько подходов к изучению данного языка. Иногда начинают с классического C и лишь после этого переходят к расширению языка C++, собственно классам и ООП. В последнее время многие авторы сразу начинают изложение с введения классов, однако автор данной книги не считает это правильным, выбирая гибридный подход к изложению материала — расширения языка включены с самого начала описания, а переход к классам осуществлен лишь после того, как детально разобран процедурный подход к программированию.

Правда, и здесь трудно удержаться и не воспользоваться возможностями потокового ввода/вывода. Этот современный подход используется в данной книге сразу, а соответствующие функции классического C не рассмотрены.

В первых главах книги не рассмотрены также и графические построения, поскольку в современных операционных системах графика тесно связана с оконным интерфейсом, чему у нас посвящена отдельная, последняя глава. В ней собраны простейшие графические функции и приведены примеры их использования.

Книга состоит из пяти глав.

Глава 1. "Основы программирования на C++" — описание синтаксиса языка C, рассмотрение приемов и методов программирования в стиле классического C.

Глава 2. "Классы" — вводится понятие класса, рассматриваются шаблонные классы, вопросы наследования и построения сложных классов. Приводятся многочисленные примеры.

Глава 3. "Потоки" — рассматривается современный подход к организации ввода/вывода при помощи потоковых классов.

Глава 4. "Стандартная библиотека шаблонов STL" — описание стандартной библиотеки. На многочисленных примерах рассмотрено использование STL.

Глава 5. "Организация оконного интерфейса" — рассматривается структура скелета Windows-программы, методы обработки стандартных сообщений. Приведены многочисленные примеры вывода текста и простейшей графики. Программирование основано на непосредственном использовании API-функций.

Сразу оговоримся для будущих критиков, что данная книга ни в коей мере не претендует ни на полноту, ни на строгость, все это принесено в жертву ясности изложения. Некоторые вопросы рассматриваются поверхностно, некоторые вообще опущены. Это лишь вводный курс, цель которого — изучение синтаксиса языка C++ и основных методов программирования. В книге достаточно поверхностно рассмотрено создание Win32-приложений — лишь на том уровне, чтобы у читателя сложилось представление о структуре стандартной Windows-программы и принципах ее функционирования.

В заключение отметим, что все примеры, рассмотренные в книге, являются фрагментами работающих программ, а тексты некоторых программ приведены полностью в листингах. Все программы тестировались в MS Visual C++ 6.0 под управлением операционной системы Windows XP.

ГЛАВА 1

Основы программирования на C++

Часто формализм языка рассматривается безотносительно конкретной реализации. И все же читателю будет предложено использовать в качестве основного инструмента для построения программ консольное приложение среды MS Visual C++ 6.0 (Win32 Console Application), ориентируясь на приведенные далее примеры, реализованные в этой среде программирования. Язык программирования в классическом стиле, рассмотренный в данной главе, будет дополнен лишь элементами расширения, принципиально не меняющими его сути. Однако для организации ввода/вывода воспользуемся подходом C++, применив потоковые классы, подробнее о которых будет рассказано далее, в *главе 3*.

Структура классической C-программы

#*Директивы препроцессора*	Объявление файлов включения
Объявление функций	Описание прототипов функций
Описание глобальных переменных и констант	

`main` (*параметры*) // *заголовок программы*
{
Описание локальных переменных и констант
Операторы
}
	Головная функция

Описание функций	

В основу любого компилятора для языка C и C++ заложена однопроходовая схема, т. е. процесс построения программы осуществляется за один проход по тексту. Таким образом, все имена переменных и констант программы должны быть объявлены до их первого использования.

Препроцессор

Для создания дополнительного сервиса, а также улучшения переносимости программ, в C++ реализован так называемый *препроцессор*, который выполняет набор инструкций перед началом компиляции программы. Все директивы препроцессора начинаются с символа #.

#include

Директива включения файла. Обычно используется для вызова так называемых файлов включений, содержащих *прототипы* (т. е. описания) библиотечных функций. Также позволяет включить часть текста программы, записанного в другой файл. По сути дела, как в текстовом редакторе, объединяются два файла.

Формат директивы:

```
#include <имя файла включения>
```

Здесь угловые скобки < > являются частью конструкции и указывают на то, что файл ищется в каталоге файлов включения, прописанном в среде программирования.

Формат директивы:

```
#include "имя файла включения"
```

Двойные кавычки " " означают, что поиск файла включения осуществляется в том же каталоге, в котором открыт исходный файл.

```
#include <stdlib.h>   // Включение стандартной библиотеки
#include "meny.h"     // Включение файла meny.h
```

#define

Определение символического обозначения. Устаревшая конструкция, вместо нее более целесообразно определять константные переменные, но об этом чуть позже.

Формат директивы:

```
#define ИМЯ ЗНАЧЕНИЕ
```

В результате действия директивы все символические обозначения ИМЯ в тексте программы заменятся на ЗНАЧЕНИЕ перед началом компиляции. Исключе-

ние составляют текстовые константы, заключенные в двойные кавычки "...", внутри которых подстановки не производятся. Опять же, проводя аналогию с текстовым редактором, это просто операция *"поиск и замена"*, например:

```
#define PI 3.14159
```

Перед компиляцией все переменные `PI` будут заменены их значением `3.14159`.

Примечание

Компилятор языка C++ различает строчные и прописные буквы, поэтому для выделения в тексте программы символических констант их принято писать заглавными буквами.

Некоторые другие директивы мы рассмотрим по мере надобности, но на данном этапе главная директива препроцессора, с которой будут начинаться все наши программы, — это директива включения `#include`.

Создание простой программы

C-программа имеет ярко выраженный модульный характер. Ее головная функция состоит из множества обращений к функциям, выполняющим те или иные действия, и отличается от прочих только тем, что имеет фиксированное имя `main`.

Листинг 1.1. Простейшая C-программа

```
#include <iostream.h>                // Включение библиотеки
main()
{ int i;                             /* Описание целой переменной*/
       for (i = 0; i < 10; i++)      // Оператор цикла
              cout << i << endl;     // Вывод числа
}
```

Здесь после двойного слэша `//` помещен комментарий, который занимает остаток строки и игнорируется компилятором. Можно также располагать комментарий между парами ограничителей `/*...*/`, в этом случае он может и не ограничиваться одной строкой.

Не вдаваясь пока в детали, прокомментируем приведенный текст программы:

1. Первая строка — подключение файла `iostream.h`, расположенного в стандартном каталоге `include` среды разработки (на это указывают угловые скобки `< >`), который должен быть в ней прописан, что обычно происходит при инсталляции компилятора. Этот файл необходим для того, чтобы

можно было воспользоваться оператором потокового вывода `cout <<`. Вообще-то `cout` — это экземпляр класса `ostream`, но пока не будем углубляться, отметим только, что это работает.

2. `main()` — имя головной функции. Именно ей, согласно стандарту языка, передается управление при запуске программы. В С-программе должна обязательно присутствовать функция `main()`.
3. Тело функции располагается между открывающей и закрывающей скобками `{ }`. Надо иметь в виду, что компилятор очень скрупулезно подсчитывает, сколько скобок открыто и сколько закрыто — если скобку забыли закрыть, он выдает сообщение об ошибке.
4. `int i;` — описываем переменную для хранения целого числа.
5. `for (i = 0; i < 10; i++)` — оператор цикла, обеспечивает выполнение оператора вывода на консоль 10 раз, изменяя значение переменной `i` от 0 до 9.
6. `cout << i << endl;` — вывод числа на экран монитора (консоль вывода), после чего производится переход в начало следующей строки `endl` *(end of line)*. Читаем `cout` как *console output*. Как видно из примера, оператор `cout <<` можно применять агрегатно и выводить сразу несколько значений, а тип выводимого значения оператор определит сам.

Для выполнения рассмотренного примера откроем среду программирования MS Visual C++ 6.0, запустив головной модуль msdev.exe. Пока будем работать с консольными приложениями.

1. Выполнив команду меню **File | New**, выберем **Win32 Console Application** в окне создания проекта New, открытом на вкладке **Projects** (рис. 1.1).
2. В поле **Location** введем вручную (либо воспользуемся поиском по кнопке **...**) имя рабочей папки. Например, `C:\WORK`.
3. В поле **Project name** введем имя нашего будущего проекта — `example1`, после чего нажмем кнопку **OK**.
4. Появляется диалоговое окно, приведенное на рис. 1.2, в котором мы оставим настройку по умолчанию — **An empty project** (Пустой проект).
5. Получаем диалоговое окно **New Project Information**. Писать программу пока негде, на данном этапе лишь создана папка для нового проекта `C:\WORK\example1`.
6. Выполнив команду меню **File | New**, выберем на вкладке **Files** пункт **C++ Source File** (рис. 1.3) и в поле **File name** введем имя файла, например `main`. (Проверьте, что перед полем **Add to project** выставлен флаг, иначе потом придется "вручную" добавлять файл к проекту.)

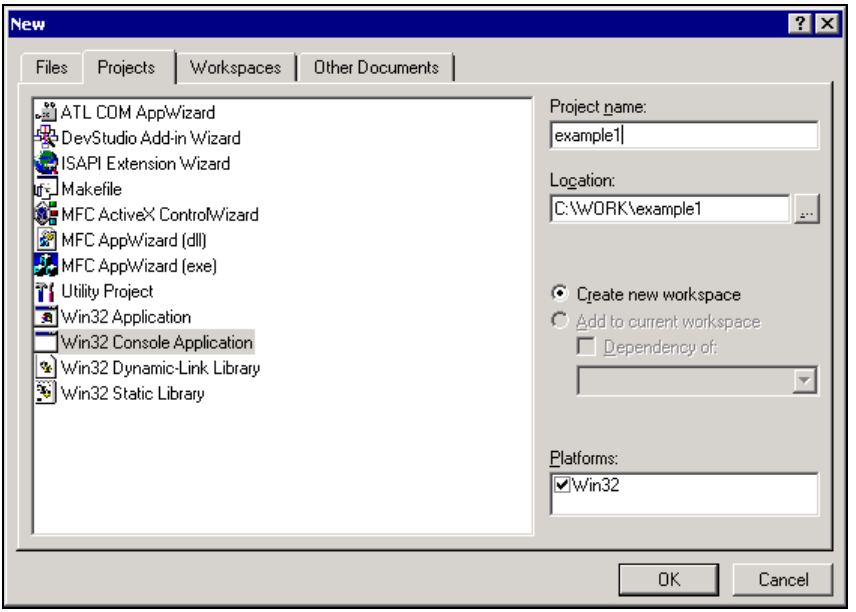

Рис. 1.1. Окно создания проекта

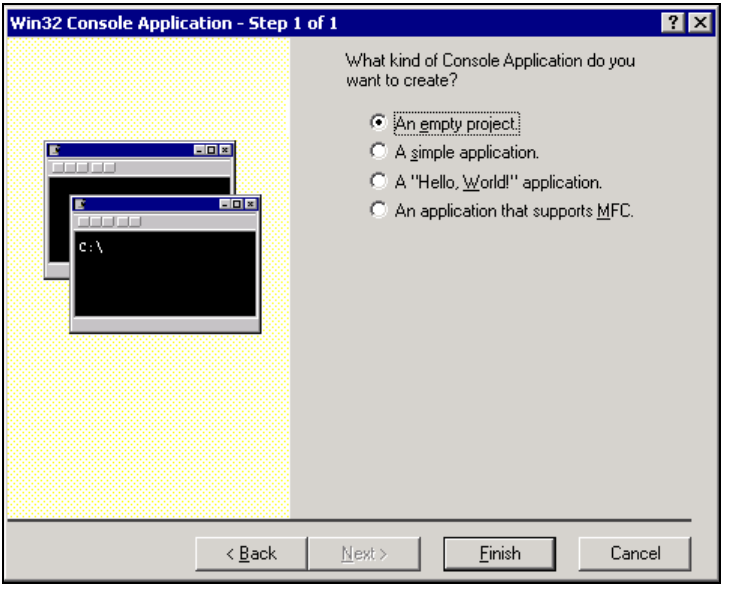

Рис. 1.2. Создание проекта Win32 Console Application. Шаг 1

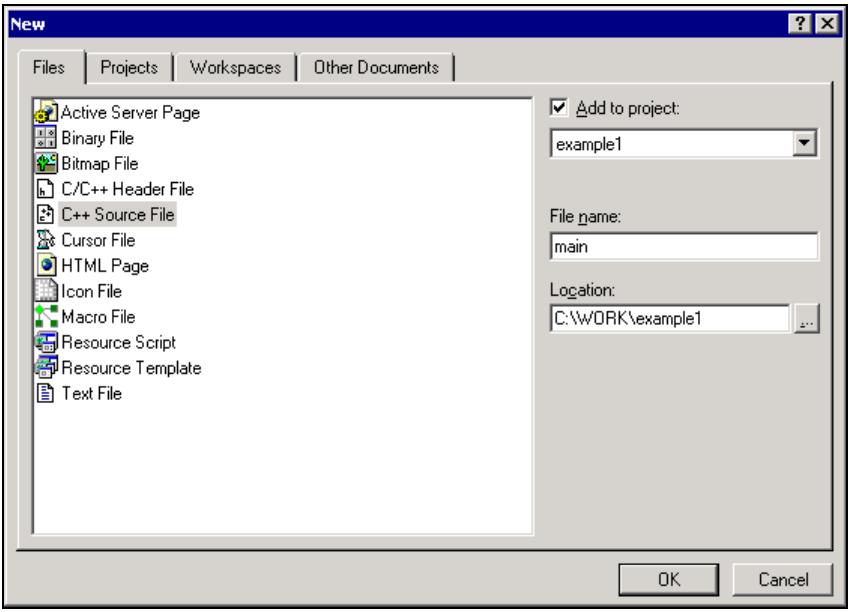

Рис. 1.3. Добавление файла к проекту

7. Введем текст программы и запустим на выполнение нажатием кнопки **!**. Это единственная кнопка со значком красного цвета. Получим результат, приведенный на рис. 1.4.

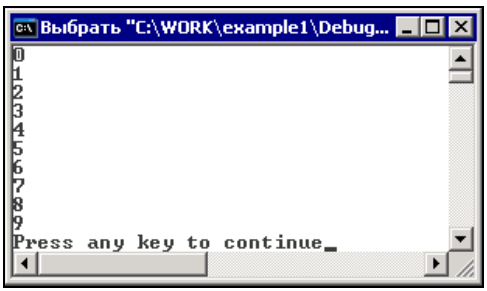

Рис. 1.4. Результат выполнения программы

Набирая текст этой программы, сложно допустить ошибку, но если это все-таки произошло, перейдите в окно вывода **Output** на вкладку **Build** и установите указатель на сообщение об ошибке. После двойного щелчка на нем компилятор отметит строку с ошибкой. Исправьте ошибку и повторите процедуру запуска программы. Если у вас есть навыки работы в текстовых редакторах, то ввод и редактирование текста программы не должны вызывать никаких проблем. Однако, поскольку это все-таки специализированный ре-

дактор, он имеет некоторые дополнительные возможности. Например, если в тексте программы установить курсор рядом со скобкой либо выделить ее, можно использовать одну из следующих полезных команд: поиск парной скобки и выделение блока текста, заключенного в скобки.

Полный список команд редактора можно посмотреть, выполнив команду меню **Help | Keyboard Map**.

Типы данных. Описание переменных

Вначале рассмотрим подробнее, как хранятся переменные различных типов в памяти. Адресуемая единица памяти, *байт*, состоит из 8 разрядов *(бит)*, куда и записывается число в двоичном коде, например:

$00010101 = 2^4 + 2^2 + 2^0 = 16 + 4 + 1 = 21$.

Примечание

Двоичная система счисления — позиционная система с основанием 2.

Правила двоичной арифметики очень просты (табл. 1.1).

Таблица 1.1. *Двоичная арифметика*

`0 + 0 = 0`	`0 * 0 = 0`
`0 + 1 = 1`	`0 * 1 = 0`
`1 + 1 = 10`	`1 * 1 = 1`

В один байт может быть записан код от 0 до 255 (2^8–1).

Таблица 1.2. *Перевод двоичных чисел в восьми-, десяти- и шестнадцатеричное представление*

2	16	8	10	2	16	8	10
0000	0	0	0	1000	8	10	8
0001	1	1	1	1001	9	11	9
0010	2	2	2	1010	A	12	10
0011	3	3	3	1011	B	13	11
0100	4	4	4	1100	C	14	12
0101	5	5	5	1101	D	15	13
0110	6	6	6	1110	E	16	14
0111	7	7	7	1111	F	17	15

Пользуясь табл. 1.2, можно очень просто представить двоичное число в виде его шестнадцатеричного аналога. На самом деле, числа записывают в двоичной форме очень редко, поскольку тогда они занимают много разрядов. Обычно двоичное число записывают более компактно, используя шестнадцатеричную форму записи, пример которой приведен на рис. 1.5.

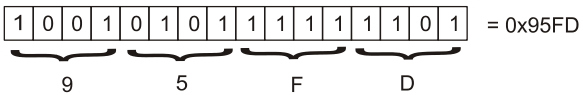

Рис. 1.5. Преобразование двоичного числа в шестнадцатеричное представление

Нетрудно было разбить это число на тетрады и записать, сверившись с таблицей кодов, результат. Обратное преобразование также не вызывает сложности, достаточно вместо каждой шестнадцатеричной цифры записать соответствующую двоичную тетраду.

Размер выделенной памяти под переменные различных типов и диапазон их изменения приведен в табл. 1.3—1.4.

Таблица 1.3. *Размер типа данных Visual C++*

Тип	Размер в битах	Диапазон
`bool`	8	true, false
`char`	8	–128 .. 127
`short`	16	–32768 .. 32767
`int, long`	32	–2147483648 .. 2147483647
`float`	32	3.4E-38 .. 3.4E+38
`double`	64	1.7E-308 .. 1.7E+308
`long double`	80	3.4E-4932 .. 1.1E+4932

Примечание

- Тип `bool` отсутствовал в классическом C и появился лишь в C++.
- Для 32-битных приложений тип `int` совпадает с `long` (полная запись `long int`).
- Целые типы данных могут иметь модификатор `unsigned` (беззнаковый). Диапазон таких переменных отсчитывается от 0.

Таблица 1.4. *Размер целых беззнаковых типов*

Тип	Размер в битах	Диапазон
`unsigned char`	8	0 .. 255
`unsigned short`	16	0 .. 65535
`unsigned int, long`	32	0 .. 4294967295

Рассмотрим подробнее, как в памяти представлена переменная типа `short`. Например, константа 1 в двоичном представлении имеет вид:

```
0000 0000 0000 0001.
```

Чтобы получить число 2, нужно прибавить к 1 еще 1 и т. д., в результате получаются все целые числа.

```
    + 0000 0000 0000 0001
      0000 0000 0000 0001
    = 0000 0000 0000 0010
```

Отрицательные числа для процессора 80×86 представлены в дополнительном коде и образуются инверсией числа с добавлением 1 (при инвертировании кода 1 заменяется на 0, а 0 на 1), например –1 получается по следующей схеме:

```
    + 1111 1111 1111 1110        + ~1
      0000 0000 0000 0001          1
    = 1111 1111 1111 1111        = -1
```

Такой способ образования отрицательных чисел позволяет свести операцию вычитания к операции сложения с отрицательным числом. Таким образом, операция вычитания является излишней для системы команд процессора и может быть исключена. Пример: 1 – 1 = 1 + (–1) = 0.

```
    + 0000 0000 0000 0001
      1111 1111 1111 1111
    =  0000  0000  0000  0000
```

C++ относится к таким языкам программирования, в которых все переменные, используемые программой, должны быть предварительно описаны. В описании переменной задается ее класс памяти и тип для того, чтобы транслятор выделил нужное количество памяти и в нужном разделе. Для обозначения переменной служит имя — символическое обозначение переменной. Начинается имя с буквы или символа подчеркивания "_", может содер-

жать буквы латинского алфавита, цифры и знак подчеркивания. Длина имени может быть произвольной, но распознается по первым 32 символам (это зависит от настроек компилятора). Строчные и прописные буквы различаются компилятором как разные буквы.

Переменные могут быть описаны в любом месте текста программы, но обязательно до их использования. Имена переменных в операторах описания отделяются запятыми. Пример:

```
int i,j,k;
```

Возможна инициализация переменных при описании, например:

```
int i = 0;
```

Переменные целого типа: *bool*, *char*, *short*, *int*, *long*

При описании переменной целого типа необходимо учитывать диапазон ее изменения. Если переменная может принимать одно из двух значений (`true`, `false`), используется тип `bool`. Если же необходим больший диапазон выбора значений, можно использовать переменную одного из следующих типов: `char` (от –127 до +127), `short` (от –32 768 до +32 767) или `int` (от –2 147 483 648 до +2 147 483 647). Тип `long` совпадает с типом `int`. Однако для переменных целого типа необходимо иметь в виду, что при выполнении арифметических операций процессор разворачивает число до длины машинного слова (для 32-битных приложений удвоенное машинное слово — 4 байта), т. е. никакой экономии в скорости выполнения операций не достигается при использовании типа меньшей размерности, так что выбор типа данных определяется лишь экономией памяти при хранении данных.

Если заведомо известно, что переменная принимает только положительные значения, целесообразно использовать модификатор `unsigned`, который обеспечит дополнительный контроль со стороны компилятора на присваивание отрицательного значения и увеличит вдвое диапазон положительных чисел.

enum — перечисляемый тип

Перечисляемый тип используется, как правило, для описания символических констант. Переменным, перечисленным в списке `enum`, присваиваются последовательно целые значения, начиная с 0. Пример:

```
enum { BLACK, BLUE };
```

Здесь описан неименованный объект перечисляемого типа, в котором определены две символические константы: `BLACK = 0` и `BLUE = 1`.

Можно, однако, описать переменную и присваивать значение в списке `enum` явно, например:

```
enum io_state { goodbit = 0x00,
                eofbit  = 0x01,
                failbit = 0x02,
                badbit  = 0x04 };
```

В данном контексте переменная `io_state` может принимать только значения, указанные в перечислении.

Переменные с плавающей точкой: *float, double, long double*

Внутреннее представление переменных с плавающей точкой более сложно. Число представляется в виде: $0.m*2^p$, где первый сомножитель *(мантисса)* лежит в диапазоне от 0.5 до 1, а порядок `p`, чтобы не вводить знаковый разряд, записывается с дополнением.

Основные типы переменных с плавающей точкой

1. `float` занимает 4-байтную область памяти, которая делится на три поля:
   - Знаковый разряд — 1 бит.
   - Порядок с дополнением +127 — 8 бит.
   - Мантисса — 23 бита.

1	8	23
знак	порядок+127	мантисса

Так, например, число $0{,}75*2^2$ может быть представлено в виде:

знак + = 0, порядок p = 127+2=129, мантисса m = 0.75= ½ + ¼.

На самом деле ситуация еще интересней. Если мантисса больше 0.5, то первый разряд мантиссы будет всегда заполнен, а если это так, то нет никакого смысла хранить этот разряд в памяти. Так и происходит, первый разряд мантиссы при записи числа исключается, а перед операцией процессор автоматически добавляет этот разряд (½ не отображается).

0	10000001	10000000000000000000000

Примечание

С этим связано разночтение у некоторых авторов. Иногда пишут, что мантисса лежит в диапазоне от 0 до 0.5.

2. `double` занимает 8-байтную область, которая также делится на три поля.
   - Знаковый разряд — 1 бит.
   - Порядок с дополнением +1023 — 11 бит.
   - Мантисса — 52 бита.

1	11	52
знак	порядок+1023	мантисса

   Особенность типа `double` заключается лишь в увеличении диапазона для порядка до 11 разрядов и 52 разряда под мантиссу, что позволяет увеличить точность записи числа. Так, если для типа `float` числа записываются с точностью до 5—6 значащих цифр, то для `double` — с точностью до 15—16 значащих цифр.

3. `long double` имеет расширенный диапазон `3.4e-4932 .. 1.1e+4932` и занимает 10-байтную область. Этот тип используется редко.

   Своеобразие арифметики с плавающей точкой в языке C++ заключается в том, что для повышения точности вычислений компилятор для переменных типа `float` автоматически ставит преобразование в `double` перед операцией, а результат приводит вновь к типу `float`. Таким образом, использование типа `float` оправдано лишь экономией памяти при хранении данных, но приводит к "буйному" преобразованию типов, что вряд ли следует приветствовать. Поэтому мы в большинстве случаев будем использовать тип `double` для данных с плавающей точкой.

Целые константы

Целые константы могут быть записаны в десятичном, восьмеричном и шестнадцатеричном представлении. Отличительным признаком восьмеричной константы является первый символ "0", а шестнадцатеричной — "0x".

Пример:

- 2, 5, –18 — десятичные константы;
- 076, –0237 — восьмеричные константы;
- 0xf1, 0x4ea3 — шестнадцатеричные константы.

Знак "-" перед константой рассматривается как унарная операция "минус".

Примечание

Унарная операция имеет 1 операнд, а бинарная, соответственно, два.

По умолчанию целые константы конструируются компилятором типа `int`. Суффикс `u (U)` определяет модификацию типа `unsigned`, а `l (L)` — `long`.

Пример:

```
129u, 0xf0e1u;
2l, 0xffl;
```

Константы с плавающей точкой

Если константа содержит десятичную точку "`.`" или символ экспоненты "`e`", то она рассматривается как *константа с плавающей точкой* типа `double`. Константа типа `float` может быть получена добавлением суффикса `f`.

Пример:

```
1.2e-2, .85, 10., 1e-6f;
```

Константа `1.2e-2` интерпретируется как $1.2*10^{-2}$.

Символьные константы

Символьные константы состоят из символа, заключенного в одинарные кавычки, например: `'a'`, `'b'`, `'0'`. Константа является целым числом, равным коду символа. Допускается использование четырехсимвольных констант, таких как `'abcd'`, которые занимают удвоенное машинное слово (4 байта), при этом первый символ находится в младшем байте, а второй — в старшем и т. д.

Примечание

Зависит от размера типа `int`, в Turbo C допускаются 2-символьные константы.

Для ввода специальных символов в C++ зарезервированы управляющие символьные константы, перечисленные в табл. 1.5.

Таблица 1.5. *Управляющие символьные константы*

`\a`	Звуковой сигнал
`\b`	Возврат курсора на одну позицию
`\f`	Перевод страницы
`\n`	Перевод строки

Таблица 1.5 *(окончание)*

\a	Звуковой сигнал
\r	Возврат каретки
\t	Табуляция
\v	Вертикальная табуляция
\\	Обратный слэш
\'	Одинарная кавычка
\"	Двойная кавычка
\?	Знак вопроса
\bbb	Любой символ, где bbb — восьмеричное число
\xhh	Любой символ, где hh — шестнадцатеричное число

Строковые константы

Любая последовательность символов, заключенная в двойные кавычки, например `"текст"`, рассматривается как *текстовая (строковая) константа*. В C++ принято считать признаком конца текста символ `'\0'`, который добавляется автоматически. Таким образом, любая текстовая константа занимает в памяти на 1 байт больше количества символов. В C++ также предусмотрена неявная конкатенация, т. е. "склеивание" строк.

Так, для константы:

```
"строка 1" "строка 2";
```

компилятор построит следующую строку:

```
"строка 1строка 2".
```

Для переноса длинной строковой константы в тексте программы необходимо завершить переносимую строку символом "\".

Пример 1.

```
"Очень длинный текст, который хотелось бы \
         разместить в одной строке"
```

Пример 2.

```
#include <iostream.h>
main()
{
   int i, j = 6, k = -1;
   char h = '0', ch;
```

```
   double pi = 3.14;
   i = j + k;
   cout << "int i=j+k: " << i << endl;
   ch = h + 1;
   cout << h << '\t' << ch << endl;
   cout << "pi: " << pi << endl;
}
```

Первая строка, как обычно, подключает стандартную библиотеку потокового ввода/вывода. Далее в функции `main()` описываем переменные целого типа:

```
int i, j = 6, k = -1;
```

Причем первая переменная объявлена, но значение ей не присвоено (значением может быть любой "мусор"), второй переменной присвоено значение 6, а третьей –1.

```
char h = '0', ch; // Две переменных типа char.
double pi = 3.14; // Одна переменная double,
        // инициализированная значением 3.14
i = j + k;     // Обычная операция сложения, об этом чуть позже.
cout << "int i=j+k: " << i << endl;
```

Здесь выводим текстовую строку и переменную `i`. Отметим также, что оператор вывода `<<` корректно выведет данные различного типа. Пока воспринимаем это как данность, с потоковым выводом мы будем разбираться позже.

```
ch = h + 1;
```

А вот здесь интересная операция, к символу `'0'` прибавили 1. Ничего удивительного, наверное, нет в том, что в результате получится символ `'1'`, но, пожалуй, ни один другой язык программирования не позволит такой операции.

```
cout << h << '\t' << ch << endl;
```

Чтобы убедиться в этом, посмотрите результат вывода двух символов. Мы вставили между ними разделитель "горизонтальная табуляция" и завершили вывод специальным манипулятором `endl (end of line)`, который обеспечит перевод строки. Он описан в файле включений `iostream.h`.

```
cout << "pi: " << pi << endl;
```

И наконец, вывод числа с плавающей точкой.

Переменные-константы

Когда мы определяем в программе константу, мы, как правило, хотели бы быть уверены, что значение этой константы случайно не изменилось. Конеч-

но, по большей части такие проблемы остаются на совести программиста, но компилятор дает возможность обеспечить дополнительный контроль над изменением значений переменных. Для этого служит модификатор `const`, который ставится перед указанием типа переменной. Например, можно было бы написать:

```
const double pi = 3.14;
const int NULL = 0;
```

Теперь, при попытке изменить значение константной переменной, будет выдано сообщение об ошибке.

```
NULL = 1;      // Ошибка
```

Комментарии

Для включения комментария в текст программы на языке C++ предусмотрены две конструкции:

- Если *комментарий* занимает одну строку, то достаточно поставить два символа `//` (двойной слэш) и текст, расположенный далее до конца строки, будет считаться комментарием и игнорироваться компилятором.
- Если же комментарий занимает несколько программных строк, он ограничивается с обеих сторон парой символов `/* . . . */`.

```
// Простой комментарий
/*Комментарий,
        занимающий
        несколько строк */
```

Классы памяти

Как правило, программиста не заботит проблема, в какой области памяти компилятор разместит переменные, но иногда возникает необходимость управлять этим процессом. По умолчанию компилятор строит переменные *класса памяти* `auto` — это автоматические переменные, которые создаются в стеке при входе в функцию и освобождаются при выходе из нее, причем начальные значения таких переменных могут быть произвольными. Переменные класса `static` располагаются в фиксированном блоке памяти, выделяемом перед началом выполнения программы, инициализируются нулевым значением. Время жизни таких переменных совпадает с временем жизни программы. И наконец, для переменных целого типа, определен весьма специфичный класс `register`. В этом случае компилятор пытается выделить свободный регистр, и если это удается, операции с такой переменной будут осу-

ществляться существенно быстрее, если же свободного регистра нет, строится обычная переменная класса `auto`.

```
static const int NULL = 0;
register int k;
```

Операторы и операции

Основной *операцией* любого языка программирования является *операция присваивания*:

операнд = выражение;

```
i = j + k;
```

В отличие от других языков программирования, в C++ допускается многократное присваивание, причем операция выполняется справа налево, например:

```
i = j = k = 0;
```

или так:

```
i = 2 + (k = 3);
```

Во втором случае переменная `i` принимает значение `5` и, попутно, переменная `k` принимает значение `3`, т. к. в языке C++ принято, что скобка имеет значение последней выполняемой в ней операции.

Арифметические операции

В языке имеется набор обычных математических операций, выполнение которых осуществляется в соответствии с общепринятыми правилами соблюдения приоритетов. Причем операции одного приоритета, например умножение и деление, выполняются слева направо.

- `*` — умножение
- `/` — деление
- `%` — остаток от деления (для целых)
- `+` — сложение
- `-` — вычитание

Примечание

- Операция `%` корректно работает с положительными значениями, но для отрицательных чисел ее результат не очевиден и зависит от компилятора.

- Необходимо иметь в виду, что в языке C++ отсутствует операция возведения в степень, однако имеется функция, реализующая данную операцию.

Приведение типов

При выполнении арифметических операций результат зависит от типов операндов. Так, если операнды целого типа, то и результат будет целого типа. Однако часто возникает ситуация, когда операнды принадлежат разным типам. В этом случае работает правило *приведения типа*, согласно которому типы операндов приводятся к более общему типу.

Примечание

При совершении арифметической операции с переменными типа `char` и `short`, они разворачиваются до базового типа `int`.

Так, в выражении:

```
double avg, sum;
int n;
. . .
avg = sum/n;
double num = n;
```

переменная типа `double` делится на переменную типа `int`. Здесь компилятор вставит *автоматическое преобразование типа* `int` в `double`, который является более общим типом. То же самое произойдет и в следующей операции присваивания.

Однако часто бывает необходимо *явно преобразовать тип данных*, например, если мы хотим получить результат от деления двух целых чисел:

```
int a = 3, b = 2;
double r = a/b;
```

то мы получим `r = 1.0` вместо `1.5`. Дело в том, что вначале происходит деление двух целых чисел, результат которого `1`, и лишь затем это значение будет приведено к типу `double`. Для того чтобы деление произошло над данными типа `double`, необходимо хотя бы один из операндов привести к типу `double`. Для этого нужно воспользоваться оператором явного преобразования типа одним из трех способов:

- `(double)a` — базовый оператор классического C.
- `double(a)` — расширение языка, преобразование типа как функция.
- `static_cast<double>(a)` — современный стиль.

Если мы напишем выражение:

```
double r = double(a)/b;
```

получим: `r = 1.5;`

Операции ++ и --

Специфичные для языка C++ *операции приращения* для переменной целого типа `char`, `short`, `int`, `long` приводят к увеличению `++ (increment)` или уменьшению `-- (decrement)` значения переменной на `1`. Если данная операция стоит перед переменной, то она будет произведена перед использованием переменной *(префиксная операция)*, если после переменной — после ее использования *(постфиксная операция)*.

Именно наличием этих операций объясняется название языка C++ (следующий шаг после C). Эти операции сводятся к одной машинной команде и выполняются с большей эффективностью, чем операции сложения и вычитания.

```
i = 0;
j = ++i;       // j = 1, i = 1
k = i--;       // k = 1, i = 0
```

Битовые операции

Осуществляются над переменными, рассматриваемыми как битовые цепочки, и применяются для целого типа `char`, `short`, `int`, `long`. *Битовая операция* производится над каждым разрядом без переносов.

Таблица 1.6. *Битовые операции*

`&`	Битовое умножение
	Пример: (10010011) & (00111101) = (00010001)
		`1 & 1 = 1`
		`0 & 1 = 0`
		`0 & 0 = 0`

`|`	Битовое сложение (ИЛИ)
	Пример: `(01101110) | (10000000) = (11101110)`
		`1 | 1 = 1`
		`0 | 1 = 1`
		`0 | 0 = 0`

`^`	Битовое исключающее сложение (исключающее ИЛИ)
	Пример: `(01010101) ^ (11111111) = (10101010)`
		`1 ^ 1 = 0`
		`0 ^ 1 = 1`
		`0 ^ 0 = 0`

`~`	Битовое отрицание (НЕ) Пример: `~(10011010) = (01100101)`	`~1 = 0` `~0 = 1`

Таблица 1.6 *(окончание)*

<<	Логический сдвиг влево Все разряды переменной сдвигаются на указанное количество разрядов влево, разряды, выходящие за разрядную сетку, теряются, а освобождающиеся разряды справа устанавливаются в 0. Пример: `(00000001) << 4 = (00010000)`	
>>	Логический сдвиг вправо Все разряды переменной сдвигаются на указанное количество разрядов вправо, разряды, выходящие за разрядную сетку, теряются, а освобождающиеся разряды слева устанавливаются в 0. Пример: `(00010000) >> 4 = (00000001)`	

Примеры использования битовых операций:

```
i << 1;        // умножить на 2
i >> 2;        // поделить на 4
i & 1;         // проверка на нечетность
i | 128;       // установить в 1 7-й бит
i ^ 128;       // инвертировать 7-й бит
~i;            // инвертировать все биты
```

Примечание

Умножение и деление нацело на число, кратное 2, при помощи операций сдвига корректно только для положительных значений.

Комбинированные операции

Часто возникает необходимость осуществлять операции с одной переменной в левой и правой части операции присваивания. В этом случае удобно использовать так называемые *комбинированные операции*. Например, вместо операции `i = i + 2` можно написать `i += 2`, что имеет такой же смысл, но облегчает компилятору построение более эффективного программного кода.

Приведем список комбинированных операций:

```
i += j;        // i = i + j;
i -= j;        // i = i - j;
i *= j;        // i = i * j;
i /= j;        // i = i / j;
i %= j;        // i = i % j;
i <<=j;        // i = i <<j;
```

```
i >>=j;        // i = i >>j;
i &= j;        // i = i & j;
i |= j;        // i = i | j;
i ^= j;        // i = i ^ j;
```

Операции отношения

Для сравнения значений переменных используются *операции отношения*:

```
>   больше               (a > b)
>=  больше или равно     (a >= b)
<   меньше               (i < 0)
<=  меньше или равно     (i <= j)
==  равно                (i == k)
!=  не равно             (ch != 'y')
```

Примечание

Равенство выполнено двумя знаками равно. Типичная ошибка начинающего программиста, когда вместо равенства записывается операция присваивания =.

В классическом C не было логических переменных, они появились лишь в C++. Однако и при отсутствии логических переменных язык C обходился без больших проблем. Было принято считать, что результат имеет значение ИСТИНА, если он отличен от 0, и ЛОЖЬ в противном случае. В C++ можно использовать как логические переменные, так и подход классического C, когда вместо логических выражений применяются арифметические значения. Например, для проверки нечетности целого значения можно воспользоваться выражением `(i & 1) != 0` или просто `(i & 1)`.

В первом случае получаем логическое выражение, равное `true`, если переменная `i` равна нечетному числу, во втором случае получаем просто `1`, что и так ИСТИНА. Вообще, в языке C++ считается дурным тоном сравнивать с `0`.

Логические операторы

Логические операторы применяются над логическими операндами, в качестве которых могут выступать операции отношения или те же логические операторы (табл. 1.7).

Таблица 1.7. *Логические операторы*

&&	И	(i > j) && (k != 1)
||	ИЛИ	(ch == 'y') || (ch == 'Y')
!	НЕ	!(i > 1)

Примечание

Обычно компилятор оптимизирует код операции логического умножения, и если первый операнд имеет значение `false`, то и результат операции `&&` также будет `false`, так что второе выражение можно и не вычислять.

С помощью логических операторов можно конструировать сложные логические выражения. Нужно иметь в виду, что операторы языка выполняются в соответствии с их приоритетами, таблица приоритетов помещена в *приложении*, но если есть сомнения в порядке выполнения операций, можно использовать скобки.

Операторы языка C++

Оператор может быть простым либо составным. *Простой оператор* заканчивается символом точка с запятой `";"` — частью оператора, отличающейся C++ от других языков программирования, где символ `";"` — разделитель операторов. *Составной оператор* представляет собой последовательность простых операторов, заключенную в фигурные скобки.

Каждый оператор может иметь метку, которая используется оператором перехода `goto`. *Метка* — это имя, за которым стоит двоеточие `":"`. Областью действия метки является текущая функция. В C++ метки предварительно не описываются.

```
{ int i;
  home: i = 0;
  . . .
}
```

Оператор перехода *goto*

Оператор перехода служит для безусловной передачи управления оператору с данной меткой в пределах текущей функции:

```
goto метка;
```

Хорошо структурированная программа не должна вызывать необходимости использования оператора перехода, поэтому в современном стиле программирования считается дурным тоном использование операторов `goto`. Однако они могут быть весьма полезны для обработки критических ситуаций или выхода из глубоко вложенных циклов.

Примечание

Сейчас для обработки критической ситуации используется механизм исключений.