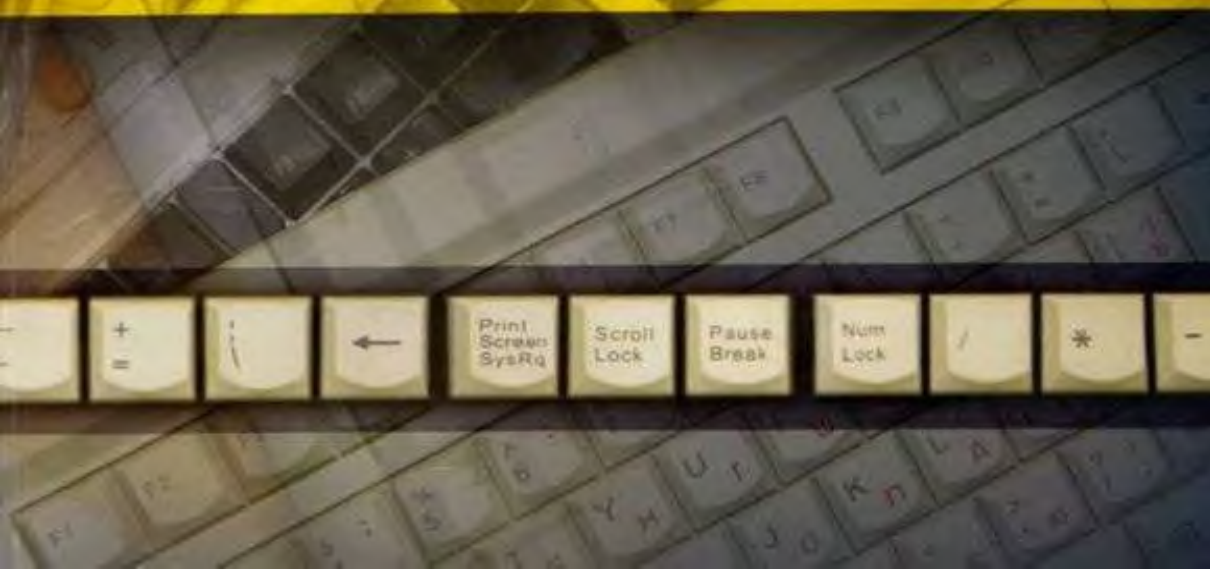
В.М.Дёмкин

ИМПЕРАТИВНОЕ ПРОГРАММИРОВАНИЕ

в примерах на C++



ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

В.М.Дёмкин

ИМПЕРАТИВНОЕ ПРОГРАММИРОВАНИЕ **в примерах на C++**

Практикум

Издательский дом ГУ ВШЭ
Москва 2007

Предисловие

Предлагаемый вниманию читателя практикум по императивному программированию на языке C++ продолжает собой серию методических материалов, предназначенных для формирования у студентов младших курсов направлений “Бизнес-информатика” и “Прикладная математика и информатика” целостного взгляда на информатику. Автор на протяжении многих лет не без успеха последовательно отстаивает свою позицию в том, что реальным помощником в реализации этой цели могут служить отдельные учебные пособия и практикумы по основным разделам курса “Информатика”, связанные единой формой изложения. Так, уже вышли в свет первые учебные пособия по основам алгоритмизации и императивного программирования в примерах на Turbo Pascal, по основам объектно- ориентированного программирования в примерах на C++, а следом за ними и первый практикум по объектно-ориентированному программированию на языке C++. Прежде всего, сама идея издания подобной литературы связана со стремлением автора рассматривать информатику под углом зрения проектирования алгоритмов и абстрактных типов данных, придерживаясь рамок требуемой для решения задачи парадигмы программирования (например, императивной, объектно-ориентированной или обобщенной) при помощи таких языков программирования для персональных компьютеров, как Fortran, Basic, Turbo Pascal, C, C++ и Java.

Излагаемый материал практикума соответствует разделам учебного пособия “Основы алгоритмизации и императивного программирования в примерах на C++” и является одной из трех составных частей лабораторного практикума учебной дисциплины “Методы программирования”. Программный код прошел апробацию на современных компиляторах платформ Windows и Linux.

Изучение учебного материала практикума предполагает знакомство читателя с императивной и модульной парадигмами программирования для языка C++.

Язык программирования C++ является одним из представителей семейства гибридных языков, он поддерживает четыре парадигмы программирования: императивную, модульную, объектно-ориентированную и обобщенную.

C++ был создан Бьярном Страуструпом (Bjame Stroustrup) в начале 1980-х годов. Перед Страуструпом стояли тогда две основные задачи: во-первых, сделать C++ совместимым со стандартным языком C, во-вторых, расширить C конструкциями объектно-ориентированного программирования, основанными на концепции классов из языка Simula 67. Язык C был создан Дэннисом Ритчи (Dennis M.Ritchie) как язык системного программирования в начале 1970-х годов. История C связана с разработкой операционной системы Unix для мини-компьютера DEC PDP-11.

C++ развивался не только благодаря идеям, взятых из других языков программирования, но и в соответствии с запросами и опытом большого числа пользователей различного уровня, использующих C++ в самых разнообразных прикладных областях. В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882). Стандартный C++ и его стандартная библиотека до сих пор остаются одним из основных инструментов для разработки приложений, предназначенных для широкого диапазона использования.

Первое знакомство с программой C++

Программирование в стиле C

> Пример 1

Редкая книга по программированию на C или C++ обходится без первой учебной программы, в которой не была бы представлена операция вывода строкового литерала ***''Hello, world!\n".*** Отдавая дань этой традиции и придерживаясь рамок императивной парадигмы, попытаемся на первых порах проиллюстрировать с помощью такого рода программ стили программирования на C++.

Известно, что C++ создан на основе языка программирования C, и C остается подмножеством C++. Зачастую эту основу называют C-подмножеством, поэтому и первые программы поначалу представим для C-подмножества C++:

```
/* Программирование в стиле C */
#include <stdio.h> int main()
{
printf("Hello, world!\n");
return 0;
}
```

Результат работы программы:

```
Hello, world!
```

Как увидим далее, здесь есть почти все, что так характерно для большинства программ на C или C++.

Во-первых, это многострочный комментарий, заимствованный от C. Признаком многострочного комментария являются два его ограничителя: началом комментария служит последовательность символов /*, а соответственно его концом -*/. Следует привыкнуть к тому, что комментарий должен быть непременным атрибутом любой программы, так как он является одним из средств улучшения понимания ее кода.

Во-вторых, это директива препроцессора C ***Mnclude*** для включения заголовочного файла **<*stdio.h*>** стандартной библиотеки в исходный файл с исполняемым кодом. Отметим здесь, что угловые скобки в именовании заголовочных файлов указывают компилятору на стандартное размещение этих файлов в системных директориях.

Напомним, что в заголовочном файле **<*stdio.h*>** представлены объявления функций ввода-вывода. Напомним также, что объявление функции в C или C++ зачастую называют ее прототипом. В частности, в заголовочном файле **<*stdio.h*>** представлен прототип используемой здесь библиотечной функции форматного вывода ***printft***):

Intprintf(const char*, ...);

Одной из характерных особенностей объявления такого рода функций стандартной библиотеки является неуказанное количество их аргументов. Первым аргументом вызова является указатель на константную форматную строку, которая определяет различные виды преобразования, форматирования и вывода в поток очередных аргументов вызова. Отметим, что библиотечные функции форматного ввода-вывода реализуют простую модель текстового ввода-вывода — поток символов. Текстовый поток состоит из последовательности строк, строка заканчивается символом новой строки (ASCII-код 12).

Наряду с заголовочными файлами стандартной библиотеки пользователь вправе воспользоваться и своими собственными, которые обычно располагаются в текущей директории. Далее будет сказано, что для указания компилятору на размещение заголовочных файлов в какой-либо директории, отличной от стандартной, в именовании этих файлов вместо угловых скобок используются кавычки. Такой способ именования заголовочных файлов пользователя, которые, как правило, представляют собой интерфейсы модулей, необходим для поддержки модульной парадигмы программирования, когда программа собирается из почти независимых частей. Далее также будет сказано, как следует разрабатывать программу из набора модулей и когда следует использовать раздельную компиляцию модулей.

В-третьих, это функция ***main(),*** тело которой — это составная-инструкция или блок. Блок содержит две инструкции. Первая инструкция — это инструкция-выражение, с ее помощью реализуется вызов функции ***printj***0 с одним аргументом вызова (строковый литерал ***''Hello, world!\n").*** Одним из символов строкового литерала здесь является езсаре-последовательность ***\n*** — символ новой строки (от слов ***new line),*** благодаря которому и осуществляется переход на новую строку. Напомним, что последним символом строкового литерала в соответствии с соглашением для С-строк является так называемый нулевой символ — символьный литерал ***'\0',*** по которому как раз и определяется конец строки. Напомним также, что при выводе строковых литералов, если только в них нет символа ***%,*** по умолчанию используется форматная строка со спецификацией преобразования ***%s*** (от слова ***string).*** Обычно форматная строка указывается явно в качестве первого аргумента библиотечных функций форматного ввода-вывода. Далее будет сказано, что в форматной строке каждая спецификация преобразования начинается с символа ***%*** и заканчивается символом-спецификатором, а между символом ***%*** и символом-спецификатором может стоять один из элементов управления форматом, составляющих широкий набор форматных спецификаций. Также будет сказано, что помимо спецификаций преобразования форматная строка может содержать символы, которые просто копируются в текстовый поток вывода.

Например, строковый литерал ***''Hello, world!\n"*** можно вывести еще и так:

```
printf("%s", "Hello, world!\n");
```
или
```
printf("%s\n", "Hello, world!");
```

Здесь спецификация преобразования ***%s*** используется для явного указания вывода строкового литерала в консольный поток, ***s*** является символом-спецификатором, а символ ***\n*** из форматной строки просто копируется в текстовый поток вывода.

Вторая инструкция принадлежит к разновидности инструкций-перехода, благодаря которой реализуется механизм возврата результирующего значения функции ***main().***

> **Пример 2**

Если программе на C или C++ не требуется передавать какие-либо параметры при ее запуске, то функция ***main(),*** как правило, определяется с пустой сигнатурой, в противном случае функция ***main()*** определяется с двумя аргументами, которые как раз и необходимы ей для обработки командной строки. Заметим, что в силу причин, уходящих корнями глубоко в предысторию языка C, имена этих аргументов стали таким же незыблемым оплотом традиции, как и само приветствие ***''Hello, world!".*** Первый аргумент ***argc*** (от слов ***argument counter)*** представляет собой количество передаваемых параметров командной строки. Так как каждый параметр командной строки, в свою очередь, представляет собой тоже символьную строку, то второй аргумент ***argv*** (от слов ***argument values)*** обычно объявляется как массив указателей на ***char*** и за редким исключением как указатель на указатель на ***char.***

Представим программу, где с помощью инструкции-итерации проиллюстрируем механизм визуализации параметров командной строки для платформы Windows:

```
/* Программирование в стиле C */
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("%d\n", argc);
for (int i = 0; i < argc; ++i) printf("%s\n", argv[i]);
return 0;
}
```

Результат работы программы для платформы Windows:

```
1
probleml_02.exe
```

Как видим, обычный запуск такой программы приводит к протоколу, где единственным параметром, передаваемым функции ***main()*** является собственно само имя загружаемого файла (например, ***problem 1_ 02. exe).***

Запуск этой программы, например, с двумя параметрами командной строки

```
F:\C++\Unit l>probleml_02.exe Hello, world!
```

приведет к такому протоколу:

```
3
probleml_02.exe
Hello,
world!
```

Отметим, что помимо уже известной спецификации преобразования ***%s*** здесь используется также спецификация преобразования ***%d*** (от слова ***decimal)*** для указания преобразования целочисленного аргумента в десятичный вид и вывода его в текстовый поток.

> Пример 3

Памятуя о “традиции” объявлять аргумент ***argv*** функции ***main()*** как массив указателей на ***char***, обратимся теперь к программе, где аргумент ***argv*** объявляется как указатель на указатель на ***char.***

```
/* Программирование в стиле C */
#include <stdio.h>
int main(int argc, char** argv)
{
  printf("%d\n", argc);
for (int i = 0; i < argc; ++i) printf("%s\n”, argv[i]);
return 0;
}
```

Результат работы программы для платформы Windows:

```
1
probleml_03.exe
```

Как и следовало ожидать, обычный запуск такой программы и здесь приводит к протоколу, где единственным параметром, передаваемым функции ***main(),*** является собственно само имя загружаемого файла (например***, probleml_03.exe).***

Далее будет сказано, что для разыменования указателей, т.е. для доступа к тем объектам программы, на которые они как раз и указывают, равноправно могут быть использованы как оператор разыменования *, так и оператор индексации [].

Если аргумент ***argv*** функции ***main()*** объявляется как массив указателей на ***char,*** то доступ к i-му элементу массива (указателю на ***char)*** возможен двумя способами: либо с помощью индексной арифметики ***argv[i],*** либо с помощью адресной арифметики ****(argv+i)*** в соответствии с соглашением, что имя массива является указателем на его первый элемент.

Если аргумент ***argv*** функции ***main()*** объявляется как указатель на указатель на ***char,*** то доступ к i-му указателю на ***char*** так же возможен двумя способами: либо с помощью адресной арифметики ****(argv+i),*** либо с помощью индексной арифметики ***argv[i].*** Остается лишь отметить, что компилятор индексную арифметику всегда заменит адресной.

Программирование в стиле C++

> Пример 4

Получив с помощью первых программ для С-подмножества С++ представление о стиле С, теперь можно шаг за шагом приближаться к стилю С++. И первым шагом на этом пути станет объявление однострочного комментария, а также стандартного заголовочного файла.

Началом однострочного комментария в С++ является последовательность символов //. Зачастую таким комментарием завершаются строки с инструкциями.

Известно, что средства стандартных библиотек С и С++ представлены набором заголовочных файлов. Для стандартных заголовочных файлов в библиотеке С обязательным атрибутом является расширение ***.ft,*** в то же время для указания стандартных заголовочных файлов библиотеки С++ этого расширения не требуется. Однако отсутствие расширения ***.И*** вовсе не подразумевает какого-либо конкретного способа хранения заголовочных файлов, при этом даже не требуется, чтобы стандартные заголовочные файлы хранились обычным способом. Обработка стандартных заголовочных файлов и оптимизация реализации самой стандартной библиотеки определяются конкретной реализацией компилятора.

Отметим, что стандартный заголовочный файл в библиотеке С++, имя которого начинается с буквы ***c*** (например, ***<cstdio>),*** эквивалентен заголовочному файлу в стандартной библиотеке С (например, ***<stdio.h>).*** Отметим также, что средства стандартной библиотеки С++ определены в пространстве имен ***std,*** в то время как средства стандартной библиотеки С определены в глобальном пространстве имен.

Итак, сохраняя возможность работы с библиотечной функцией форматного вывода ***printf(),*** вначале представим программу, где стиль С++ пока будет применен только при объявлении комментария и заголовочного файла стандартной библиотеки:

```
// Программирование в стиле C++ — первый шаг #include <cstdio> int
main()
{
  printf("Hello, world!\n");
  return 0;
}
```

Результат работы программы:

```
Hello, world!
```

> Пример 5

По умолчанию каждая программа на C++ может пользоваться стандартными консольными потоками: стандартным вводом ***(tin)*** и стандартным выводом (***cout***), а также стандартным выводом ошибок (ce/т и ***clog).*** Эти потоки представляют собой последовательности символов (например, обычных ***char*** или расширенных ***wcharjt).***

Для использования стандартной библиотеки C++ не нужно знать методы, которые были применены при реализации библиотеки потоков. К тому же, эти приемы в разных реализациях различаются. Все, что сейчас необходимо знать, — это то, что стандартные потоки ввода-вывода ***tin, cout, cerr*** и ***clog*** определены в пространстве имен ***std*** и представлены заголовочным файлом ***<iostream>.***

Здесь также следует отметить, что для понимания основополагающих принципов организации стандартной библиотеки C++ наряду с представлением об императивной и модульной парадигмах программирования требуется также представление и о двух других парадигмах — объектно-ориентированной и обобщенной. Это необходимо для понимания таких сущностей, как класс, функция-член класса, операторная функция, функция-шаблон и класс-шаблон.

Одной из важных концепций, принятых при построении стандартной библиотеки потоков, является реализация операций ввода-вывода объектов встроенных типов в виде перегруженных стандартных операторов сдвига C++: оператора сдвига вправо >> (“прочесть из”) и оператора сдвига влево << (“записать в”). Перегруженные операторы сдвига >> и << представляют собой операторные функции классов ***istream*** и ***ostream*** и, соответственно, называются операторами ввода и вывода.

Итак, теперь следующим шагом на пути к стилю C++ станет использование потока стандартного вывода ***cout:***

```
// Программирование в стиле C++ — второй шаг #include <iostream> int
main ()
{
  std::cout << "Hello, world!\n";
  return 0;
}
```

Результат работы программы:

```
Hello, world!
```

Здесь операция вывода строкового литерала ***’’Hello, world!\n”*** осуществляется с помощью инструкции-выражения. Выражение составлено из бинарного оператора вывода << и двух его операндов — потока и строкового литерала. Чтобы указать на принадлежность имени потока ***cout*** пространству имен ***std,*** как видим, применяется механизм квалификации имени ***cout*** с помощью унарного оператора разрешения области видимости

> Пример 6

На этапе первого знакомства с библиотекой потоков C++, как правило, не требуется явного привлечения механизма форматирования ввода-вывода, достаточно положиться на умолчание. Отметим, что управление форматированием ввода-вывода производится средствами класса ***basicios*** и его базового класса ***iosbase*** (набор флагов формата и операций для их установки и сброса). Для управления состоянием потока также можно воспользоваться и более изящными средствами стандартной библиотеки — манипуляторами — набором функций для манипулирования состоянием потока. Ключевая идея заключается в том, чтобы вставлять между объектами (читаемыми из потока или записываемыми в поток) операцию, которая изменяла бы состояние потока. Стандартные манипуляторы определены в пространстве имен ***std*** и представлены заголовочными файлами **<*iostream*>** и **<*iomanip*>.**

Итак, следующим шагом на пути к стилю C++ станет использование стандартного манипулятора ***endl*** (от слов ***end of line),*** который выводит в поток символ новой строки 1л и очищает буфер потока:

```
// Программирование в стиле C++ — третий шаг #include <iostream> int
main ()
{
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

Результат работы программы:

```
Hello, world!
```

Ранее уже отмечалось, что оператор ввода >> и оператор вывода << представляют собой операторные функции классов ***istream*** и ***ostream.*** Так как возвращаемым значением этих функций является ссылка на ***istream*** или ***ostream,*** то это как раз и позволяет применять к ней друг за другом все последующие операторы ввода или вывода в инструкции-выражении, соблюдая при этом их естественный порядок следования. Таким образом, вставка манипулятора ***endl*** в поток ***cout*** следом за строковым литералом возможна благодаря лишь тому, что возвращаемым значением операторной функции ***operator*** <<0 является ссылка на ***ostream,*** что и позволяет применить к ней следующий оператор вывода.

> **Пример 7**

Одной из фундаментальных концепций, принятых при построении стандартной библиотеки C++, является пространство имен — область видимости (или действия).

В программе на C имеется единая глобальная область действия, что неизбежно порождает конфликты имен. С появлением пространства имен глобальная область действия стала всего лишь еще одним пространством имен. Особенность глобального пространства имен в C++ состоит только в том, что его имя не обязательно явно квалифицировать. Квалификация имени только с помощью унарного оператора разрешения области видимости :: означает, что это имя принадлежит глобальному пространству имен, а не тому, в котором такое же имя было повторно объявлено. Классы и обычные локальные области видимости тоже являются пространствами имен. Класс — это тип, определенный именованной локальной областью видимости, которая описывает, как создаются и используются объекты данного типа.

Пространство имен является механизмом определения области действия, обобщающего глобальные объявления в C и C++. Такие области действия разрешается именовать, а к их членам можно обращаться с помощью обычного для членов класса синтаксиса. Имена, объявленные внутри пространств имен, не конфликтуют ни с глобальными именами, ни с именами из других пространств имен. Наряду с пространством имен ***std,*** в котором определены средства стандартной библиотеки C++, пользователь вправе использовать и свои собственные пространства имен, поместив в них объявления необходимых ему средств. Далее будет сказано, как объявляются пространства имен и как их следует использовать для поддержки модульной парадигмы программирования.

С помощью ***using-объявлений*** имена из пространств имен можно добавлять к локальной области видимости, а ***using-директивы*** делают доступными имена из пространств имен в той области видимости, в которой они были указаны.

Итак, завершающим шагом на пути к стилю C++ станет иллюстрация механизма доступа к пространству имен ***std*** в локальной области видимости функции ***main(),*** основанного на использовании локальной ***using-директивы***:

```
// Программирование в стиле C++ — четвертый жат #include <iostream>
int main ()
{
  using namespace std;
  cout << "Hello, world!" << endl;
  return 0;
}
```

Результат работы программы:

```
Hello, world!
```

Управление форматированием вывода

Управление состоянием потока вывода — первый шаг

Вывод символьных литералов

> Пример 8

На этапе первого знакомства с фундаментальными типами C++ — символьными, целыми и с плавающей точкой — рассмотрим свойства различных механизмов управления форматированием вывода на примере таких объектов программы, как целые литералы (символьные, десятичные, восьмеричные и шестнадцатеричные), литералы с плавающей точкой и строковые литералы.

Вначале представим программу для иллюстрации форматного вывода символьных литералов из набора ASCII, которые относятся к фундаментальному типу C++ ***char***:

```
// C++ Символьные литералы #include <iostream> int main ()
{
  using namespace std;
  printf("%c%c%c%c%c%c%c",     'H', 'e' ,'1', '1', 'o', '!', '\n');
  printf("%c%c%c%c%c%c\n\n",     'H', 'e', 4', '1', 'o', '!');
  cout << 'H' << 'e' << '1' << '1' << 'o' << '!' << endl;
  return 0;
}
```

Результат работы программы:

```
Hello!
Hello!
Hello!
```

Как видим, при выводе символьных литералов с помощью библиотечной функции ***printj()*** используется форматная строка со спецификациями преобразования ***%c*** (от слова ***char),*** количество которых соответствует количеству элементов списка вывода.

Напомним, что форматная строка может содержать два типа объектов — символы, которые просто копируются в поток вывода, и спецификации преобразования, каждая из которых приводит к преобразованию и выводу в поток каждого очередного аргумента вызова. Каждая из спецификаций преобразования начинается с символа **%** и заканчивается символом-спецификатором.

Здесь, как и ранее, использование стандартного потока вывода ***cout*** представляет собой то, что обычно называют неформатированным выводом, поскольку управление форматированием вывода символьных и строковых литералов производится в соответствии с правилами умолчания.

> **Пример 9**

Отметим, что в С символьные литералы относятся к типу ***int***, поэтому уже на первых порах крайне важно понимание свойств фундаментального типа С++ ***char.***

Для описания свойств целых чисел С++ унаследовал от С макросы, которые определены в стандартных заголовочных файлах **<*climits*>** и **<*Iimits.h*>.** Например, с помощью макроса ***CHAR BIT*** можно выяснить количество бит, необходимых для представления типа ***char*** в данной реализации, а с помощью макросов ***CHAR MIN*** и ***CHAR_MAX*** можно получить ответ на вопрос, будет ли ***char*** со знаком или без знака.

Итак, представим программу для иллюстрации свойств типа С++ ***char*** с использованием макросов стандартной библиотеки:

```
// C++ Макросы стандартной библиотеки
#include <iostream>
#include <climits>
int main O {
  using namespace std;
  printf("%d %d %d\n", CHAR_BIT, CHARJMIN, CHAR_MAX);
  cout << CHARJ3IT << '     ' << CHARJMIN << '   ' << CHAR_MAX << endl;
  return 0;
}
```

Результат работы программы:

8 -128 127
8 -128 127

Как видим здесь, в реализации предполагается, что для представления типа ***char*** требуется 8 бит, а сам ***char*** будет со знаком.

Необходимо отметить, что макросы имеют большое значение в С, но в С++ они используются значительно реже. Так как макросы изменяют текст программы до обработки его компилятором благодаря выполняемым с помощью препроцессора С макроподстановкам, они зачастую становятся источником нежелательных побочных эффектов и порой неустранимых проблем. Здесь же и далее стандартные макросы станут одним из инструментов для исследования деталей реализации стандартной библиотеки С++.

> **Пример 10**

Продолжая иллюстрацию форматного вывода символьных литералов, относящихся к фундаментальному типу C++ ***char,*** теперь обратимся к программе, где символьные литералы представлены с помощью escape-последовательностей:

```
// C++ Символьные литералы #include <iostream> int main ()
{
using namespace std;
printf("%c%c%c%c%c", '\103', '\53', '\53', '\41', '\12');
printf("%c%c%c%c%c\n",      '\x43',  '\x2b',  '\x2b',  '\x21',  '\xa');
cout << '\103' << '\53' << '\53' << ' \41' << endl;
cout << '\x43' << '\x2b' << '\x2b' << '\x21' << endl;
return 0;
}
```

Результат работы программы:

```
C++!
C++!
C++!
C++!
```

Так как для представления фундаментального типа C++ ***char*** используется 8 бит, то для представления символьных литералов с помощью escape-последовательностей можно использовать либо от одной и до трех восьмеричных цифр, либо от одной и до двух шестнадцатеричных цифр. При этом, как видим, в случае шестнадцатеричного представления символьных литералов обязательно требуется указание системы счисления, именно поэтому следом за escape-символом \ следует символ ***x*** (от слова ***hexadecimal),*** ставший в C и C++ традиционным “указателем” шестнадцатеричной системы счисления.

Вывод десятичных литералов

> Пример 11

Представим первую программу для иллюстрации форматного вывода десятичных литералов, которые относятся к фундаментального типу C++ ***int*** (целое со знаком):

```
// C++ Десятичные литералы #include <iostream> int main()
{
  using namespace std;
  printf("%d\t\t%d\n", -32768, 32767);
  printf("%o\t%o\n", -32768, 32767);
  printf("%x\t%X\n\n", -32768, 32767);
  cout  <<  -32768  << "\t\t"  << 32767 << endl;
  cout  <<  oct  << -32768  << '\t' << 32767 << endl;
  cout  <<  hex  << -32768  << '\t' << uppercase << 32767 << endl;
  cout  <<  dec  << -32768  << "\t\t" << 32767 << endl;
  return 0;
}
```

Результат работы программы:

```
-32768                32767
37777700000           77777
ffff8000               7FFF

-32768                32767
37777700000           77777
ffff8000               7FFF
-32768                32767
```

Отметим, что если литерал десятичный и не имеет суффикса, то он относится к первому из следующих типов, который окажется достаточным для представления его значения: ***int, long int, unsigned long int.***

Спецификации преобразования ***%o*** (от слова ***octal)*** и ***%x*** (от слова ***hexadecimal)*** или ***%X*** (если требуется переключение с нижнего на верхний регистр) используются для указания преобразования целочисленного аргумента вызова функции ***printj***0, соответственно, в восьмеричный и шестнадцатеричный вид. Использование символа горизонтальной табуляции **t** позволяет организовать вывод в две колонки.

Для управления механизмом фиксации состояния формата стандартного потока вывода ***cout*** здесь используются манипуляторы ***oct*** (от слова ***octal), hex*** (от слова ***hexadecimal), dec*** (от слова ***decimal)*** и ***uppercase*** (верхний регистр).

Как уже отмечалось, для описания свойств целых чисел C++ унаследовал от C макросы, которые определены в стандартных заголовочных файлах ***<climits>*** и ***<limits.h>.*** Например, вместо десятичных литералов ***-32768*** и ***32767*** теперь можно воспользоваться макросами ***SHRT_MIN*** и ***SHRT_MAX,*** представляющими собой предельные значения для фундаментального типа C++ ***short int*** (короткое целое со знаком, диапазон изменения значений от -32768 и до 32767 для 32-разрядной архитектуры).

Итак, представим программу для иллюстрации форматного вывода предельных значений для типа ***short int*** с использованием макросов стандартной библиотеки:

```
// C++ Макросы стандартной библиотеки
#include <iostream>
#include <climits>
int main()
{
using namespace std;
printf("%d\t\t%d\n", SHRT_MIN, SHRT_MAX);
printf("%o\t%o\n", SHRT_MIN, SHRT_MAX);
printf("%x\t%x\n\n", SHRT_MIN, SHRT_MAX);
cout  <<    SHRT_MIN << "\t\t"    << SHRT__MAX << endl;
cout  <<    oct <<  SHRT_MIN   << '\t' << SHRT_MAX << endl;
cout  <<    hex <<  SHRT_MIN   << '\t' << SHRT_MAX << endl;
cout  <<    dec <<  SHRT_MIN   << "\t\t" << SHRT_MAX << endl;
return 0;
}
```

Результат работы программы:

```
-32768                  32767
37777700000             77777
ffff8000                7fff

-32768                  32767
37777700000             77777
ffff8000                Ifff
-32768                  32767
```

> Пример 13

Продолжая иллюстрацию форматного вывода десятичных литералов, выясним, как при этом будут выглядеть предельные значения для фундаментального типа C++ ***int*** (целое со знаком, диапазон изменения значений от -2147483648 и до 2147483647 для 32-разрядной архитектуры). Наряду с десятичными литералами будем использовать также и макросы стандартной библиотеки ***INT MIN*** и ***INT_MAX,*** представляющие собой предельные значения для типа ***int.***

Итак, представим программу для иллюстрации форматного вывода предельных значений для типа ***int*** с использованием десятичных литералов и макросов:

```
// C++ Десятичные литералы и стандартные макросы
#include <iostream>
#include <climits>
int main()
{
 using namespace std;
 printf("%d\t%d\n", -2147483648, 2147483647);
 printf("%d\t%d\n", (-2147483647 – 1), 2147483647);
 printf("%d\t%d\n", INT_MIN, INT_MAX);
 printf("%o\t%o\n", -2147483648, 2147483647);
 printf("%x\t%x\n\n", -2147483648, 2147483647);
 cout << -2147483648 << '\t' << 2147483647 << endl;
 cout << (-2147483647 – 1) << '\t' << 2147483647 << endl;
 cout  <<  INT_MIN << ' \t'  << INT_MAX  << endl;
 cout  <<  oct  <<   -2147483648  << '\t' << 2147483647   << endl;
 cout  <<  hex  <<   -2147483648  << '\t' << 2147483647   << endl;
 cout  <<  dec  <<   -2147483648  << '\t' << 2147483647   << endl;
 return 0;
}
```

Результат работы программы:

```
-2147483648         2147483647
-2147483648         2147483647
-2147483648         2147483647
20000000000         17777777777
80000000            7fffffff

2147483648          2147483647
-2147483648         2147483647
-2147483648         2147483647
20000000000         17777777777
80000000            7fffffff
2147483648          2147483647
```

Как видим, вполне “безобидный” код привел к весьма неожиданному результату при выводе десятичного представления литерала ***-2147483648*** с использованием стандартного потока вывода ***cout.*** Мало того, компиляторы для платформы Linux выдают предупреждающее сообщение времени компиляции о том, что этот литерал будет ***unsigned*** (без знака) только для стандарта ISO C90 (стандарт C ISO/IEC 9899, принятый в 1990 году).

Чтобы понять причину получения такого результата, необходимо здесь отметить, что тип целого литерала зависит от его формы, значения и суффикса. Если литерал десятичный и не имеет суффикса, он, как уже было сказано, относится к первому из следующих типов, который окажется достаточным для представления его значения: ***int, long int, unsigned long int.*** Как видим, в случае использования стандартного потока вывода ***cout*** литерал ***-2147483648*** может быть отнесен компилятором только к типу ***unsigned long int*** (длинное целое без знака), невзирая на то, что его можно отнести и к типу ***int.*** Здесь своеобразным “порогом” является значение ***-2147483647,*** которое компилятор относит к типу ***int.***

Конечно же, всякий раз необходимо помнить о подобного рода “подстраховках” со стороны компилятора и уметь их обходить. Далее будет сказано, как можно решить эту проблему, если воспользоваться инструкциями-объявлениями.

Отметим здесь также, что если литерал восьмеричный или шестнадцатеричный и не имеет суффикса, он относится к первому из следующих типов, который окажется достаточным для представления его значения: ***int, unsigned int, long int, unsigned long int.***

Чтобы понять причину получения ожидаемого результата в случае использования макросов, необходимо обратиться к их определениям в стандартных заголовочных файлах ***<climits>*** и ***<limits.h>.*** Отметим, что для определения макросов используется директива препроцессора C ***Udefine.*** Так, среди возможных способов определения макросов, представляющих собой предельные значения для фундаментального типа C++ ***int,*** например, могут быть такие:

```
#define INT_MIN        (-2147483647 — 1)
#define INT_MAX        2147483647
```

или такие:

```
#define    INT_MAX     2147483647
#define    INT_MIN     (-INT_MAX — 1)
```

Как видим здесь, такого рода приемы определения макроса ***INT MIN*** позволяют сохранить тип ***int*** для вычисленного значения арифметического выражения.

Ранее были использованы макросы ***SHRT_MIN*** и ***SHRT MAX,*** представляющие собой предельные значения для фундаментального типа C++ ***short int.*** Как правило, эти макросы определяются в виде десятичных литералов:

```
#define    SHRT_MIN    (-32768)
#define    SHRT_MAX    32767
```

Однако возможны и другие способы определения, подобные тем, что были представлены для макросов ***INT_MIN*** и ***INT MAX.***

> Пример 14

Продолжая иллюстрацию форматного вывода десятичных литералов, выясним, как теперь при этом будут выглядеть предельные значения для фундаментального типа C++ ***long ini*** (длинное целое со знаком, диапазон изменения значений от -2147483648 и до 2147483647 для 32-разрядной архитектуры). Наряду с десятичными литералами будем использовать и макросы стандартной библиотеки ***LONG MIN*** и ***LONG MAX,*** представляющие собой предельные значения для типа ***long int.***

Итак, представим программу для иллюстрации форматного вывода предельных значений для типа ***long int*** с использованием десятичных литералов и макросов:

```
// C++ Десятичные литералы и стандартные макросы
#include <iostream>
#include <climits>
int main()
{
 using namespace std;
 printf("%d\t%ld\n", -2147483648L, 2147483647L);
 printf("%d\t%ld\n",      (-2147483647L   - 1), 2147483647L);
 printf("%d\t%ld\n", LONG_MIN, LONG_MAX);
 printf("%o\t%lo\n", -2147483648L, 2147483647L);
 printf("%x\t%lx\n\n", -2147483648L, 2147483647L);
 cout << -2147483648L << '\t' << 2147483647L << endl;
 cout << (-2147483647L – 1) << '\t' << 2147483647L << endl;
 cout  <<  LONG_MIN << '\t'  <<  LONG_MAX << endl;
 cout  <<  oct <<   -2147483648L  <<'\t'  <<   2147483647L  << endl;
 cout  <<  hex <<   -2147483648L  <<'\t'  <<   2147483647L  << endl;
 cout  <<  dec <<   -2147483648L  <<'\t'  <<   2147483647L  << endl;
 return 0;
}
```

Результат работы программы:

```
-2147483648         2147483647
-2147483648         2147483647
-2147483648         2147483647
20000000000         17777777777
80000000            7fffffff

2147483648          2147483647
-2147483648         2147483647
-2147483648         2147483647
20000000000         17777777777
80000000            7fffffff
2147483648          2147483647
```

Как видим, и здесь при выводе десятичного представления литерала ***-2147483648L*** с использованием стандартного потока вывода ***cout*** получен тот же результат, что и ранее для литерала ***-2147483648.*** Отметим, что если литерал десятичный и имеет суффикс / или ***L,*** то он относится к первому из следующих типов, который окажется достаточным для представления его значения: ***long int***, ***unsigned long int.***

Таким образом, и литерал ***-2147483648L*** может быть отнесен компилятором только к типу ***unsigned long int*** (длинное целое без знака), невзирая на то, что его можно отнести и к типу ***long int.*** Только в этом случае своеобразным "порогом" является значение ***-2147483647L,*** которое компилятор относит к типу ***long int.***

Как и ранее, причину получения ожидаемого результата в случае использования макросов, следует искать в их определениях в стандартных заголовочных файлах ***<climits>*** и ***<limits.h>.*** Среди возможных способов определения макросов, которые представляют собой предельные значения для фундаментального типа C++ ***long int,*** например, могут быть такие:

```
#define LONG_MIN      (-2147483647L    - 1)
#define LONG_MAX      2147483647L
```

или такие:

```
#define LONG_MAX      2147483647L
#define LONG_MIN      (-LONG_MAX -   1)
```

Как видим, теперь и здесь такого рода приемы определения макроса ***LONGJMIN*** позволяют сохранить тип ***long int*** для вычисленного значения арифметического выражения.

Отметим, что в случае использования функции ***printj***0 для уточнения размера типа ее аргументов вызова ***(short*** или ***long)*** перед каждым символом-спецификатором в форматной строке можно указать префикс ***h*** (от слова ***short)*** или / (от слова ***long).***

Итак, первые результаты, полученные при использовании в программе десятичных литералов, наглядно демонстрируют, что от пользователя требуется соблюдение известных мер предосторожности, особенно в тех случаях, когда значения литералов близки к предельным. Операции вывода десятичного представления литералов лишь выявили эту проблему, для решения которой существует немало приемов. Одним из таких приемов, как уже было сказано, станет использование инструкций-объявлений, с помощью которых от десятичных литералов в дальнейшем можно будет перейти и к другим объектам программы — константам и переменным.

И завершая иллюстрацию форматного вывода десятичных литералов, выясним, как при этом будут выглядеть предельные значения для двух фундаментальных типов C++ ***unsigned int*** и ***unsigned long int*** (целое без знака и длинное целое без знака, диапазон изменения значений от 0 и до 4294967295 для 32-разрядной архитектуры). Наряду с десятичными литералами будем также использовать и макросы стандартной библиотеки ***UINT MAX*** и ***ULONG MAX,*** которые представляют собой предельные значения для типов ***unsigned int*** и ***unsigned long int.***

Итак, представим программу для иллюстрации форматного вывода предельных значений для типов ***unsigned int*** и ***unsigned long int*** с использованием десятичных литералов и макросов:

```
// C++ Десятичные литералы и стандартные макросы
#include <iostream>
#include <climits>
int main()
{
using namespace std;
printf("%d\t\t%ld\n", 4294967295U, 4294967295UL);
printf("%o\t%lo\n", 4294967295U, 4294967295UL);
printf("%x\t%lx\n", 4294967295U, 4294967295UL);
printf("%u\t%lu\n", 4294967295U, 4294967295UL);
printf("%u\t%lu\n\n", UINT_MAX, ULONG_MAX);
cout << 4294967295U << '\t' << 4294967295UL << endl;
cout  <<  UINT_MAX << '\t' << ULONG_MAX          << endl;
cout  <<  oct  <<4294967295U   << '\t'  <<   4294967295UL << endl;
cout  <<  hex  <<4294967295U   << '\t'  <<   4294967295UL << endl;
cout  <<  dec  <<4294967295U   << '\t'  <<   4294967295UL << endl;
return 0;
}
```

Результат работы программы:

```
-1                      -1
37777777777             37777777777
ffffffff                ffffffff
4294967295              4294967295
4294967295              4294967295

4294967295              4294967295
4294967295              4294967295
37777777777             37777777777
ffffffff                ffffffff
4294967295              4294967295
```

Как видим, все операции вывода дали вполне ожидаемые результаты, остается лишь отметить, что если литерал десятичный и имеет суффикс ***ul, lu***, ***uL, Lu, Ul, IU***, ***UL*** или ***LU,*** то он относится к типу ***unsigned long int.*** Это правило справедливо также для восьмеричных и шестнадцатеричных литералов.

Помимо уже известных спецификаций преобразования здесь в форматной строке функции ***printf()*** используется еще одна: ***%u*** (от слова ***unsigned)*** для указания преобразования целочисленного аргумента без знака в десятичный вид.

Обратим теперь внимание на определения в стандартных заголовочных файлах **<*climits*>** и **<*limits.h*>** макросов ***UINT MAX*** и ***ULONG_MAX,*** представляющих собой предельные значения фундаментальных типов C++ ***unsigned int*** (целое без знака) и ***unsigned long int*** (длинное целое без знака). Как правило, эти макросы определяются в виде шестнадцатеричных литералов:

```
#define UINT_MAX 0xffffffff #define ULONG_MAX 0xffffffffUL
```

Подобным образом в стандартных заголовочных файлах **<*climits*>** и **<*limits.h*>** определяется и макрос ***USHRT_MAX,*** представляющий собой предельные значения фундаментального типа C++ ***unsigned short int*** (короткое целое без знака, диапазон изменения значений от 0 и до 65535 для 32-разрядной архитектуры):

```
#define USHRT_MAX 0xffff
```

Напомним также, что использование суффиксов ***L, U*** и ***UL*** необходимо не только для явного указания типа литералов при их записи, но прежде всего для того, чтобы избежать зависимостей от реализации при определении размера и знака целых чисел. Главным в таком подходе должно быть стремление к созданию переносимого кода.

В заключение отметим, что наряду с традиционным использованием макросов из стандартных заголовочных файлов **<*climits*>** и **<*limits.h*>** существует и другой способ удовлетворения любопытства по поводу предельных значений целых чисел, который напрямую связан с обобщенной парадигмой программирования, так как он основан на использовании специализаций класса-шаблона ***numeric limits,*** представленного в заголовочном файле **<*limits*>.** Под специализацией шаблона понимается одна из его альтернативных версий, которая выбирается компилятором по аргументу шаблона, но об этом уже в следующем примере.

> Пример 16

Итак, представим программу для иллюстрации форматного вывода предельных значений для целых типов C++ ***short int, int, long int, unsigned short int, unsigned int*** и ***unsigned long int*** с использованием специализаций класса-шаблона ***numericlimits:***

```
// C++ Специализации шаблона numeric_limits
#include <iostream>
#include <limits>
int main()
{
  using namespace std;
  cout << numeric__limits<short int>: :min() << "\t\t"
       << numeric_limits<short int>::max() << endl;
cout << numeric__limits<int>: :min () << '\t'
       << numeric_limits<int>::max() << endl;
cout << numeric_limits<long int>::min() << '\t'
       << numeric_limits<long int>::max() << endl;
cout << numeric_limits<unsigned short int>::max() << endl;
cout << numeric_limits<unsigned int>::max() << endl;
cout << numeric_limits<unsigned long int>::max() << endl;
return 0;
}
```

Результат работы программы:

```
-32768              32767
-2147483648         2147483647
-2147483648         2147483647
65535
4294967295
4294967295
```

Как видим, аргументы шаблона (целые типы), для которых должна использоваться специализация, задаются в угловых скобках после ее имени. Квалификация имен функций-членов ***min()*** и ***max()*** с помощью унарного оператора разрешения области видимости :: означает, что эти имена принадлежат пространству имен каждой из специализаций класса-шаблона ***numeric limits.***

Вывод восьмеричных литералов

Пример 17

Итак, получив представление о форматном выводе десятичных литералов в виде восьмеричных и шестнадцатеричных чисел, теперь легко перейти к иллюстрации форматного вывода восьмеричных и шестнадцатеричных литералов. Поначалу представим программу для иллюстрации форматного вывода восьмеричных литералов:

```
// C++ Восьмеричные литералы #include <iostream> int main ()
{
  using namespace std;
  printf("%o\t%o\n", 037777700000, 077777);
  printf("%d\t\t%d\n", 037777700000, 077777);
  printf("%o\t%o\n", 020000000000, 017777777777);
  printf("%d\t%d\n", 020000000000, 017777777777);
  printf("%o\t%lo\n", 037777777777, 037777777777L);
  printf("%d\t\t%ld\n\n", 037777777777, 037777777777UL);
cout  <<     oct << 037777700000   << '\t'  <<077777 << endl;
cout  <<     dec << 037777700000   << '\t'  <<077777 << endl;
cout  <<     oct << 020000000000   <<>>\t'  <<017777777777  <<   endl;
cout  <<     dec << 020000000000   << '\t'  <<017777777777  <<   endl;
cout << oct << 037777777777 << '\t' << 037777777777L << endl;
cout << dec << 037777777777 << '\t' << 037777777777UL << endl;
return 0;
}
```

Результат работы программы:

```
37777700000         77777
-32768              32767
20000000000         17777777777
-2147483648         2147483647
37777777777         37777777777
-1                  -1

37777700000         77777
4294934528          32767
20000000000         17777777777
2147483648          2147483647
37777777777         37777777777
4294967295          4294967295
```

Вывод шестнадцатеричных литералов

Пример 18

В завершение представим программу для иллюстрации форматного вывода шестнадцатеричных литералов:

```
// C++ Шестнадцатеричные литералы #include <iostream> int main ()
{
using namespace std;
printf("%x\t%x\n", 0xffff8000, 0X7FFF);
printf("%d\t\t%d\n", 0xffff8000, 0x7FFF);
printf("%x\t%x\n", 0x80000000, 0x7fffffff);
printf("%d\t%d\n", 0x80000000, 0x7fffffff);
printf("%x\t%lx\n", 0xffffffff, 0xffffffffL);
printf("%d\t\t%ld\n\n", 0xffffffff, 0xffffffffUL);
cout  <<    hex  << 0xffff8000   <<  '\t'  <<    0X7FFF << endl;
cout  <<    dec  << 0xffff8000   <<  '\t'  <<    0x7FFF << endl;
cout  <<    hex  << 0x80000000   <<  '\t'  <<    0x7fffffff << endl;
cout  <<    dec  << 0x80000000   <<  '\t'  <<    0x7fffffff << endl;
cout << hex << 0xffffffff << '\t' << 0xffffffffL << endl;
cout << dec << 0xffffffff << '\t' << 0xffffffffUL << endl;
return 0;
}
```

Результат работы программы:

```
ffff8000            7fff
-32768              32767
80000000            7fffffff
-2147483648         2147483647
ffffffff            ffffffff
-1                  -1

ffff8000            7fff
4294934528          32767
80000000            7fffffff
2147483648          2147483647
ffffffff            ffffffff
4294967295          4294967295
```

Вывод литералов с плавающей точкой

> Пример 19

Представим первую программу для иллюстрации форматного вывода литералов с плавающей точкой, относящихся к фундаментального типу C++ ***double*** (число с плавающей точкой двойной точности):

```
// C++ Литералы с плавающей точкой #include <iostream> int main ()
{
 using namespace std;
printf("%g\t\t%G\n", 100000., -100000.);
printf("%g\t\t%G\n", 1000000.0, -1000000.0);
printf("%g\t\t%G\n", 0.0001, -0.0001);
printf("%g\t\t%G\n", 0.00001, -0.00001);
printf("%g\t%G\n", 0.123456789, -123456789.0);
printf("%e\t%E\n", 0.123456789, -123456789.0);
printf("%f\t%f\n", 1., -1.0);
printf("%f\t%f\n\n", 0.123456789, -123456789.0);
cout << 100000. << "\t\t" << -100000. << endl;
cout << 1000000.0 << "\t\t" << -1000000.0 << endl;
cout << 0.0001 << "\t\t" << -0.0001 << endl;
cout << 0.00001 << "\t\t" << -0.00001 << endl;
cout << 0.123456789 << '\t' << -123456789.0 << endl;
return 0;
}
```

Результат работы программы для платформы Windows:

```
100000              -100000
1e+006              -1E+006
0.0001              -0.0001
le-005              -IE-005
0.123457            -1.23457E+008
1.234568e-001       -1.234568E+008
1.000000            -1.000000
0.123457            -123456789.000000

100000              -100000
le+006              -le+006
0.0001              -0.0001
le-005              -le-005
0.123457            -1.23457E+008
```

Результат работы программы для платформы Linux:

```
100000              -100000
1e+06               -1E+06
0.0001              -0.0001
1e-05               -1E-05
0.123457            -1.23457E+08
1.234568e-01        -1.234568E+08
1.000000            -1.000000
0.123457            -123456789.000000

100000              -100000
1e+06               -1e+06
0.0001              -0.0001
1e-05               -1e-05
0.123457            -1.23457E+08
```

Там, где это необходимо, будут представлены два протокола результатов: первый — для платформы Windows, а второй — для платформы Linux. Будут представлены также и разные версии программы. А причина этому — различия в реализации.

Отметим, что если литерал с плавающей точкой и не имеет суффикса, то он относится к типу ***double.*** Отметим также, что вывод литералов с плавающей точкой определяется форматом и точностью. При этом, как видим, их вывод осуществляется только с округлением. Здесь представлены все форматы вывода — универсальный, научный и фиксированный. Для всех форматов точность по умолчанию — 6 цифр.

Универсальный формат дает реализации самой выбирать формат представления числа в том виде, который наилучшим образом представит это число. Максимальное количество цифр определяется точностью. Универсальный формат соответствует спецификации преобразования ***%g*** или ***%G*** (от слова ***general).***

Научный формат представляет число десятичной дробью с одной цифрой перед точкой, цифрами дробной части, максимальное количество которых определяется точностью, и показателем степени. Научный формат соответствует спецификации преобразования ***%e*** или ***%E*** (от слова ***exponential).*** Иногда научный формат называют экспоненциальным, однако здесь будем придерживаться его традиционного для C++ именования. Из протоколов результатов ясно видно, что различия в реализации пока касаются лишь научного формата.

Фиксированный формат представляет число как целую часть с дробной частью, отделенной точкой. Точность определяет максимальное количество цифр дробной части числа. Фиксированный формат соответствует спецификации преобразования ***%of*** (от слова ***fixed).***

Как видим, вывод литералов с плавающей точкой с использованием стандартного потока вывода ***cout*** осуществляется здесь под управлением универсального формата, являющегося форматом по умолчанию. Далее, шаг за шагом, будет сказано, какими еще средствами для управления состоянием формата потока располагает стандартная библиотека.

> Пример 20

Получив представление о манипуляторах стандартной библиотеки, фиксирующих состояние формата, теперь для управления состоянием стандартного потока вывода ***cout*** можно воспользоваться еще двумя такими манипуляторами: ***scientific*** (научный формат) и ***fixed*** (фиксированный формат). Манипулятор ***scientific*** сам по себе соответствует спецификации преобразования ***%e,*** а вместе с уже известным манипулятором ***uppercase*** — спецификации преобразования ***%E.*** Манипулятор ***fixed*** соответствует спецификации преобразования ***%f.***

Продолжим иллюстрацию форматного вывода литералов с плавающей точкой, относящихся к типу ***double***, привлекая манипуляторы ***scientific, uppercase*** и ***fixed***:

```
// C++ Литералы с плавающей точкой #include <iostream> int main()
{
  using namespace std;
  printf("%g\t\t%g\n", lei, -IE-2);
  printf("%G\t\t%G\n", lei, -IE-2);
  printf("%e\t%e\n", lei, -IE-2);
  printf("%E\t%E\n", lei, -IE-2);
  printf("%f\t%f\n\n", lei, -IE-2);
  cout << lel << "\t\t" << -IE-2 << endl;
  cout << scientific << lel << '\t' << -IE-2 << endl;
  cout << uppercase << lel << '\t' << -IE-2 << endl;
  cout << fixed << lel << '\t' << -IE-2 << endl;
  return 0;
}
```

Результат работы программы для платформы Windows:

```
10                  -0.01
10                  -0.01
1.000000e+OOl       -1.000000e-002
1.000000E+001       -1.000000E-002
10.000000           -0.010000

10                  -0.01
1.000000e+001       -1.000000e-002
1.000000E+001       -1.000000E-002
10.000000           -0.010000
```

> Пример 21

Если литерал с плавающей точкой и имеет суффикс ***f*** или ***F,*** то он относится к фундаментальному типу C++ ***float*** (число с плавающей точкой одинарной точности), а если суффикс *l* или ***L,*** то — к фундаментальному типу C++ ***long double*** (число с плавающей точкой расширенной точности).

Итак, представим программу для иллюстрации форматного вывода литералов с плавающей точкой, относящихся к типам ***float*** и ***long double***:

```
// C++ Литералы с плавающей точкой #include <iostream> int main()
{
 using namespace std;
 printf("%g\t\t%g\n", 10. f, -0.01F);
 printf("%g\t\t%g\n", lelf, -1E-2F);
 printf("%e\t%e\n", lelf, -1E-2F);
 printf("%f\t%f\n", lelf, -1E-2F);
 printf("%e\t%e\n", lell, -1E-2L);
 printf("%f\t%f\n\n'\ lell, -1E-2L);
 cout << 10.f << "\t\t" << -0.01F << endl;
 cout << lelf << "\t\t" << -1E-2F << endl;
 cout << scientific << lelf << '\t' << -1E-2F << endl;
 cout << fixed << lelf << '\t' << -1E-2F << endl;
 cout << scientific << lell << '\t' << -1E-2L << endl;
 cout << fixed << lell << '\t' << -1E-2L << endl;
 return 0;
}
```

Результат работы программы для платформы Windows:

```
10                  -0.01
10                  -0.01
1.000000e+001       -1.000000e-002
10.000000           -0.010000
1.000000e+001       -1.000000e-002
10.000000           -0.010000

10                  -0.01
10                  -0.01
1.000000e+001       -1.000000e-002
10.000000           -0.010000
1.000000e+001       -1.000000e-002
10.000000           -0.010000
```

> Пример 22

Чтобы выяснить, какими будут предельные значения для фундаментальных типов C++ ***float***, ***double*** и ***long double,*** как и в случае с целыми типами, поначалу обратимся к макросам. В стандартных заголовочных файлах <***cfloat***> и <***float.h***> определены макросы, описывающие свойства чисел с плавающей точкой. Обратимся к макросам, которые представляют собой предельные значения самых малых и самых больших положительных чисел с плавающей точкой. Так, для типа ***float*** — это макросы ***FLT MIN*** и ***FLT MAX,*** для типа ***double*** — это макросы ***DBL MIN*** и ***DBL MAX,*** а для типа ***long double*** **—** это макросы ***LDBL MIN*** и ***LDBL MAX.***

Итак, представим программу для иллюстрации форматного вывода предельных значений для типов ***float, double*** и ***long double*** для 32-разрядной архитектуры с использованием макросов стандартной библиотеки:

```
// C++ Макросы стандартной библиотеки
#include <iostream>
#include <cfloat>
int main ()
{
using namespace std;
printf("%.9e\t\t%.9e\n", FLT_MIN, FLT_MAX);
printf("%.16e\t\t%.16e\n", DBL_MIN, DBL_MAX);
printf("%.16e\t\t%.16e\n\n", LDBL_MIN, LDBL_MAX);
cout.precision(10);
cout << FLT_MIN << "\t\t" << FLT_MAX << endl;
cout.precision(17);
cout << DBL_MIN << "\t\t" << DBL_MAX << endl;
cout << LDBL_MIN << "\t\t" << LDBL_MAX << endl;
return 0;
}
```

Результат работы программы для платформы Windows:

```
1.175494351e-038                        3.402823466e+038
2.2250738585072014e-308                 1.7976931348623157e+308
2.2250738585072014e-308                 1.7976931348623157e+308

1.175494351e-038                        3.402823466e+038
2.2250738585072014e-308                 1.7976931348623157e+308
2.2250738585072014e-308                 1.7976931348623157e+308
```

Чтобы задать требуемую точность для научного формата при выводе предельных значений для типов ***double*** и ***long double,*** необходимо воспользоваться механизмом явного указания точности либо в виде количества цифр дробной части, либо в виде максимального количества выводимых цифр числа.

Так, в форматной строке функции ***printf()***в каждой спецификации преобразования для научного формата, как видим, содержатся два элемента для указания точности: символ “точка” и число, обозначающее количество цифр дробной части. А с помощью инструкции-выражения для стандартного потока вывода ***cout,*** как видим, вызывается функция-член ***precision***() с аргументом, обозначающим максимальное количество выводимых цифр числа. Очевидно, что максимальное количество выводимых цифр должно быть на единицу больше заданного количества цифр дробной части числа.

Отметим здесь, что функция-член ***precision()*** принадлежит классу ***ios base,*** а сами инструкции-выражения **`cout.precision(10);`**

и

`cout.precision(17);`

представляет собой так называемые уточненные имена — выражения с бинарным оператором . (операция выбора члена), где операндами являются имя потока ***cout*** и имя функции-члена ***precision().*** Уточненное имя функции-члена класса обеспечивает ее вызов для обработки данных именно того объекта класса, имя которого как раз и было использовано в уточненном имени этой функции.

Отметим также, что вызов функции-члена ***precision()*** влияет на все последующие операции вывода чисел с плавающей точкой в поток и действует до следующего обращения к ней.

Будут ли предельные значения для типов ***double*** и ***long double*** одинаковы или нет, как видим, зависит от реализации. Мало того, если обратиться к самим определениям стандартных макросов для типов ***double*** и ***long double,*** оказывается, что определены они не для одинаковых предельных значений:

```
#define DBL_MIN 2.2250738585072014e-308 #define DBL_MAX
1.7976931348623157e+308
  #define LDBL_MIN      3.362103143112093506 3e-4932L
  #define LDBL_MAX      1.189731495357231765e+4932L
```

Лишь благодаря директивам условной трансляции препроцессора С эти макросы для платформы Windows оказались связаны с одними и теми же значениями.

Поскольку для платформ Windows и Linux от реализации зависит еще и сам вывод под управлением научного формата, то теперь необходимо представить версию этой программы и для платформы Linux.

> Пример 23

Итак, представим программу для платформы Linux для иллюстрации форматного вывода предельных значений для типов ***float***, ***double*** и ***long double*** для 32-разрядной архитектуры с использованием макросов стандартной библиотеки:

```
// C++ Макросы стандартной библиотеки
#include <iostream>
#include <cfloat>
int main()
{
using namespace std;
printf("%.9e\t\t\t%.9e\n", FLT_MIN, FLT_MAX);
printf("%.16e\t\t%.16e\n", DBL_MIN, DBL_MAX);
printf("%.19e\t%.19e\n\n", LDBL_MIN, LDBL_MAX);
cout.precision(10);
cout << FLT_MIN << "\t\t\t" << FLT_MAX << endl;
cout.precision(17);
cout << DBL_MIN << "\t\t" << DBL_MAX << endl;
cout.precision(20);
cout << LDBL_MIN << '\t' << LDBL_MAX << endl;
return 0;
}
```

Результат работы программы для платформы Linux:

```
1.175494351e-38                         3.4028234ббe+38
2.2250738585072014e-308                 1.797б931348623157e+308
3.3621031431120935063e-4932             1.189731495357231765e+4932
1.175494351e-38                         3.4028234ббe+038
2.2250738585072014e-308                 1.797б931348623157e+308
3.3621031431120935063e-4932             1.189731495357231765e+4932
```

Будут ли предельные значения для типов ***double*** и ***long double*** одинаковы или нет, как видим, и здесь зависит от реализации.

> Пример 24

И в завершение представим программу для иллюстрации форматного вывода предельных значений для типов ***float, double*** и ***long double*** для 32-разрядной архитектуры с использованием специализаций класса-шаблона ***numeric limits***:

```
// C++ Специализации шаблона numeric_limits
#include <iostream>
#include <limits>
int main()
{
using namespace std;
 cout.precision(10);
 cout << numeric_limits<float>::min() << "\t\t"
      << numeric_limits<float>::max() << endl;
 cout.precision(17);
 cout << numeric_limits<double>::min() << "\t\t"
      << numeric_limits<double>::max() << endl;
 cout << numeric_limits<long double>: :min () << "\t\t"
      << numeric_limits<long double>::max() << endl;
return 0;
}
```

Результат работы программы для платформы Windows:

```
1.17549435e-038                    3.402823466e+038
2.2250738585072014e-308            1.797б931348623157e+308
2.2250738585072014e-308            1.7976931348623157e+308
```

Памятуя о том, что для платформ Windows и Linux от реализации зависит еще и сам вывод под управлением научного формата, то теперь необходимо представить версию этой программы и для платформы Linux.

> Пример 25

Итак, представим программу для платформы Linux для иллюстрации форматного вывода предельных значений для типов ***float***, ***double*** и ***long double*** для 32-разрядной архитектуры с использованием специализаций класса-шаблона ***numeric limits***:

```
// C++ Специализации шаблона numeric_limits
#include <iostream>
#include <limits>
int main()
{
using namespace std;
 cout.precision(10);
  cout << numeric_limits<float>::min() << "\t\t\t"
       << numeric_limits<float>::max() << endl;
  cout.precision(17);
  cout << numeric_limits<double>::min() << "\t\t"
       << numeric_limits<double>::max() << endl;
  cout.precision(20);
  cout << numeric_limits<long double>::min() << '\t'
       << numeric_limits<long double>::max() << endl;
return 0;
}
```

Результат работы программы для платформы Linux:

```
1.175494351e-38                         3.402823466e+38
2.2250738585072014e-308                 1.7976931348623157e+308
3.362103143112093506Зe-4932             1.189731495357231765e+4932
```

Таким образом, точный смысл каждого типа с плавающей точкой действительно зависит от реализации. Известно, что выбор нужной точности в реальных задачах требует хорошего понимания природы машинных вычислений с плавающей точкой, а если его нет, то, как отмечает Страуструп, используйте тип ***double*** и надейтесь на лучшее.

Вывод строковых литералов

> Пример 26

Получив представление о форматном выводе одиночных строковых литералов, перейдем теперь к иллюстрации форматного вывода группы строковых литералов:

```
// C++ Строковые литералы #include <iostream> int main()
{
using namespace std;
printf("%s%s", "Первый ", "Второй\n");
printf("%s %s\n", "Первый", "Второй");
printf("Первый %s", "Второй\n");
printf("Первый %s\n", "Второй");
printf("Первый %s %s\n", "Второй", "Третий");
printf("Первый %s Третий\n\n", "Второй");
cout << "Первый" << " Второй\n";
cout << "Первый " << "Второй" << endl;
cout << "Первый " << "Второй " << "Третий" << endl;
return 0;
}
```

Результат работы программы:

```
Первый Второй
Первый Второй
Первый Второй
Первый Второй
Первый Второй Третий
Первый Второй Третий

Первый Второй
Первый Второй
Первый Второй Третий
```

Управление состоянием потока вывода — второй шаг

> Пример 27

Оглядываясь назад, отметим, что для управления форматированием вывода наряду с представленными ранее средствами стандартной библиотеки C++ теперь можно воспользоваться и такими средствами класса ***basic_ios*** и его базового класса ***ios_base,*** как набор флагов формата и операций для их явной установки и сброса. По-разному можно относиться к необходимости использовать флаги для управления состоянием потока. С одной стороны, эти приемы проверены временем, а с другой стороны, их можно считать несколько старомодными, зная, что именно манипуляторы стали теми современными средствами, которые избавили пользователя от необходимости иметь дело с состоянием потока посредством флагов.

Итак, представим первую программу для иллюстрации форматного вывода целых литералов с использованием операций явной установки и сброса флагов формата:

```
// C++ Флаги формата
#include <iostream>
int main ()
{
using namespace std;
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::oct, ios_base::basefield);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::hex, ios_base::basefield);
cout << 1 << '\t' << 037777777777 << '\f << 0xffffffff << endl;
cout.unsetf(ios_base::basefield);
cout << 1 << '\t* << 037777777777 << ’\t' << 0xffffffff << endl;
cout.setf(ios_base::showbase);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::oct, ios_base:rbasefield);
cout << 1 << '\t' << 037777777777 << ’\t' << 0xffffffff << endl;
cout.setf(ios_base::hex, ios_base::basefield);
cout << 1 << ’\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::uppercase);
cout << 1 << '\t' << 037777777777 << '\t’ << 0xffffffff << endl;
cout.unsetf(ios_base:ruppercase); cout.unsetf(ios_base::showbase);
cout << 1 << >\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::oct, ios_base:rbasefield);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout.setf(ios_base::dec, ios_base:rbasefield);
cout << 1 << '\t' << 037777777777 << ’\t’ << 0xffffffff << endl;
return 0;
}
```

```
1          4294967295          4294967295
1          37777777777         37777777777
1          ffffffff            ffffffff
1          4294967295          4294967295
1          4294967295          4294967295
01           037777777777        037777777777
0x1        0xffffffff          0xffffffff
0x1        0xXFFFFFFF          0xFFFFFFF
1          ffffffff            ffffffff
1          37777777777         37777777777
1          4294967295          4294967295
```

Поначалу отметим, что все флаги формата, кроме трех, — однобитовые. Например, из представленных флагов только один не однобитовый — флаг ***basefield*** (0x004a). Флаг ***basefield*** составлен при помощи логической операции ИЛИ из трех флагов: ***dec*** (0x0002), ***hex*** (0x0008) и ***oct*** (0x0040). Назначение флага ***basefield*** — служить маской при установке опций, связанных с основанием системы счисления ***{dec, hex, oct).*** Такими же масками являются еще два неоднобитовых флага — ***adjustfield*** (OxOOЬO) и ***fioatfield*** (0x0104). Далее будет сказано о составе и назначении этих флагов.

Как видим, для операций установки и сброса флагов формата здесь используются две функции-члена класса ***ios base:*** перегруженная ***setf()*** и обычная ***unsetj()***. Обе эти функции для чтения и установки флагов используют перегруженную функцию-член ***flags*** 0: без аргумента вызова — для чтения флагов, а с одним аргументом вызова — для установки флагов. Функция-член ***setf()*** с одним аргументом вызова к считанному набору флагов добавляет новый флаг при помощи логической операции ИЛИ, оставляя тем самым старые флаги неизменными. Заметим, что такой прием годится лишь тогда, когда опция формата управляется однобитовым флагом. Флаг ***uppercase*** (0x4000) позволяет переключиться с нижнего регистра на верхний при выводе шестнадцатеричного представления целого литерала, а с помощью флага ***showbase*** (0x0200) можно узнать, какая система счисления используется при выводе.

Если опция формата управляется не однобитовым флагом, к примеру, флагом ***basefield,*** то установка нового флага, связанного с основанием системы счисления, возможна лишь после сброса старого флага с помощью маски ***basefield.*** Такой прием осуществляется благодаря вызову функции ***setf()*** с двумя аргументами. Новый флаг добавляется при помощи логической операции ИЛИ к считанному набору флагов, после того как в нем при помощи логических операций НЕ и И не будет сброшен старый флаг с помощью маски ***basefield.*** Отметим, что после установки все опции формата сохраняют свое состояние, пока не будут сброшены.

Отметим также, что сброс опции формата, связанной с основанием системы счисления, возможен и с помощью такой инструкции:

```
cout.setf(ios_base::fmtflags(0), ios_base::basefield);
```

Здесь “флагом” является литерал ***0,*** тип которого ***fmtflags*** явно конструируется с помощью оператора конструирования значения.

> **Пример 28**

Продолжим иллюстрацию форматного вывода целых литералов с использованием операций явной установки и сброса флагов формата:

```
// C++ Флаги формата #include <iostream> int main ()
{
using namespace std;
cout << 123 << '\t' << -123 << endl;
cout.setf(ios_base::showpos);
cout << 123 << '\t' << -123 << endl;
cout.width(5);
cout << 123 << '|' << '\t';
cout.width(5);
cout << -123 << '|' << endl;
cout.setf(ios_base::internal, ios_base::adjustfield);
cout.width(5);
cout << 123 << '|' << '\t' ;
cout.width(5);
cout << -123 << '|' << endl;
cout.setf(ios_base:rleft, ios_base::adjustfield);
cout.width(5);
cout << 123 << '|' << ’\t’;
cout.width(5);
cout << -123 << '|' << endl;
cout.setf(ios_base::right, ios_base::adjustfield);
cout.width(5);
cout << 123 << '|' << ' \t';
cout.width(5);
cout << -123 << '|' << endl;
cout.unsetf(ios_base::showpos);
cout << 123 << '\t' << -123 << endl;
return 0;
}
```

Результат работы программы:

```
123     -123
+123    -123
+123|   -123|
+123|   -123|
+123|   -123|
+123|   -123|
123     -123
```

Флаг ***showpos*** (0x0800) позволяет выводить явный знак + при выводе десятичного представления целого литерала. В форматной строке функции ***printf()*** флагу ***showpos*** соответствует знак +, который может следовать сразу за знаком ***%*** в спецификации преобразования.

Флаг ***adjustfield*** (OxOOЬO) составлен при помощи логической операции ИЛИ из трех флагов: ***internal*** (0x0010), ***left*** (0x0020) и ***right*** (0x0080). При выводе десятичного представления целого литерала флаг ***internal*** позволяет управлять отступом справа от знака, а флаги ***left*** и ***right*** позволяют управлять выравниванием, соответственно, по левому и правому краю поля вывода. В форматной строке функции ***printf()*** флагу ***internal*** нет соответствия, флагу ***left*** соответствует знак -, который может следовать сразу за знаком ***%*** в спецификации преобразования, а флаг ***right*** подразумевается по умолчанию. Назначение флага ***adjustfield*** — служить маской при установке опций, связанных с выравниванием внутри поля вывода ***{internal, left, right).*** Эти установки выравнивания определяются функцией-членом ***width()*** класса ***iosbase*** и не влияют на другие части состояния потока вывода. Так как обращение к функции ***width()*** влияет только на непосредственно следующую за ней операцию вывода, то для обозначения поля вывода здесь можно указать лишь его правую границу, а чтобы указать также и его левую границу, то перед обращением к функции ***width()*** понадобится еще одна операция вывода.

Продолжая обсуждение флагов формата и соответствующих им спецификаций преобразования в форматной строке функции ***printf(),*** добавим к сказанному, что флагу ***showbase*** соответствует знак #, который может следовать сразу за знаком ***%*** в спецификации преобразования. Что касается флагов ***dec, hex*** и ***oct ,*** то каждому из них, как это было видно из примеров, соответствует своя уникальная спецификация преобразования: ***%d, %x*** и ***%o.*** Совместному действию флагов ***uppercase*** и ***hex,*** как это было видно там же, соответствует уникальная спецификация преобразования ***%X.***

Прежде чем перейти к версии программы в стиле С, отметим, что использование функции ***printf()*** небезопасно в том смысле, что проверки типа ее аргументов вызова никак не производится. Это означает, что любое несоответствие типа аргумента и его предполагаемой спецификации преобразования в форматной строке может привести к неожиданному побочному эффекту, а порой и к непредсказуемому результату.

Однако функция ***printf()*** обеспечивает чрезвычайную гибкость, поэтому многим пользователям, знакомым со стилем С, трудно от нее отказаться. Мало того, поскольку программы на С и С++ зачастую смешиваются, потоки ввода-вывода С++ совмещаются с функциями ввода-вывода С. Ввод-вывод в стиле С и С++ в программе может смешиваться посимвольно, например, вызов функции-члена ***syncwithstdio()*** класса ***ios base*** до первой операции с потоками ввода-вывода С++ гарантирует, что операции ввода-вывода С и С++ будут иметь общие буферы. Ввод-вывод в стиле С и С++ в программе можно и “разъединить”, если функцию ***sync with stdio()*** вызвать с аргументом ***false,*** что в некоторых реализациях даже может привести к улучшению производительности.

Подытоживая сказанное и заглядывая вперед, отметим, что главное преимущество функций потокового вывода над функцией ***printf()*** заключается в том, что потоки безопасны с точки зрения типов и имеют общий стиль для вывода объектов как встроенных типов, так и типов, определяемых пользователем.

> Пример 29

Продолжим иллюстрацию форматного вывода целых литералов с использованием только одной функции ***printf()*** рассматривая ее в качестве альтернативы функциям потокового вывода для операций установки и сброса флагов формата:

```
// C++ Управление форматированием вывода в стиле C #include <iostream>
int main ()
{
 using namespace std;
 printf("%d\t%d\t\t%d\n", 1, 037777777777, 0xffffffff);
 printf("%#o\t%#o\t%#o\n", 1, 037777777777, 0xffffffff);
 printf("%#x\t%#x\t%#x\n", 1, 037777777777, 0xffffffff);
 printf("%#X\t%#X\t%#X\n\n", 1, 037777777777, 0xffffffff);
 printf("%d\t%d\n", 123, -123);
 printf("%+d\t%+d\n", 123, -123);
 printf("|%+5d|\t|%+5d|\n", 123, -123);
 printf("|%-+5d|\t|%-+5d|\n", 123, -123);
 return 0;
}
```

Результат работы программы:

```
1          -1                  -1
01         037777777777        037777777777
0x1        0xffffffff          0xffffffff
0X1        FFFFFFFF            0xFFFFFFFF

123        -123
+123       -123
| +123|    | -1231
1+123     |1-123 |
```

> Пример 30

Продолжая иллюстрацию форматного вывода целых литералов, от флагов формата и операций для их установки и сброса перейдем к манипуляторам с аргументами, которые определены в стандартном заголовочном файле **<*iomanip*>**:

```
// C++ Манипуляторы стандартной библиотеки
#include <iostream>
#include <iomanip>
int main()
{
 using namespace std;
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << setbase(8);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << setbase(l0);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << setbase(16);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << resetiosflags(ios_base::basefield);
cout << setiosflags(ios_base::showbase);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << setiosflags(ios_base::oct);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << resetiosflags(ios_base::oct);
cout << setiosflags(ios_base::hex);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << setiosflags(ios_base::uppercase);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
cout << resetiosflags(ios_base::showbase);
cout << resetiosflags(ios_base::uppercase);
cout << 1 << '\t' << 037777777777 << '\t' << 0xffffffff << endl;
return 0;
}
```

Результат работы программы:

```
1          4294967295          4294967295
1          37777777777        37777777777
1          4294967295          4294967295
1          ffffffff            ffffffff
1          4294967295          4294967295
01         037777777777           037777777777
0x1        0xffffffff          0xffffffff
0X1        0xFFFFFFFF          0xFFFFFFFF
1          ffffffff            ffffffff
```

> Пример 31

Представим теперь программу для иллюстрации форматного вывода литералов с плавающей точкой, начиная с использования операций явной установки и сброса флагов формата:

```
// C++ Флаги формата #include <iostream> int main ()
{
 using namespace std;
 cout << 1E1 << "\t\t" << -IE-2 << endl;
 cout.setf(ios_base::scientific, ios_base::floatfield);
 cout << 1E1 << '\t' << -IE-2 << endl;
 cout.setf(ios_base::uppercase);
 cout << 1E1 << '\t' << -IE-2 << endl;
 cout.unsetf(ios_base::uppercase);
 cout << 1E1 << ’\t' << -IE-2 << endl;
 cout.setf(ios base::fixed, ios_base::floatfield);
 cout << 1E1 << '\t’ << -IE-2 << endl;
 cout.unsetf(ios_base::fixed);
 cout << 1E1 << "\t\t" << -IE-2 << endl;
 return 0;
}
```

Результат работы программы для платформы Windows:

```
10                  -0.01
1.000000e+0001      -1.000000e-002
1.000000E+001       -1.000000E-002
1.000000e+001       -1.000000e-002
10.000000           -0.010000
10                  -0.01
```

Флаг ***floatfield*** (0x0104) составлен при помощи логической операции ИЛИ из двух флагов: ***fixed*** (0x0004) и ***scientific*** (0x0100). Назначение флага ***floatfield*** — служить маской при установке опций формата вывода ***(fixed***, ***scientific).*** Для управления состоянием научного формата вместе с флагом ***scientific*** может использоваться также и флаг ***uppercase***, если необходимо соответствие спецификации преобразования ***%E.***

Отметим, что сброс опций формата вывода возможен также и с помощью такой инструкции:

```
cout.setf(ios_base::fmtflags(0), ios_base::floatfield);
```

Как и ранее, здесь “флагом” является литерал ***0,*** тип которого ***fmtflags*** явно конструируется с помощью оператора конструирования значения

> **Пример 32**

Продолжим иллюстрацию форматного вывода литералов с плавающей точкой с использованием операций явной установки и сброса флагов формата:

```
// C++ Флаги формата #include <iostream> int main()
{
using namespace std;
cout << 1E1 << '\t' << -IE-2 << endl;
cout.setf(ios_base:rshowpos);
cout << 1E1 << '\t' << -IE-2 << endl;
cout.width(6);
cout << 1E1 << '|' << '\t' ;
cout.width(6);
cout << -IE-2 << '|' << endl;
cout.setf(ios_base::internal, ios_base::adjustfield);
cout.width(6);
cout << 1E1 << '|' << '\ t' ;
cout.width(6);
cout << -IE-2 << '|' << endl;
cout.setf(ios_base::left, ios_base::adjustfield);
cout.width(6);
cout << 1E1 << '|' << '\t' ;
cout.width(6);
cout << -IE-2 << '|' << endl;
cout.setf(ios_base::right, ios_base::adjustfield);
cout.width(6);
cout << 1E1 << '|' << '\t' ;
cout.width(6);
cout << -IE-2 << '|' << endl;
cout.unsetf(ios_base::showpos);
cout << 1E1 << '\t' << -IE-2 << endl;
return 0;
}
```

Результат работы программы:

```
10         -0.01
+10        -0.01
+10|      -0.01|
+10|       -O.O1|
+10|       -0.01|
+10|      -0.01|
10         -0.01
```

> **Пример 33**

Продолжим иллюстрацию форматного вывода литералов с плавающей точкой с использованием операций явной установки и сброса флагов формата:

```
// C++ Флаги формата #include <iostream> int main ()
{
  using namespace std;
  cout << 1.2 << ”\t\t" << 12.3 << endl;
  cout.setf(ios_base::showpoint);
  cout << 1.2 << "\t\t" << 12.3 << endl;
  cout << 123.4 << "\t\t" << 1234.5 << endl;
  cout << 12345.6 << "\t\t" << 123456.7 << endl;
  cout << 1234567.8 << '\t' << 12345678.9 << endl;
  cout.unsetf(ios_base::showpoint);
  cout << 0.1 << "\t\t" << 0.012 << endl;
  cout.setf(ios_base::showpoint);
  cout << 0.1 << '\t' << 0.012 << endl;
  cout << 0.00123 << '\t' << 0.0001234 << endl;
  cout << 0.000012345 << '\t' << 0.00000123456 << endl;
  cout << 0.0000001234567 << '\t' << 0.000000012345678 << endl;
  return 0;
}
```

Результат работы программы для платформы Windows:

```
1.2                 12.3
1.20000             12.3000
123.400             1234.50
12345.6             123457.
1.23457e+006        1.23457e+007
0.1                 0.012
0.100000            0.0120000
0.00123000          0.000123400
1.23450e-005        1.23456e-006
1.23457e-007        1.23457e-008
```

При выводе десятичного представления литерала с плавающей точкой флаг ***showpoint*** (0x0400) позволяет управлять выводом незначащих нулей дробной части.

> **Пример 34**

Завершая иллюстрацию форматного вывода литералов с плавающей точкой, теперь перейдем к манипуляторам с аргументами:

```
// C++ Манипуляторы стандартной библиотеки
#include <iostream>
#include <iomanip>
int main()
{
using namespace std;
cout << 0.123456789 << endl;
cout << setprecision(9) << 0.123456789 << endl;
cout << setiosflags(ios_base::scientific);
cout << 0.123456789 << endl;
cout << setiosflags(ios_base::uppercase);
cout << 0.123456789 << endl;
cout << resetiosflags(ios_base::scientific);
cout << resetiosflags(ios_base::uppercase);
cout << 0.123456789 << endl;
cout << setprecision(4) << 0.123 << endl;
cout << setiosflags(ios_base::showpoint);
cout << 0.123 << endl;
cout << resetiosflags(ios_base::showpoint);
cout << setiosflags(ios_base::fixed);
cout << 0.123 << endl;
return 0;
}
```

Результат работы программы для платформы Windows:

```
0.123457
0.123456789
1.234567890e-001
1.234567890E-001
0.123456789
0.123
0.1230
0.1230
```

Отметим, что сброс опций формата вывода возможен также и с помощью такой инструкции:

```
cout << resetiosflags(ios_base::fmtflags(0xffffffff));
```

Здесь, как и ранее, тип литерала явно конструируется с помощью оператора конструирования значения.

Стандартные операторы

> Пример 35

Получив представление о механизмах управления форматированием вывода в C++, перейдем теперь к обзору лишь тех его стандартных операторов, которые могут быть проиллюстрированы с помощью операций вывода. Из унарных операторов, которые упоминались ранее, отметим :: (разрешение области видимости), ++ (префиксный инкремент), * (разыменование), — (минус) и оператор конструирования значения, а из бинарных операторов отметим . (выбор члена), [] (доступ по индексу), () (вызов функции), + (сложение), — (вычитание) и < (меньше). Следует также отметить, что не каждому из перечисленных операторов нашлось место для комментария.

Поначалу выясним размеры арифметических типов для 32-разрядной архитектуры при помощи оператора ***sizeof.***

```
// C++ Стандартные операторы #include <iostream> int main ()
{
using namespace std;
cout << sizeof(bool) << '\t' << sizeof(char) << endl;
cout << sizeof(short int) << '\t'
      << sizeof(int) << '\t'
      << sizeof(long int) << endl;
cout << sizeof(float) << '\t'
      << sizeof(double) << '\t'
      << sizeof(long double) << endl;
return 0;
}
```

Результат работы программы для платформы Windows:

```
1       1
2       4       4
4       8       8
```

Результат работы программы для платформы Linux:

```
1       1
2       4       4
4       8       12
```

Напомним здесь, что в C++ размеры объектов или типов выражаются в единицах размера ***char.***

> Пример 36

Продолжая обзор стандартных операторов C++, здесь и далее рассмотрим лишь те из них, которые могут быть использованы при составлении арифметических и логических выражений, где операндами будут только целые литералы.

Поначалу представим программу для иллюстрации форматного вывода значений арифметических выражений, составленных с помощью десятичных литералов и унарных операторов (дополнение, отрицание, минус, плюс):

```
// C++ Стандартные операторы #include <iostream> int main()
{
using namespace std;
cout << oct << -2 << '\t' << —2 << endl;
cout << oct << -1 << '\t' << ~-1 << endl;
cout << oct << 0 << "\t\t" << ~0 << endl;
cout << oct << 1 << "\t\t" << ~+1 << endl;
cout << oct << 2 << "\t\t" << ~2 << endl;
cout << oct << -1 << '\t' << !-1 << endl;
cout << oct << 0 << "\t\t" << !0 << endl;
cout << oct << 1 << "\t\t" << !+1 << endl;
cout << oct << 2 << "\t\t" << !2 << endl;
return 0;
}
```

Результат работы программы:

```
37777777776         1
37777777777         0
0                   37777777777
1                   37777777776
2                   37777777775
37777777777         0
0                   1
1                   0
2                   0
```

Отметим здесь, что оператор ~ (дополнение) является побитовым логическим оператором. Побитовое дополнение — это дополнение до одного (обратный код). Оператор ! (отрицание) — это логический оператор НЕ. Напомним, что результат логического отрицания приводится к логическому типу ***bool.*** Напомним также, что объекты типа ***bool*** могут принимать только одно из двух значений: ***истина {true)*** или ***ложь {false).*** Как видим, при выводе ***true*** имеет значение 1 при преобразовании к целому типу, а ***false*** — значение 0.

> Пример 37

Теперь представим программу для иллюстрации форматного вывода значений арифметических выражений, составленных с помощью десятичных литералов и бинарных операторов целочисленной арифметики (умножение, деление, деление по модулю, сложение, вычитание):

```
// C++ Стандартные операторы #include <iostream> int main ()
{
 using namespace std;
 cout << 2*3 << '\t' << 2*+3 << '\t' << +2*+3 << '\t'
 <<         2*-3 << '\t'  << +2*-3  << '\t'
 <<         -2*3 << '\t'  << -2*+3  << '\t' <<  -2*-3 << endl;
 cout << 3/2 << '\t' << 3/+2 << '\t' << +3/+2 << '\t'
 <<         3/-2 << '\t'  << +3/-2  << '\t'
 <<         -3/2 << '\t'  << -3/+2  << '\t' <<  -3/-2 << endl;
 cout << 2%3 << '\t' << 2%+3 << '\t' << +2%+3 << '\t'
 <<         2%-3 << '\t'  << +2%-3  << '\t'
 <<         -2%3 << '\t'  << -2%+3  << '\t' <<  -2%-3 << endl;
 cout << 3+2 << '\t' << 3+(+2) << '\t' << +3+(+2) << '\t'
 <<         3+-2 << '\t' << +3+-2 << '\t'
 <<         -3+2 << '\t' << -3+(+2) << '\t'  <<-3+-2 << endl;
 cout  <<  3-2 << '\t' << 3—1-2 << '\t' << +3-+2    <<       '\t'
 <<         3-(-2) << '\t' << +3-(-2) << '\t'
      << -3-2 << '\t' << -3-+2 << '\t' << -3- (-2) << endl;
return 0;
}
```

Результат работы программы:

```
6         6         6        −6     −6     −6     -6     6
1         1         1        -1     -1     -1     -1     1
2         2         2         2      2     -2     -2     -2
5         5         5         1      1     -1     -1     -5
1          1        1         5      5     -5     -5     -1
```

> Пример 38

А теперь представим программу для иллюстрации форматного вывода значений арифметических выражений, составленных с помощью десятичных литералов и бинарных операторов сдвига (сдвиг влево, сдвиг вправо):

```
// C++ Стандартные операторы #include <iostream> int main ()
{
 using namespace std;
 cout << 5 << "\t\t" << (5>>1) << "\t\t" << (5>>2)
      << "\t\t" << (5>>3) << endl;
      cout << -5 << "\t\t" << (-5>>1) << "\t\t" << (-5>>2)
      << "\t\t" << (— 5>>3) << endl;
      cout << oct << -5 << '\t' << (—5>>1) << '\t' << (-5>>2)
      << '\t' << (-5>>3) << endl;
      cout << dec << 1 << "\t\t" << (1 <<1) << "\t\t" << (1 <<2)
      << "\t\t" << (1 <<3) << endl;
      cout << -1 << "\t\t" << (-1 <<1) << ″\t\t" << (-1 <<2)
      << "\t\t" << (-1 <<3) << endl;
      cout << oct << -1 << '\t' << (-1 <<1) << '\t' << (—1 <<2)
      << '\t' << (-1 <<3) << endl;
      return 0;
}
```

Результат работы программы:

```
5                   2                   10
-5                  -3                  -2                  -1
37777777773         37777777775         37777777776         37777777777
12                                      4                   8
-1                  -2                  -4                  -8
37777777777         37777777776         37777777774         37777777770
```

Отметим здесь, что оператор << (сдвиг влево) и оператор >> (сдвиг вправо), как и оператор ~ (дополнение), тоже принадлежат к побитовым логическим операторам. Напомним, что побитовые логические операторы применяются только к объектам целых типов, а результатом побитовых логических операций тоже являются целые.

Отметим также, что рассматриваемые здесь операторы сдвига принадлежат к типу операторов арифметического сдвига, называемых так потому, что они осуществляют операции целочисленной арифметики — умножения и деления на число, кратное 2.

> Пример 39

Продолжая обзор, перейдем теперь к иллюстрации форматного вывода значений логических выражений, составленных с помощью десятичных литералов и бинарныхоператоров отношения (меньше, меньше или равно, больше, больше или равно, равно, не равно):

```
// C++ Стандартные операторы #include <iostream> int main ()
{
  using namespace std;
  cout << sizeof (1<2) << endl;
  cout  <<   (1<2)  <<'\t'  << (1<=2) << endl;
  cout  <<   (1>2)  <<'\t'  << (1>=2) << endl;
  cout << (1==2) << '\t' << (1!=2) << endl;
  return 0;
}
```

Результат работы программы:

```
1
1         1
0         0
0         1
```

Как видим, результат логических операций отношения приводится к логическому типу ***bool.*** Напомним, что объекты типа ***bool*** могут принимать только одно из двух значений: ***истина (true)*** или ***ложь (false).*** Ранее было сказано, что по определению, ***true*** имеет значение 1 при преобразовании к целому типу, а ***false*** — значение 0.

> Пример 40

А теперь представим программу для иллюстрации форматного вывода значений логических выражений, составленных с помощью десятичных литералов и бинарных побитовых логических (булевых) операторов (побитовое И, побитовое исключающее ИЛИ, побитовое ИЛИ):

```
// C++ Стандартные операторы
#include <iostream>
int main ()
{
  using namespace std;
  cout << oct << (037777777777&037777777777) << '\t'
       << (037777777777^037777777777) << "\t\t"
       << (037777777777|037777777777) << endl;
   cout << (037777777777&037777777707) << '\t'
       << (037777777777^037777777707) << "\t\t"
       << (037777777777 1037777777707) << endl;
   cout << (037777777777&00) << "\t\t" << (037777777777^00) << '\t'
       << (037777777777|00) << endl;
   cout << (01&01) << "\t\t" << (01^01) << "\t\t"
       << (01|01) << endl;
   cout << (01&00) << "\t\t" << (01^00) << "\t\t"
       << (01|00) << endl;
return 0;
}
```

Результат работы программы:

```
37777777777     0                   37777777777
37777777707     70                  37777777777
0               37777777777         37777777777
1               0                     1
0                 1                   1
```

Отметим здесь, что для записи булевых операторов наряду с русскоязычной зачастую используется также и англоязычная нотация. Так, например, булев оператор НЕ (логическое отрицание) — это NOT, булев оператор И (логическая конъюнкция, логическое умножение) — это AND, булев оператор ИЛИ (логическая дизъюнкция, логическое сложение) — это OR, а булев оператор исключающее ИЛИ — это XOR (от слов ***exclusive or).*** Кроме того, в булевой алгебре для записи булевых операторов используется еще и такая нотация для знаков операций, где, например, оператору НЕ соответствует знак операции ⌐ перед операндом (или знак — над операндом), оператору И — знак операции ^ (или •), оператору ИЛИ — знак операции **v** (или +), а оператору исключающее ИЛИ — знак операции ⊕.

> Пример 41

И в завершение этого обзора представим программу для иллюстрации форматного вывода значений логических выражений, составленных с помощью десятичных литералов и бинарных логических операторов (логическое И, логическое ИЛИ):

```
// C++ Стандартные операторы #include <iostream> int main ()
{
 using namespace std;
 cout << oct << (037777777777&&037777777777) << '\t'
      << (037777777777||037777777777)  <<endl;
 cout << (037777777777&&01) << '\t'
       << (037777777777||01)  <<endl;
 cout << (037777777777&&00) << '\t'
      << (037777777777||00)  <<endl;
 cout << (01&&01) << '\t' << (01||01) << endl;
 cout << (01&&00) << '\t' << (01||00) << endl;
return 0;
}
```

Результат работы программы:

```
1          1
1          1
0          1
1          1
0          1
```

Отметим, что логические операторы ***&&,*** || и ! возвращают ***true*** или ***false*** и в основном используются в тех инструкциях, где логические выражения необходимы в операциях сравнения.

Инструкции

> Пример 42

Прежде чем продолжить обзор стандартных операторов C++, поначалу перейдем к иллюстрации механизма объявления таких объектов программы, как переменные и константы, с помощью инструкций-объявлений. Объявление вводит имя объекта программы в пространство имен. Объявления не только связывают тип с именем объекта, но почти всегда являются еще и определениями, т.е. они определяют некую сущность, которая соответствует имени. Например, для переменных этой сущностью является объект, которому выделено подходящее количество памяти.

Представим вначале программу, где в глобальной области видимости объявляются переменные типа ***int*** (глобальные переменные), которые размещаются в статической памяти и по умолчанию инициализируются значением 0:

```
// C++ Инструкции
#include <iostream>
// Статические объекты — переменные
int var01;
int var02 = 102;
int var03(103);
int var04, var05;
int var06 = 106, var07;
int var08, var09 = 109;
int var10 = 110, var11 = 111;
int var12(112), var13;
int var14, var15(115);
int var16(116), var17(117);
int var18 = 118, var19(119);
int var20(120), var21 = 121;
int main ()
{
  using namespace std;
cout    << var01 << '\t' << var04 << '\t' << var05 << '\t'
       << var07 <<'\t' << var08  <<'\t'  <<var13 <<'\t'
        << var14 <<    endl;
cout << var02 << '\t' <<var03 << '\t' << var06 << '\t'
     << var09 << '\t' <<var10 << '\t' << var11 << '\t'
     << var12 << endl;
cout << var15 << '\t' <<var16 << '\t' << var17 << '\t'
     << var18  << '\t' << var19  <<'\t'  << var20 <<'\t'
     << var21  << endl;
  return 0;
}
```

Результат работы программы:

```
0       0       0       0       0       0         0
102     103     106     109     110     111       112
115     116     117     118     119     120       121
```

Напомним, что объявление может состоять из четырех частей: необязательного спецификатора, базового типа, объявляющей части и, возможно, инициализатора.

Объявленные в глобальном пространстве имен переменные программы называют статическими, так как они размещаются в статической памяти во время компиляции, т.е. по фиксированному адресу. По умолчанию эти переменные инициализируются значением 0 соответствующего типа.

Как видим, для инициализации переменных в C++ можно использовать одну из двух форм. Первая форма инициализации, традиционная для C, связана с операцией присваивания инициализирующего значения. Инициализирующее значение принято называть инициализатором.

Вторая форма инициализации связана с механизмом конструирования значения, принятого в C++ для инициализации объектов класса при помощи его конструкторов, являющихся функциями-членами класса. Инициализаторы передаются конструктору класса как аргументы вызова. Таковым, например, здесь является конструктор по умолчанию, который генерируется компилятором для конструирования значения объектов встроенного типа ***int.***

В объявлении переменной может отсутствовать инициализирующая часть, в этом случае либо полагаются на умолчание, либо откладывают инициализацию до первой попытки использования этой переменной, что не всегда безопасно.

Отметим, что объявление в C++ разрешается помещать не только в том месте, где допустима инструкция. В частности, локальную переменную лучше объявлять в тот момент, когда ей надо присвоить значение, чтобы исключить попытки использования локальной переменной до момента ее инициализации. Так, одним из элегантных применений этой концепции является объявление переменной в условии, например, инструкции-выбора ***if*** или в части инициализации инструкции-итерации ***for***, но об этом позже.

Здесь и далее пока будут рассматриваться только глобальные имена и локальные имена в области видимости функции ***main().*** Напомним, что область видимости глобальных имен простирается от места их объявления до конца файла, содержащего объявление, а область видимости имен, объявленных внутри функции, начинается с места их объявления и заканчивается в конце блока, в котором эти имена объявлены. Напомним также, что блоком называют составную-инструкцию, которая заключает в фигурных скобках {} последовательность инструкций, в том числе и пустую. Тело функции можно называть как блоком, так и составной-инструкцией.

> Пример 43

Итак, переменные программы, объявленные в глобальной области видимости, по умолчанию размещаются в статической памяти. Для явного указания компилятору на размещение переменной в статической памяти служит спецификатор ***static.***

"Игнорируя" советы Страуструпа об ограниченном использовании глобальных переменных, а также об использовании спецификатора ***static*** только внутри функций и классов, представим программу, где глобальные переменные будут явно объявлены как ***static***:

```
// C++ Инструкции #include <iostream>
// Статические объекты — переменные
static int var01;
static int var02 = 102;
static int var03(103);
static int var04, var05;
static int var06 = 106, var07;
static int var08, var09 = 109;
static int var10 = 110, var11 = 111;
static int var12(112), var13;
static int var14, var15(115);
static int var16(116), var17 (117);
static int var18 = 118, var19(119);
static int var20(120), var21 = 121;

int main()
{
  using namespace std;
  cout  <<  var01  << '\t'  << var02  << '\t'  << var03 << '\t'
 <<         var04  << '\t'  << var05  << '\t'  << var06 << '\t'
 <<         var07  <<    endl;
  cout  <<  var08  << '\t'  << var09  << '\t'  << var10 << '\t'
 <<         var11  << '\t'  << var12  << '\t'  << var13 << '\t'
 <<         var14  <<    endl;
  cout  <<  var15  << '\t'  << var16  << '\t'  << var17 << '\t'
 <<         var18  << '\t'  << var19  << '\t'  << var20 << '\t'
 <<         var21  <<    endl;
  return 0;
}
```

Результат работы программы:

```
0         102       103      0         0         106          0
0         109       110      111       112       0            0
115       116       117      118       119       120        121
```

> Пример 44

Локальная (автоматическая) переменная инициализируется в момент выполнения строки, содержащей ее определение. По умолчанию это, например, происходит при каждом вызове функции, где есть объявление такой переменной, при этом каждый раз в стеке создается новый объект. Если внутри функции переменная объявлена как ***static,*** она инициализируется только при первом вызове функции, а для хранения ее значения используется единственный, статически размещенный в памяти объект.

Невзирая на то, что функция ***main()*** вызывается всего лишь один раз, представим программу, где локальные переменные в функции ***main()*** будут объявлены как ***static:***

```
// C++ Инструкции #include <iostream> int main ()
{
 using namespace std;
// Статические объекты — переменные
static int var0l;
static int var02 = 102;
static int var03(103);
static int var04, var05;
static int var06 = 106, var07;
static int var08, var09 = 109;
static int var10 = 110, var1l = 111;
static int var12(112), var13;
static int var14, var15(115);
static int var16(116), var17(117);
static int var18 = 118, var19(119);
static int var20(120), var21 = 121;
 cout  <<   var0l  << '\t'  << var02  << '\t'  << var03 << '\t'
 <<         var04  << '\t'  << var05  << '\t'  << var06 << '\t'
 <<         var07  <<    endl;
 cout  <<   var08  << '\t'  << var09  << '\t'  << var10 << '\t'
 <<         var11  << '\t'  << var12  << '\t'  << var13 << '\t'
 <<         var14  <<    endl;
 cout  <<   var15  << '\t'  << var16  << '\t'  << var17 << '\t'
 <<         var18  << '\t'  << var19  << '\t'  << var20 << '\t'
 <<         var21  <<    endl;
return 0;
}
```

Результат работы программы:

```
0          102       103       0         0         106          0
0          109       110       111       112       0            0
115        116       117       118       119       120        121
```

> **Пример 45**

К любой переменной в объявлении может быть применен квалификатор ***const*** для указания того, что в текущей области видимости ее значение не может измениться. Чтобы переменная стала константной, в объявление необходимо добавить не только квалификатор ***const,*** но также и инициализатор константы. Отметим, что ***const*** всего лишь модифицирует тип, т.е. ограничивает возможное использование объекта, но не указывает способ размещения константного объекта.

Поначалу представим программу, где в глобальной области видимости будут объявлены константы типа ***int*** (глобальные константы), которые по умолчанию размещаются в статической памяти:

```
// C++ Инструкции
#include <iostream>
// Статические объекты – константы
const int const01 = 201;
const int const02(202);
const int const03 = 203, const04 = 204;
const int const05 = 205, const06(206);
const int const07(207), const08 = 208;
const int const09(209), const10(210);
int main()

{
  using namespace std;
  cout  <<  const01  << '\t'  <<  const02 << '\t'  <<  const03  << '\t'
 <<         const04  << '\t'  <<  const05 <<    endl;
  cout  <<  const06  << '\t'  <<  const07 << '\t'  <<  const08  << '\t'
 <<         const09  << '\t'  <<  const10 <<    endl;
  return 0;
}
```

Результат работы программы:

```
201       202       203      204         205
206       207       208      209         210
```

Здесь необходимо отметить, что компилятор может по-разному воспользоваться свойством константности объекта программы. Так, например, если инициализатор константы является константным выражением, его можно вычислить во время компиляции. Более того, если компилятору известны все случаи использования константы, он может не выделять под нее память. Отметим также, что в объявлении константного компонента данных класса может отсутствовать инициализатор, в этом случае инициализация такого компонента класса может быть осуществлена только конструктором со списком инициализации.

> Пример 46

Как и в случае объявления переменной, для явного указания на размещение константы в статической памяти необходимо воспользоваться спецификатором ***static.*** Представим теперь программу, где глобальные константы будут явно объявлены как ***static:***

```
// C++ Инструкции
#include <iostream>
// Статические объекты – константы
static const int const0l = 201;
static const int const02(202);
static const int const03 = 203, const04 = 204;
static const int const05 = 205, const06(206);
static const int const07(207), const08 = 208;
static const int const09(209), const10(210);

int main()
{
  using namespace std;
cout  <<   const0l   << '\t'  <<  const02 << '\t'  << const03  <<  '\t'
 <<        const04   << '\t'  <<  const05 <<    endl;
cout  <<   const06   << '\t'  <<  const07 << '\t'  << const08  <<  '\t'
 <<        const09   << '\t'  <<  const10 <<    endl;
return 0;
}
```

Результат работы программы:

```
201        202      203      204        205
206        207      208      209        210
```

> Пример 47

Как и в случае объявления переменной, константа, объявленная внутри функции как ***static,*** инициализируется только при первом вызове функции, а для хранения ее значения используется единственный, статически размещенный в памяти объект.

По-прежнему, невзирая на то, что функция ***main()*** вызывается всего лишь один раз, представим программу, где локальные константы в функции ***main()*** будут объявлены как ***static:***

```
// C++ Инструкции #include <iostream> int main ()
{
using namespace std;
// Статические объекты — константы
static const int const01 = 201;
static const int const02(202);
static const int const03 = 203, const04 = 204;
static const int const05 = 205, const06(206);
static const int const07(207), const08 = 208;
static const int const09(209), const10(210);
cout  <<   const01  <<  '\t'  <<const02   <<'\t'  <<  const03  <<  '\t'
 <<        const04  <<  '\t'  <<const05   <<endl;
cout  <<   const06  <<  '\t'  <<const07   <<'\t'  <<  const08  <<  '\t'
 <<        const09  <<  '\t'  <<const10   <<endl;
return 0;
}
```

Результат работы программы:

```
201        202       203       204       205
206        207       208       209       210
```

> Пример 48

В автоматической памяти, называемой также “памятью в стеке”, по умолчанию располагаются локальные переменные и константы, а еще, как далее будет сказано, и аргументы функции, передаваемые по значению. Для явного указания компилятору на размещение локального объекта в автоматической памяти можно воспользоваться спецификатором ***auto.*** Напротив, если нет обращений к адресу локального объекта и при этом есть “свободные” регистры процессора, для явного указания компилятору на размещение такого объекта в регистровой памяти можно воспользоваться и спецификатором ***register.***

В завершение представим программу, где локальные переменные и константы в функции ***main()*** будут явно объявлены как ***auto*** и ***register.***

```
// C++ Инструкции #include <iostream> int main()
{
 using namespace std;
 // Автоматические объекты — переменные
 register int var01;
 register int var02 = 102;
 register int var03(103);
 auto int var04;
 auto int var05 = 105;
 auto int var06(106);
 int var07;
 int var08 = 108;
 int var09(109);
// Автоматические объекты — константы
register const int const01 = 201;
register const int const02(202);
auto const int const03 = 203;
auto const int const04(204);
const int const05 = 205;
const int const06(206);
cout << var02 << '\t' << var03 << ' \t' << var05 << '\t'
      << var06 << ’\t' << var08 << '\t’ << var09 << endl;
cout << const01 << '\t' << const02 << '\t' << const03 << '\t'
      << const04 << '\t' << const05 << '\t' << const06 << endl;
return 0;
}
```

Результат работы программы:

```
102         103       105       106       108       109
201         202       203       204       205       206
```

> Пример 49

Получив начальные сведения о механизме объявления переменных и констант в глобальной области видимости и области видимости функции ***main(),*** продолжим обзор стандартных операторов C++, привлекая наряду с инструкциями-выражениями теперь и инструкции-объявления. Как и ранее, обратимся к иллюстрации форматного вывода значений арифметических выражений, операндами которых на этот раз будут глобальные и локальные переменные.

Представим программу, где в арифметических выражениях будут использованы унарные операторы (разрешение области видимости, постфиксный инкремент и декремент, размер объекта, префиксный инкремент и декремент), а также бинарный оператор (простое присваивание):

```
// C++ Инструкции
#include <iostream>
int d = 0;
int main ()
{
 using namespace std;
 int a, b, c, d(1);
 double e(0.5);
 cout << (d + e) << endl;
 cout << ::d << '\t' << d << endl;
 cout << d << '\t';
 c — d++;
 cout << c << '\t' << d << ' \t';
 b = d--;
 cout << b << '\t' << d << endl;
 cout << sizeof ::d << '\t' << sizeof d << endl;
 cout << b << ' \ t';
 a = ++b;
 cout << a << '\t' << b << ' \ t';
 c = --b;
 cout << c << '\t' << b << endl;
return 0;
}
```

Результат работы программы:

```
1.5
0          1
1          1       2      2      1
4          4
2          3       3      2     2
```

Объявление имени в блоке может скрыть объявление этого имени в охватывающем блоке или глобальное имя. Для доступа к скрытому глобальному имени ***d***, как видим, здесь используется оператор разрешения области видимости.

В терминологии языка Fortran операторы, которые работают с операндами разного типа, реализуют так называемую смешанную арифметику. При вычислении значений арифметических выражений в смешанной арифметике для операндов родственных типов обычно полагаются на механизм неявного преобразования типов, например, тип с плавающей точкой в интегральный, и наоборот. Явное преобразование типа, называемое также приведением типа, иногда очень важно, однако, традиционно оно используется неоправданно часто и является основным источником ошибок.

В большинстве случаев в C++ нет нужды в явном преобразовании типов, если это все же так необходимо, можно воспользоваться либо оператором конструирования значения ***имя_типа(выражениё),*** либо одним из четырех именованных операторов приведения типа, либо оператором приведения типа ***(имя_типа)выражение.***

Среди именованных операторов отметим единственный оператор динамического приведения типов — оператор преобразования с проверкой во времени выполнения ***dynamic_cast<имя_типа>(выражение)*** — и три оператора статического приведения типов: для родственных типов — оператор преобразования с проверкой во времени компиляции ***static cast<имя_типа>(выражение);*** для несвязанных типов — оператор преобразования без проверки ***reinterpret_cast<имя_типа_(выражение***); для "снятия константности" с объектов — оператор ***const_cast_<имя_типа>(выражение).***

Оператор ++ (инкремент) используется для явного указания операции увеличения на 1 вместо менее явной записи +=, состоящей из комбинации операций сложения и присваивания. Например, инструкция-выражение с постфиксным инкрементом

```
c = d++;
```

эквивалентна последовательности из двух инструкций-выражений:

```
c = d;
d = d + 1;
```

а инструкция-выражение с префиксным инкрементом

```
a = ++b;
```

эквивалентна последовательности из двух инструкций-выражений:

```
b = b + 1;
a = b;
```

Оператор -- (декремент) используется для явного указания операции уменьшения на 1 вместо менее явной записи -=, состоящей из комбинации операций вычитания и присваивания. Например, инструкция-выражение с постфиксным декрементом

```
b = d--;
```

эквивалентна последовательности из двух инструкций-выражений:

```
b = d;
d = d — 1;
```

а инструкция-выражение с префиксным декрементом

```
c = --b;
```

эквивалентна последовательности из двух инструкций-выражений:

```
b = b - 1;
c = b;
```

> **Пример 50**

Конструкторы по умолчанию можно вызывать явно как для типов, определяемых пользователем, так и для встроенных типов, т.е. встроенные типы тоже имеют конструкторы по умолчанию. Результатом явного вызова конструктора по умолчанию для встроенных типов является 0, преобразованный в соответствующий тип, например, ***int()*** является значением ***int*** по умолчанию, т.е. равным 0.

Использование конструкторов по умолчанию для встроенных типов весьма важно при написании шаблонов, когда пользователь заранее не знает, какой же тип будет использоваться в качестве параметра шаблона — встроенный или определяемый пользователем.

Теперь представим программу, где инициализация глобальной переменной будет осуществляться благодаря вызову конструктора по умолчанию для встроенного типа ***int,*** а операции приведения операндов типа ***int*** к типу ***double*** в арифметических выражениях будут осуществляться с помощью операторов статического приведения типов — оператора конструирования значения, оператора преобразования с проверкой во времени компиляции и оператора приведения типа:

```
// C++ Инструкции
#include <iostream>
int a = int();
int main ()
{
  using namespace std;
  int a(1);
  double b(0.5);
  cout << ::a << '\t' << a << endl;
  cout << (double(a) + b) << '\t'
 << (static_cast<double>(a) + b) << '\t'
 << ((double)a + b) << endl;
  return 0;
}
```

Результат работы программы:

```
0           1
1.5         1.5         1.5
```

> **Пример 51**

Объявляющая часть объявления состоит из имени и, возможно, оператора объявления. Префиксные операторы объявления *и & используются при объявлении указателей и ссылок. Понятия ***указатель*** и ***ссылка*** могут означать как встроенный тип, так и сам объект этого типа. Так, ***имя_типа**** — это встроенный тип, называемый указателем на ***имя типа,*** а ***имя_типа&*** — это, соответственно, встроенный тип, называемый ссылкой на ***имя типа.***

Переменная типа ***имя типа**** содержит адрес объекта типа ***имя типа.*** Основной операцией над указателями является разыменование (или косвенное обращение), т.е. получение объекта, на который указывает указатель. “Привязка” указателя к объекту может быть выполнена как в момент его инициализации, так и в любое другое время с помощью операции присваивания адреса уже существующего объекта и при этом сколь угодно раз.

Переменная типа ***имя_типа&*** является альтернативным именем объекта, т.е. его псевдонимом. “Привязка” ссылки к объекту может быть выполнена только один раз и только в момент инициализации при ее объявлении.

Продолжая обзор стандартных операторов C++, представим программу, где в функции ***main()*** будут объявлены ссылка и указатель на ***int,*** а в операциях над объектами будут использованы унарные операторы (адрес, разыменование):

```
// C++ Инструкции #include <iostream> int main()
{
using namespace std;
 int a (3);
 int& referenceOnlnt = a;
 int* pointerOnlnt = &a;
 printf("%p\t%p\n", &a, pointerOnlnt);
 cout << &a << '\t' << pointerOnlnt << endl;
 printf("%d\t%d\t%d\n", a, referenceOnlnt, *pointerOnInt);
 cout << a << '\t' << referenceOnlnt << '\t'
      << *pointerOnInt << endl;
 a = 4;
 cout << a << '\t' << referenceOnlnt << '\t'
      << *pointerOnInt << endl;
 referenceOnlnt = 5;
 cout << a << '\t' << referenceOnlnt << '\t'
      << *pointerOnInt << endl;
 *pointerOnlnt = 3;
 cout << a << '\t' << referenceOnlnt << '\t'
      << *pointerOnInt << endl;
return 0;
```

Результат работы программы:

```
0076FDF4            0076FDF4
0x7 6fdf4           0x76fdf4
3         3          3
3         3          3
4         4          4
5         5          5
3         3          3
```

Как видим, для вывода адреса объекта с помощью библиотечной функции ***printf()*** в форматной строке используется спецификация преобразования ***%p*** (от слова ***pointer***).

Отметим, что указатели были задуманы с целью непосредственного отражения механизмов адресации компьютеров. Так, в C++ наименьшим объектом, который можно независимо разместить в памяти и на который можно независимо указать с помощью встроенных типов, является объект типа ***char.*** Встроенные указатели могут быть "нацелены" на любой статический, автоматический или динамический объект, в том числе и на функции.

Немалая роль отведена также указателям на типы, определяемые пользователем, т.е. классы. Во-первых, это связано с отражением идей объектно-ориентированной парадигмы программирования. Так, здесь и реализация концепции динамического полиморфизма, где по значению указателя на класс осуществляется вызов требуемой виртуальной функции в иерархии классов. Здесь и указатели на общедоступные нестатические компоненты класса, которые не могут адресовать никакого участка памяти, так как память выделяется не классу, а его объектам при их создании. Здесь и "умные указатели" или интеллектуальные указатели — объекты класса, которые ведут себя как встроенные указатели и, кроме того, выполняют некоторые действия, когда с их помощью осуществляется доступ к компонентам другого класса, на объекты которого эти указатели как раз и ссылаются.

Во-вторых, это связано с развитием принципов и идей обобщенной парадигмы программирования, где реализована концепция параметрического полиморфизма на основе функций-шаблонов и классов-шаблонов. Так, доступ к данным как элементам последовательности объектов реализуется с помощью итераторов, представляющих собой абстракцию понятия указателя.

Отметим здесь также, что ссылки в C++ были введены в основном для поддержки перегрузки стандартных операторов. Зачастую ссылки используются для указания аргументов функций и их возвращаемых значений. Необходимо лишь помнить, что ссылка, в отличие от указателя, не является объектом, над которым можно выполнять операции. Так, несмотря на форму записи операции над ссылкой, ни один оператор на самом деле не выполняет каких-либо действий над ней. Более того, компилятор может оптимизировать ссылку таким образом, что во время исполнения вообще не будет объекта, представляющего ссылку. Напротив, можно считать, что реализацией ссылки, например, может являться константный указатель, при использовании которого каждый раз происходит его разыменование.

> **Пример 52**

Применяя квалификатор ***const*** при объявлении указателей и ссылок, можно в разной степени ограничивать права доступа к тем объектам, к которым эти указатели или ссылки будут "привязаны". Так, при объявлении указателя как ***const имя типа**** константным становится не указатель, а объект, к которому будет "привязан" этот указатель. Подобным образом, при объявлении ссылки как ***const имя_типа&*** константой становится не ссылка, а объект, к которому будет "привязана" эта ссылка.

Префиксный оператор объявления ****const*** используется только при объявлении константных указателей. Указатель становится константным, если он объявляется как ***имя типа*const,*** и указатель становится константным указателем на константу, если он объявляется как ***const имя типа*const.***

Представим программу, где в функции ***main()*** будут объявлены указатель на ***int,*** указатель и ссылка на константу, а также константный указатель и константный указатель на константу:

```
// C++ Инструкции
#include <iostream>
int main()
{
 using namespace std;
 int a(3), b(5), c(7) , d(9);
 int* pointerOnlnt = &a;
 const int& referenceOnlntConstant = a;
 const int* pointerOnlntConstant = &b;
 int*const constantPointerOnlnt = &c;
 const int*const constantPointerOnlntConstant = &d;
 cout << &a << '\t' << pointerOnlnt << '\t'
      << *pointerOnInt << '\t' << referenceOnlntConstant << endl;
 cout << &b << '\t' << pointerOnlntConstant << '\t'
      << *pointerOnIntConstant << endl;
 cout << &c << '\t' << constantPointerOnlnt << '\t'
      << *constantPointerOnInt << endl;
 cout << &d << '\t' << constantPointerOnlntConstant << '\t'
      << *constantPointerOnlntConstant << endl;
pointerOnlnt = &b;
 cout << &b << '\t' << pointerOnlnt << '\t'
      << *pointerOnInt << endl;
 *pointerOnlnt = 6;
 cout << &b << '\t' << pointerOnlnt << '\t'
       << *pointerOnInt << endl;
 pointerOnlntConstant = &d;
 cout << &d << '\t' << pointerOnlntConstant << '\t'
      << *pointerOnIntConstant << endl;
return 0;
}
```

Результат работы программы:

```
0x76fdf4          0x76fdf4          3       3
0x7 6f dff0       0x7 6fdf 0        5
0x76fdec          0x76fdec          7
0x7 6fde8         0x76fde8          9
0x76fdf0          0x7 6f df 0       5
0x76fdf0          0x76fdf0          6
0x76fde8          0x7 6fde8         9
```

Итак, благодаря использованию указателей и ссылок на константы в качестве посредников для доступа к объектам, к которым эти указатели или ссылки могут быть "привязаны", становится возможным ограничить доступ к этим объектам теперь только для чтения. Подобного рода защита зачастую применяется для аргументов функций, так, например, объявив аргумент указателем или ссылкой на константу, функция уже не сможет изменить объект, на который этот аргумент ссылается. Далее будет сказано, как можно преодолеть эту защиту, если воспользоваться механизмом "снятия константности", основанного на статическом приведении типов.

Следует отметить, что ссылки на переменные и ссылки на константы различаются. Инициализация ссылки тривиальна, когда инициализатором является ***lvalue*** (объект, адрес которого можно получить). Напомним здесь, что в C++ именующее выражение ***lvalue*** (от слов ***left value***) — это выражение, ссылающееся на объект как на "нечто, что есть непрерывная область памяти". Исходным смыслом понятия ***lvalue*** было "нечто, что может быть использовано в левой части оператора присваивания", однако в C++ не каждое ***lvalue*** может использоваться в левой части оператора присваивания.

Так, инициализатором для ***имя типа&*** должно быть ***lvalue*** типа ***имятипа.*** Инициализатор для ***const имя_типа&*** не обязан быть ***lvalue*** и даже иметь тип ***имя типа,*** в таких случаях осуществляется, если это необходимо, неявное преобразование к типу ***имя типа,*** результирующее значение помещается в созданный временный объект типа ***имя типа,*** который затем используется уже как инициализатор объявляемой ссылки на константу. Временный объект, созданный для хранения инициализатора, при этом будет существовать до конца области видимости инициализируемой им ссылки. Например, объявление ссылки на константу **`const doubles r = 1;`** приведет к объявлению временной переменной с соответствующей инициализацией **`double temporary = double(1);`** и использованию этой переменной в качестве инициализатора объявляемой ссылки **`const doubles r = temporary;`**

Константный указатель не может теперь "расстаться" с тем объектом, к которому он был "привязан". Кроме того, что константный указатель на константу не только не может "расстаться" с тем объектом, к которому он был "привязан", с его помощью также нельзя будет изменить и сам объект. Как видим, неконстантные указатели, т.е. обычный указатель и указатель на константу, могут "расстаться" с тем объектом, к которому они были "привязаны".

> **Пример 53**

"Снятие константности" при помощи операторов статического приведения типов с объектов, к которым могут быть "привязаны" ссылки и указатели на константу, а также константные указатели на константу, является одним из возможных средств по преодолению защитного барьера "только для чтения". "Снятие константности" может быть выполнено либо в стиле С с помощью оператора приведения типа (***имя_типа)выражение***, либо в стиле С++ с помощью оператора конструирования значения ***имя_типа(выражение)*** или оператора константного преобразования ***const_cast<имя_типа>(выражение).***

Вначале представим программу, где будет проиллюстрирован механизм "снятия константности" с использованием только оператора константного преобразования:

```
// C++ Инструкции
#include <iostream>
int main ()
{
using namespace std;
int a(4), b(7);
const int& referenceOnlntConstant =1;
int& referenceOnlnt = const_cast<intS>(referenceOnlntConstant);
const int* pointerOnlntConstant = &a;
int* pointerOnlnt = const_cast<int*>(pointerOnlntConstant);
const int*const constantPointerOnlntConstant = &b;
cout << referenceOnlntConstant << '\t' << referenceOnlnt << endl;
referenceOnlnt = 2;
cout << referenceOnlntConstant << '\t' << referenceOnlnt << endl;
cons t_cas t<int&>(referenceOnlntConstant) = 3;
cout << referenceOnlntConstant << '\t' << referenceOnlnt << endl;
cout << a << '\t' << *pointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
*pointerOnInt = 5;
cout << a << '\t' << *pointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
*(const_cast<int*>(pointerOnlntConstant)) = 6;
cout << a << '\t' << *pointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
pointerOnlnt = const_cast<int*>(constantPointerOnlntConstant);
cout << b << '\t' << *constantPointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
*pointerOnlnt = 8;
cout << b << '\t' << *constantPointerOnIntConstant << '\t'
       << *pointerOn!nt << endl;
```

```
*(const_cast<int*>(constantPointerOnIntConstant)) = 9;
cout << b << '\t' << *constantPointerOnIntConstant << '\t'
      << *pointerOnInt << endl;
return 0;
}
```

Результат работы программы:

```
1       1
2       2
3       3
4       4       4
5       5       5
6       6       6
7       7       7
8       8       8
9       9       9
```

“Снятие константности” при помощи операторов статического приведения типов с объектов, к которым могут быть “привязаны” ссылки и указатели на константу, а также константные указатели на константу, может быть выполнено одним из двух способов. Так, для ссылок на константу первый способ “снятия константности”, как видим, предполагает объявление ссылки на ***имя_типа*** в качестве посредника, инициализатор которой приводится к типу ***имя_типа&*** одним из операторов статического приведения типов. Что касается указателей на константу и константных указателей на константу, го здесь, как видим, имеет место не только объявление указателя на ***имя типа*** в качестве посредника, инициализатор которого приводится к типу ***имя типа**** одним из операторов статического приведения типов, но также и традиционная для указателей операция — переопределение значения. Отметим здесь, что при объявлении ссылки в качестве посредника для преобразования типа ее инициализатора можно воспользоваться не только оператором константного преобразования или оператором приведения типа, но и оператором конструирования значения, если вместо типа ***имя_типа&*** приводить к типу ***имя типа,*** а вот при объявлении указателя — только первыми двумя, исключив оператор конструирования значения. “Снятие константности” вторым способом, как видим, здесь выполняется “на лету”, т.е. без объявления ссылки или указателя на ***имя типа*** в качестве посредника. Очевидно, что при этом можно воспользоваться только первыми двумя из перечисленных операторов статического приведения типов, исключив оператор конструирования значения.

Как видим, при объявлении ссылки на константу временный объект, созданный компилятором для хранения инициализатора, доступен для последующих изменений благодаря своему существованию до конца области видимости инициализируемой им ссылки.

> **Пример 54**

C++ унаследовал от C оператор приведения типа, который может осуществлять преобразование между родственными типами как ***static cast*** и несвязанными типами как ***reinterpret_cast,*** а также “снимать константность” как ***const cast.***

Теперь представим программу, где будет проиллюстрирован механизм “снятия константности” с использованием только оператора приведения типа:

```
// C++ Инструкции
#include <iostream>
int main ()
{
using namespace std;
int a(1), b(4), c(7);
 const int& referenceOnlntConstant = a;
 int& referenceOnlnt = (int&)referenceOnlntConstant;
 const int* pointerOnlntConstant = &b;
 int* pointerOnlnt = (int*)pointerOnIntConstant;
 const int*const constantPointerOnlntConstant = &c;
 cout << referenceOnlntConstant << '\t' << referenceOnlnt << endl;
referenceOnlnt = 2;
cout << referenceOnlntConstant << '\t' << referenceOnlnt << endl;
(int&)referenceOnlntConstant = 3;
cout << referenceOnlntConstant << '\t'  << referenceOnlnt << endl;
cout << b << '\t' << *pointerOnIntConstant << '\t' << *pointerOnInt <<
endl;
 *pointerOnlnt = 5;
 cout << b << '\t' << *pointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
 *((int*)pointerOnlntConstant) = 6;
 cout << b << '\t' << *pointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
pointerOnlnt = (int*)constantPointerOnlntConstant;
cout << c << '\t' << *constantPointerOnIntConstant << '\t' <<
*pointerOnInt << endl;
 *pointerOnInt = 8;
cout << c << '\t' << *constantPointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
 *((int*)constantPointerOnlntConstant) = 9;
cout << c << '\t' << *constantPointerOnIntConstant << '\t'
       << *pointerOnInt << endl;
return 0;
}
```

Результат работы программы:

```
1          1
2          2
3          3
4          4          4
5          5          5
6          6          6
7          7          7
8          8          8
9          9          9
```

Ссылаясь на Страуструпа, необходимо заметить, что приведение типов в стиле C намного опаснее, чем приведение типов при помощи именованных операторов преобразования в силу многих причин. В первую очередь, это связано не столько со сложностью отслеживания в программе самих преобразований как потенциального источника ошибок и побочных эффектов, сколько с неочевидностью самого вида преобразования. Напротив, использование именованных операторов приведения типа позволяет пользователю указать те места в программе, где содержатся потенциально опасные трансформации типов.

В завершение приведем здесь возможные формы записи инициализатора в случае использования оператора приведения типа и оператора конструирования значения при объявлении ссылки на константу. Так, например, с использованием оператора приведения типа возможна и такая запись:

```
int&referenceOnlnt = (int)referenceOnlntConstant;
```

а с использованием оператора конструирования значения — еще и такая запись:

```
int& referenceOnlnt = int(referenceOnlntConstant);
```

Если “снятие константности” выполнять “на лету”, то попытка использовать эти формы записи будет успешной только для компиляторов платформы Windows, для компиляторов платформы Linux эти же формы записи приведут к ошибкам времени компиляции. Так, например, с использованием оператора приведения типа для платформы Windows возможна такая запись:

```
(int)referenceOnlntConstant = 3;
```

а с использованием оператора конструирования значения — еще и такая запись:

```
int(referenceOnlntConstant) = 3;
```

> **Пример 55**

В объявляющей части объявления одни и те же операторы объявления могут следовать друг за другом, например, при объявлении указателей на указатели — это префиксные операторы *, при объявлении многомерных массивов — это суффиксные операторы []. Так, ***имя типа**** — это встроенный тип, называемый указателем на указатель на ***имя типа.*** Переменная типа ***имя типа**** содержит адрес объекта типа ***имя типа**, который, в свою очередь, содержит адрес объекта типа ***имя типа.***

Представим программу, где в функции ***main()*** будут объявлены указатель на ***int*** и указатель на указатель на ***int.***

```
// C++ Инструкции
#include <iostream>
int main ()
{
using namespace std;
int a (5);
int* pointerOnlnt = &a;
int** pointerOnPointerOnlnt = SpointerOnlnt;
cout << &a << '\t' << pointerOnlnt << '\t'
      << *pointerOnInt << endl;
cout << pointerOnPointerOnlnt << '\t' << &a << '\t' <<
*pointerOnPointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
 *pointerOnlnt = 7;
 cout << a << '\t' << *pointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
 **pointerOnPointerOnlnt = 5;
 cout << a << '\t' << *pointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
return 0;
}
```

Результат работы программы:

```
0x7 6fdf4           0x7 6fdf4                   5
0x7 6fdf0           0x7 6fdf4            0x76fdf4          5
7          7          7
5          5          5
```

Как видим, допустимое количество операторов * для разыменования указателя определяется формой его объявления. Напомним здесь, что все унарные операторы правоассоциативны, поэтому выражение ***pointerOnPointerOnlnt** с точки зрения синтаксиса означает *(***pointerOnPointerOnlnt).** Далее будет сказано, что в адресной арифметике оно еще означает и *(*(***pointerOnPointerOnlnt** + **0)** + **0)**

> **Пример 56**

Иногда вместо указателя требуется использовать ссылку на указатель, в этом случае при объявлении ссылки в качестве ее базового типа следует воспользоваться типом этого указателя. Например, ***имя типа*&*** — это встроенный тип, называемый ссылкой на указатель на ***имя типа***, а ***имя типа**&*** — это встроенный тип, называемый ссылкой на указатель на указатель на ***имя типа.***

Представим программу, где в функции ***main()*** будут объявлены указатель на ***int*** и указатель на указатель на ***int,*** а также ссылки на эти указатели:

```
// C++ Инструкции
#include <iostream>
int main()
{
using namespace std;
int a(5);
int* pointerOnlnt = &a;
int*& referenceOnPointerOnlnt = pointerOnlnt;
int** pointerOnPointerOnlnt = SpointerOnlnt;
int**& referenceOnPointerOnPointerOnlnt = pointerOnPointerOnlnt;
cout << &a << '\t' << pointerOnlnt << '\t'
      << referenceOnPointerOnlnt << '\t'  << *pointerOnInt << '\t'
      << *referenceOnPointerOnlnt << endl;
cout << pointerOnPointerOnlnt << '\t'
      << referenceOnPointerOnPointerOnlnt << endl;
cout << &a << '\t' << *pointerOnPointerOnInt << '\t'
      << *referenceOnPointerOnPointerOnInt << '\t'
      << **pointerOnPointerOnInt << '\t'
      << **referenceOnPointerOnPointerOnInt << endl;
return 0;
}
```

Результат работы программы:

```
0x7 6fdf4            0x7 6fdf4           0x76fdf4          5        5
0x7 6fdf0            0x7 6fdf0
0x7 6fdf4             0x76fdf4           0x76fdf4          5        5
```

> **Пример 57**

Известно, что время жизни именованного объекта программы определяется его областью видимости. Зачастую возникает необходимость в создании таких объектов, которые существуют вне зависимости от области видимости, в которой они были созданы. Такие объекты создаются в так называемой свободной или динамической памяти при помощи оператора ***new*** и существуют в ней до тех пор, пока не будут удалены при помощи оператора ***delete.*** Результатом операции успешного выделения свободной памяти для размещения объекта является адрес занятого участка памяти под этот объект, в противном случае — 0. Таким образом, единственным посредником для доступа к неименованному объекту в свободной памяти является указатель.

Представим программу, где в функции ***main()*** будут объявлены динамические объекты — переменная типа ***int*** и указатель на ***int:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
 using namespace std;
 int* pointerOnlnt = new int;
 *pointerOnInt = 5;
 int** pointerOnPointerOnlnt = new int*;
 *pointerOnPointerOnInt = pointerOnlnt;
 cout << pointerOnlnt << '\t' << *pointerOnInt << endl;
 cout << pointerOnPointerOnlnt << '\t'
      << *pointerOnPointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
**pointerOnPointerOnInt = 7;
  cout << *pointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
delete pointerOnlnt;
  delete pointerOnPointerOnlnt;
return 0;
}
```

Результат работы программы:

```
0x8905a0                    5
0x8905b0              0x8905a0         5
7          7
```

Отметим, что в стандартной библиотеке операторы выделения и освобождения памяти ***new*** и ***delete*** реализованы в виде операторных функций: ***void* operator new(size_t); void operator delete(void*);***

Здесь ***void*** — это встроенный тип C++, используемый либо для указания на то, что функция не возвращает значения, либо в качестве базового типа для указателей на объекты неизвестного типа, а ***sizet*** — это интегральный тип без знака, он определен в стандартной библиотеке как тип результата оператора ***sizeof.*** Напомним здесь, что указатель на ***void*** называют еще родовым или обобщенным указателем. Основными применениями типа ***void**** являются передача указателей функциям, которым не дано делать предположения о типе объектов, а также возврат объектов неизвестного типа из функций. Чтобы воспользоваться таким объектом, необходимо выполнить явное преобразование типа указателя.

Стандартная реализация функции ***operator new()*** не инициализирует по умолчанию выделяемую память. Отметим также, что стандартная реализация оператора ***new*** выделяет памяти немного больше, чем потребовалось бы для статического объекта. Это связано с необходимостью хранения размера динамического объекта, так как оператор ***delete*** должен иметь возможность определить размер уничтожаемого им объекта, чтобы вернуть обратно в свободную память захваченный у нее ресурс. Как правило, для хранения размера объекта используется одно дополнительное слово. Следует отметить, что в случае успешного выделения свободной памяти при помощи оператора ***new*** будет неявно вызван конструктор для инициализации создаваемых объектов класса, что является важным преимуществом использования именно этой операции по сравнению с другими для создания объектов в свободной памяти.

Свободная память является одним из тех ресурсов компьютера, который наиболее часто захватывается конструктором класса; другим таким ресурсом, например, является файл. Очевидно, что “динамическое” выделение памяти для объектов класса требует наличия в этом классе некоторой функции, которая будет гарантированно вызвана для освобождения выделенной памяти при уничтожении объекта класса, аналогично конструктору, который гарантированно вызывается при создании объекта. Возможность автоматического освобождения памяти, как и любого другого ресурса, который захватывается конструктором класса, обеспечивает специальная функция-член класса — деструктор. В теле деструктора пользователь класса обязан определить те действия, которые бы привели к освобождению захваченного ресурса. Отметим, что даже в случае попытки возврата в свободную память захваченного у нее ресурса “вручную”, т.е. при помощи оператора ***delete,*** все равно будет неявно вызван деструктор класса. Тем самым для динамических объектов класса, в отличие от объектов встроенных типов, деструктор должен стать единственно возможным средством для освобождения захваченного конструктором ресурса.

Остается лишь напомнить, что для встроенных типов наряду с конструктором по умолчанию разработчиками C++ также естественным образом введен и деструктор, что является последовательной реализацией цели по обеспечению одинакового поведения всех типов языка как в семантическом, так и синтаксическом отношениях.

> **Пример 58**

Создавая неименованный объект программы в свободной памяти при помощи оператора ***new,*** можно указать инициализатор, если воспользоваться механизмом конструирования значения, принятого в C++ для инициализации объектов класса при помощи его конструкторов. В этом случае вместо операнда ***имя типа*** следует воспользоваться операндом ***имя_типа(выражение).***

Представим программу, где в функции ***main()*** будут объявлены динамические объекты — переменная типа ***int,*** инициализатором которой будет константное выражение, и указатель на ***int,*** инициализатором которого будет указатель на ***int:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
 using namespace std;
 int* pointerOnInt = new int(5);
 int** pointerOnPointerOnInt = new int*(pointerOnInt);
 cout << pointerOnInt << '\t' << *pointerOnInt << endl;
 cout << pointerOnPointerOnInt << '\t'
      << *pointerOnPointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
**pointerOnPointerOnInt = 7;
 cout << *pointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
delete pointerOnInt;
delete pointerOnPointerOnInt;
return 0;
}
```

Результат работы программы:

```
0x8905a0                    5
0x8905b0            0x8905a0          5
7          7
```

> **Пример 59**

Привлекая инструкции-объявления и инструкции-выражения, продолжим обзор стандартных операторов C++, обращаясь, как и прежде, к иллюстрации форматного вывода значений арифметических выражений, операндами которых на этот раз будут локальные переменные функции ***main().***

Представим программу, где в операциях над целочисленными объектами будут использованы тернарный оператор (условное выражение), бинарные операторы (умножение и присваивание, деление и присваивание, остаток и присваивание, сложение и присваивание, вычитание и присваивание, сдвиг влево и присваивание, сдвиг вправо и присваивание, И и присваивание, ИЛИ и присваивание, исключающее ИЛИ и присваивание) и оператор

```
// C++ Инструкции
#include <iostream>
int main ()
{
using namespace std;
int a = 2;
int b = 3;
int c=a < b ? a:b;
cout << c << endl;
b *= a;
cout << b << '\t';
b /= a;
cout << b << '\t';
b %= a;
cout << b << '\t';
b += a;
cout << b << '\t';
b -= a;
cout << b << endl;
b  <<= a;
cout << b << '\t';
b >>= a;
cout << b << endl;
b &= a;
cout << b << '\t';
b^ = a;
cout << b << '\t';
b л= a;
cout << b << endl ;
++a, b++, c = a++;
cout << a << '\t' << b << '\t'  <<c << endl;
return 0;
}
```

Результат работы программы:

```
2
6       3               1       3               1
4       1
0       2               0
4       1               3
```

Напомним здесь, что все операторы присваивания правоассоциативны. Типом и значением выражения присваивания являются тип и значение его левого операнда после завершения операции присваивания. Левый операнд при этом должен быть ***lvalue.*** Например, ***b*** *= ***a*** означает ***b*** = ***(b)*** * ***(a).*** Следует отметить, что использование операторов присваивания, как правило, помогает компилятору сгенерировать более эффективный код благодаря именно тому, что значение левого операнда выражения присваивания вычисляется только один раз. Кроме того, такая форма записи выражений присваивания улучшает понимание производимых операций, поскольку она отражает природу мыслительного процесса человека, выполняющего какие-либо вычисления.

Оператор условное выражение и оператор , (запятая или последовательность) левоассоциативны. Например, ***a < b ? a : b*** означает **((a < b*) ? (a) *: (b))**. Если и второй, и третий операнды оператора условное выражение являются ***lvalue*** и при этом имеют одинаковый тип, результат имеет такой же тип и является ***lvalue.*** Следует отметить также, что из двух последних операндов оператора условное выражение всегда вычисляется значение только одного, “переключателем” здесь является значение первого операнда (***true*** или ***false).***

Оператор , гарантирует, что операнд слева будет вычислен до операнда справа. Результат операции — это значение операнда справа. Оператор , позволяет вычислять несколько выражений там, где по синтаксическим правилам возможно только одно. Например, оператор , зачастую используется в инструкции-итерации ***for,*** а именно в той ее части, где указывается список выражений, которые вычисляются всякий раз после каждой итерации.

> Пример 60

Завершая обзор стандартных операторов C++, от линейных алгоритмов перейдем к условным и циклическим, привлекая для программирования этих алгоритмов наряду с известными и остальные инструкции. Зададимся целью организовать ввод значения целочисленной переменной для указанного диапазона изменения ее значений.

Поначалу представим программу, где для организации цикла с постусловием будут использованы помеченная-инструкция, инструкция-выбора ***if*** и инструкция-перехода ***goto,*** а для операции ввода будет реализован механизм защиты, основанный на проверке состояния потока ввода ***fail()*** перед использованием предположительно считанных данных:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      repeat : cout << "Целое число [1, 100]? ";
      cin >> n;
  if (cin.fail())
      cout << "Ошибка формата!\n";
  else
      if (!(n >= 1 && n <= 100)) goto repeat;
return 0;
}
```

Результат работы программы:

```
Целое число [1, 100]? 0
Целое число [1, 100]? 101
Целое число [1, 100]? 1
```

Результат работы программы:

```
Целое число [1, 100]? !
Ошибка формата!
```

Напомним здесь, что каждый поток (***istream*** или ***ostream***) имеет связанное с ним состояние. Благодаря установке и соответствующей проверке этого состояния можно вылавливать ошибки и нестандартные условия. Так, если установилось состояние потока ввода ***fail()*** следующая операция уже не выполнится. Считается, что поток не испорчен и никакие символы не потерялись. Далее будет сказано, как поток ввода можно перевести в состояние ***good()*** и затем освободить его от “мусора”.

Ранее отмечалось, что состояние потока представляется набором флагов. Флаги состояния потока, как и флаги формата, определены в ***ios base*** — базовом классе класса ***basicios.*** Применение операции ввода-вывода к потоку приводит к установке одного из четырех флагов состояния потока — ***goodbit*** (0x00), ***badbit*** (0x01), ***eofbit*** (0x02) или ***failbit*** (0x04). Проверка флагов состояния потока осуществляется при помощи константных функций-членов класса ***basic ios — good(), bad(), eof()*** и ***fail().*** Все эти функции для получения флагов обращаются к константной функции-члену этого же класса ***rdstate().***

Так, например, функция ***good(),*** проверяя флаг ***goodbit,*** позволяет определить, что операция ввода-вывода для потока выполнилась успешно. Это означает, что следующая операция для потока тоже может выполниться. Применение операции к потоку, находящемуся не в состоянии ***good(),*** — это нулевая операция. Функция ***bad(),*** проверяя флаг ***badbit,*** позволяет определить, что поток был испорчен. Это означает, что операция для потока была недопустимой. Функция ***eof(),*** проверяя флаг ***eofbit,*** позволяет определить, что уже достигли конца файла. Функция ***fail(),*** проверяя флаг ***failbit,*** позволяет определить, что операция для потока из-за ошибок форматирования или преобразования выполнилась неуспешно. Это означает, что следующая операция ввода-вывода для потока уже не выполнится. Заметим, что в силу традиции функцией ***fail()*** наряду с флагом ***failbit*** проверяется также и флаг ***badbit,*** который в силу той же традиции зачастую становится "спутником" флага ***failbit.***

Когда поток используется как условие, состояние потока можно проверить при помощи константных операторных функций-членов класса ***basic_ios — operatorl()*** или ***operator void****0. Проверка считается успешной, если состояние потока ***fail()*** и не ***fail()*** соответственно. Например, такая форма организации цикла

```
repeat : cout << "Целое число [1, 100]? ";
 if (!(cin >> n))
    cout << "Ошибка формата!\n";
 else
if (!(n >= 1 && n <= 100)) goto repeat;
```

по сравнению с предыдущей приводит к сокращению его тела на одну инструкцию.

Отметим, что в случае отсутствия мер по защите операции ввода возможно так называемое "зацикливание" программы. Например, такая форма организации цикла

```
repeat : cout << "Целое число [1, 100]? ";
  cin >> n;
if (!(n >= 1 && n <= 100)) goto repeat;
```

небезопасна, так как может привести к "зацикливанию" программы.

Итак, проверка состояния потока ***cin*** после каждой операции ввода должна стать одним из первых шагов на пути к безопасному поведению кода программы. Далее будет сказано и о другом способе проверки состояния потока, основанном на технике обработки исключения ***ios base::failure.*** Остается только признать, что компилятор не установит состояние ***fail()*** для потока ввода, если до считывания первого "неправильного с точки зрения формата" символа будет считан хотя бы один "правильный". Например, после такой операции ввода

```
Целое число [1, 100]? 1!
```

поток будет находиться в состоянии ***good(),*** и значение переменной ***n*** будет равно 1.

> **Пример 61**

Теперь представим программу в стиле C, где для операции ввода будет реализован механизм защиты, основанный на проверке возвращаемого значения библиотечной функции форматного ввода ***scanf():***

```
// C++ Инструкции
#include <cstdio>
int main ()
{
      int n;
      repeat : printf("Целое число [1, 100]? ");
      if (!scanf("%d", &n) )
            printf("Ошибка формата!\n");
      else
            if (!(n >= 1 && n <= 100)) goto repeat;
return 0;
}
```

Результат работы программы:

Целое число [1, 100]? 0
Целое число [1, 100]? 101
Целое число [1, 100]? 1

Результат работы программы

Целое число [1, 100]? !
Ошибка формата!

Напомним, что в заголовочном файле ***<cstdio>*** представлены объявления функций ввода-вывода в стиле C, характерной особенностью которых является неуказанное количество их аргументов. Там, например, представлен и прототип используемой здесь библиотечной функции форматного ввода ***scan():***

int scanf(const char*, ...);

Отметим, что аргументы вызова функции ***printf(),*** связанные с потоком вывода, передаются по значению, в то время как аргументы вызова функции ***scanf(),*** связанные с потоком ввода, передаются по адресу. Если операция ввода-вывода выполнилась успешно, возвращается 1, в противном случае — 0.

Необходимо также отметить, что в практике программирования те программы, где для передачи управления неоправданно часто использовалась инструкция-перехода ***goto,*** что рано или поздно приводило к значительным трудностям в понимании хитроумной и весьма запутанной логики их работы, получили название BS-программ (аббревиатура от слов ***Bowl of Spaghetti*** — блюдо спагетти).

> **Пример 62**

Возвращаясь к безопасному потоковому вводу в стиле C++, теперь представим программу, где для организации цикла с предусловием будут использованы инструкция-итерации ***while,*** инструкция-выбора ***if*** и инструкция-перехода ***break:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
   using namespace std;
   int n;
   while (1)
   {
   cout << "Целое число [1, 100]? ";
   cin >> n;
   if (cin.fail())
   {
      cout << "Ошибка формата!\n";
      break;
   }
   if (n >— 1 && n <= 100) break;
}
return 0;
}
```

Результат работы программы:

```
Целое число [1,   100]? 0
Целое число [1,   100]? 101
Целое число [1,   100]? 1
```

Результат работы программы:

```
Целое число [1,   100]? !
Ошибка формата!
```

Как видим, в инструкции-итерации ***while*** “традиционное” условие выхода из цикла в виде логического выражения можно заменить и арифметическим выражением, значение которого, по крайней мере, должно быть отличным от нуля. Говорят, что в этом случае тело цикла может выполняться бесконечно много раз. Чтобы реализовать выход из такого цикла с предусловием, здесь используется инструкция-выбора ***if*** вместе с инструкцией-перехода ***break.***

> **Пример 63**

Продолжая иллюстрацию безопасного потокового ввода в стиле C++, представим теперь программу, где для организации цикла с постусловием будут использованы инструкция-итерации ***do,*** инструкция-выбора ***if*** и инструкция-перехода ***break:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      do
      {
      cout << "Целое число [1, 100]? ";
      cin >> n;
      if (cin.fail())
            {
            cout << "Ошибка формата!\n";
            break;
            }
      if (n >= 1 && n <= 100) break;
      }
while (1);
return 0;
}
```

Результат работы программы:

```
Целое число [1, 100]? 0
Целое число [1, 100]? 101
Целое число [1, 100]? 1
```

Результат работы программы:

```
Целое число [1, 100]? !
Ошибка формата!
```

Как и в предыдущем примере, вместо “традиционного” условия выхода из цикла в виде логического выражения здесь тоже используется арифметическое выражение со значением, отличным от нуля. Поэтому, чтобы реализовать выход из такого цикла с постусловием, можно также воспользоваться инструкцией-выбора ***if*** вместе с инструкцией-перехода ***break.***

> **Пример 64**

Следует признать, что стремление к достижению эффективного кода процедуры ввода в случае использования инструкции-итерации ***do*** непременно должно привести к указанию в ее части ***while*** традиционного по своей сути условия выхода из цикла в виде логического выражения.

Представим теперь программу, где для организации цикла с постусловием на этот раз будет использована "традиционная" инструкция-итерации ***do:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      do
      {
      cout << "Целое число [1, 100]? ";
      cin >> n;
      if (cin. fail ())
            {
            cout << "Ошибка формата!\n";
            break;
            }
      }
while (! (n >= 1 && n <= 100));
return 0;
}
```

Результат работы программы:

```
Целое число [1, 100]? 0
Целое число [1, 100]? 101
Целое число [1, 100]? 1
```

Результат работы программы:

```
Целое число [1, 100]? !
Ошибка формата!
```

Далее будет сказано, как для такой традиционной формы организации цикла с постусловием поток ввода ***cin*** из состояния ***fail()*** можно перевести в состояние ***good()*** и затем, освободив его от "мусора", вернуться к операции ввода, чтобы исключить возможность аварийного завершения программы.

> **Пример 65**

Инструкцию-итерации ***for,*** как правило, следует использовать для организации так называемого счетного цикла, тело которого выполняется фиксированное число раз.

Памятуя о том, что счетный цикл является частным случаем цикла с предусловием, на первых порах представим программу, где инструкция-итерации ***for*** будет “играть” уже известную роль инструкции-итерации ***while:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
 using namespace std;
 int n;
 for (;;)
 {
      cout << "Целое число [1, 100]? ";
      cin >> n;
      if (cin. fail ())
      {
            cout << "Ошибка формата!\n";
            break;
      }
      if (n >= 1 && n <= 100) break;
 }
return 0;
}
```

Результат работы программы:

```
Целое    число  [1,  100]? 0
Целое    число  [1,  100]? 101
Целое    число  [1,  100]? 1
```

Результат работы программы:

```
Целое    число  [1,  100]? !
Ошибка формата!
```

Как видим, чтобы реализовать выход из такого цикла с предусловием, тело которого, как известно, может выполняться бесконечно много раз, здесь тоже следует воспользоваться инструкцией-выбора ***if*** вместе с инструкцией-перехода ***break.***

Далее будет сказано и о других синтаксических формах инструкции-итерации ***for,*** которые позволяют строить самые разнообразные виды счетных циклов.

> Пример 66

Осуществляя контроль состояния потока ввода ***cin,*** несложно добиться также и устранения возможности аварийного завершения программы. Для этого потребуется перевести поток из состояния ***fail()*** в состояние ***good()*** и очистить его от "мусора". Так, в состояние ***good()*** поток ввода можно перевести, например, с помощью функции-члена ***clear()*** класса ***basic_ios,*** которая устанавливает флаг состояния ввода ***goodbit,*** тем самым сбрасывая флаг предыдущего состояния операции ввода ***failbit.*** Освободить поток от "мусора" можно, например, с помощью функции ***getline()*** из ***<string>,*** которая считывает из потока строку с "мусором" в указанную строку, по мере надобности расширяя последнюю. Так как символ-ограничитель (указанный символ конца строки или символ новой строки) из потока удаляется, это позволяет повторно вернуться к операции ввода и полагаться на ее благополучный исход.

Представим программу, где будет проиллюстрирован один из таких вариантов "безаварийного" механизма защиты операции ввода, если для организации цикла с постусловием будет использована инструкция-итерации ***do:***

```
// C++ Инструкции
#include <iostream>
#include <string>
int main()
{
      using namespace std;
      int n;
      string badSymbols;
      do
      {
      cout << "Целое число [1, 100]? ";
      cin >> n;
      if (cin.fail())
            {
            cout << "Ошибка формата!\n";
            cin.clear();
            getline(cin, badSymbols);
            }
      }
while (!(n >= 1 && n <= 100));
return 0;
}
```

Результат работы программы:

```
Целое число [1,   100]? 0
Целое число [If   100]? 101
Целое число [1,   100]? 1
```

Результат работы программы:

```
Целое число [1, 100]? !
Ошибка формата!
Целое число [1, 100]? 1
```

Отметим, что основным назначением функции ***clear()*** является сброс состояния потока ввода-вывода в ***good(),*** так как флаг ***ios base::.goodbit*** — это стандартное значение ее аргумента. Однако поскольку это единственная функция, которая непосредственно изменяет состояние потока ввода-вывода, именно ей и поручается сгенерировать исключение ***iosjbase::failure,*** чтобы как раз обратить внимание на это изменение состояния. Далее будет сказано, как использование техники обработки исключений может помочь контролировать поток ввода.

Здесь функция ***getline()*** считывает из потока строку, оканчивающуюся символом новой строки, который сам при этом в строку ***badSymbols*** не попадает, а из потока, как уже было сказано, он удаляется.

Отметим также, что стандартные возможности работы со строками базируются на классе-шаблоне ***basicstring,*** который предоставляет типы членов и операций, схожих с теми, что обеспечиваются стандартными контейнерами библиотеки C++. Класс-шаблон ***basic string*** и все связанные с ним средства определены в пространстве имен ***std*** и представлены заголовочным файлом ***<string>.***

Следует также отметить, что два определения типов с помощью директивы ***typedef*** обеспечивают для обычных строковых типов общепринятые имена, например, для строки из символов ***char*** — это ***typedef basic_string<char> string,*** а для строки из символов ***wchar_ t*** — это ***typedef basic_string<wchar_t> wstring.***

Известно, что класс ***string*** не так идеален, как остальные средства стандартной библиотеки, здесь многое определяется свойством реализации, однако, по мнению Страуструпа, он хорошо служит многим потребностям, широко известен и доступен, а его функции обеспечивают большинство известных стандартных операций для манипулирования строками. Считается, что для широко используемого кода следует пользоваться именно стандартным библиотечным классом ***string.***

Так, например, всего лишь одно используемое здесь свойство объекта ***badSymbols*** класса ***string,*** связанное с “расширяемостью” его размера “на лету” в зависимости от количества считанных символов из потока ***cin,*** уже само по себе чрезвычайно важно для рассматриваемого механизма защиты операции ввода.

> **Пример 67**

Для обработки ошибочных или каких-либо других исключительных ситуаций времени выполнения в C++ предусмотрен механизм обработки исключений (от слова ***exceptions).*** Ключевая идея состоит в том, что функция, обнаружившая ошибку времени выполнения, генерирует ***(throw)*** исключение в надежде, что вызвавшая ее функция сможет перехватить ***(catch)*** исключение и обработать ошибку. Далее будет сказано и о самом механизме генерации исключения.

Исключение — это объект некоторого класса (чаще стандартной библиотеки, чем самого пользователя), который является представлением исключительного случая. Стандартные исключения являются частью иерархии классов, вершина которой — это класс исключений ***exception,*** представленный в заголовочном файле ***<stdexcept>.*** Говоря о контроле состояния потока ввода-вывода, стандартным исключением здесь будет объект класса ***failure*** — вложенного класса в класс ***ios base.*** Перегруженная функция-член ***exceptions***0 класса ***basic ios*** поручает функции ***clear()*** сгенерировать исключение ***iosjbasev.failure*** для указанных флагов состояния потока ввода-вывода.

Фрагмент кода, где ожидается исключительная ситуация, локализуют при помощи ***try-***блока, за которым должен следовать, по крайней мере, хотя бы один обработчик ***catch.*** Каким бы ни был характер обработки исключительных ситуаций, функция, перехватившая исключение, не возобновляет, а завершает свое выполнение.

Представим программу, где будет проиллюстрирован способ проверки состояния потока ввода, основанный на технике обработки исключения ***iosjbase::failure***:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      cin.exceptions(ios_base::failbit);
      try
   {
      do
      {
      cout << "Целое число [1, 100]? cin >> n;
      }
      while (!(n >= 1 && n <= 100));
   }
catch (ios_base::failure)
{
   cout << "Ошибка формата!\n";
}
return 0;
}
```

Результат работы программы:

```
Целое число [1,   100]? 0
Целое число [1,   100]? 101
Целое число [1,   100]? 1
```

Результат работы программы:

```
Целое число [1,   100]? !
Ошибка формата!
```

Следует отметить, что семантика перехвата и задания имен исключений идентична семантике функции с аргументом, т.е. формальный аргумент обработчика ***catch*** инициализируется значением аргумента вызова. Здесь важен тип перехватываемого объекта, а в тех случаях, когда необходимо узнать еще и состояние этого объекта, можно задать также имя аргумента вызова. Аргумент вызова можно передавать либо по значению, либо по адресу, используя указатель или ссылку. Кроме того, к типу перехватываемого объекта можно добавить квалификатор ***const*** точно так же, как его добавляют к аргументу функции, если необходимо воспрепятствовать модификации перехваченного исключения.

Когда код, вызываемый из ***try***-блока, генерирует с помощью ***throw*** исключение, то сначала прекращается выполнение программы, а затем осуществляется “раскрутка” стека до тех пор, пока не будет найден подходящий обработчик ***catch*** среди всех указанных за этим //j-блоком обработчиков. Управление передается именно тому обработчику ***catch,*** тип аргумента которого совпадает с типом сгенерированного исключения. Поэтому и говорят, что сгенерированное исключение “срезается” до перехваченного. По завершении обработки исключения управление передается за конец списка обработчиков. Если исключение сгенерировано и не перехвачено ни одним из обработчиков, произойдет аварийное завершение программы. Если исключение не сгенерировано, обработчики игнорируются, т.е. ***try-блок*** при этом ведет себя как обыкновенный блок. Напоследок остается лишь отметить, что каждый обработчик является отдельной областью видимости.

Как видим, такого рода код оказывается не столь элегантным, поэтому, следуя совету Страуструпа, отметим, что, когда для обработки ошибки потока достаточно локальных управляющих структур, исключениями лучше не пользоваться.

> **Пример 68**

Говоря об управляющих структурах выбора, используемых при программировании условных алгоритмов, от инструкции-выбора ***if*** перейдем к инструкции-выбора ***switch***, зачастую называемой переключателем. “Традиционные” переключатели в общем случае позволяют выполнить только одну из множества альтернативных ветвей условного алгоритма. Как в C, так и в C++ инструкция-выбора ***switch*** позволяет выполнить не только одну из множества альтернативных ветвей условного алгоритма, но и несколько альтернативных ветвей подряд друг за другом, начиная с той, на которую как раз и передается управление. Такого рода “свобода” в поведении переключателя ***switch*** обусловлена наличием или отсутствием среди инструкций его case-ветвей инструкции-перехода ***break.*** Очевидно, что от пользователя потребуется теперь несколько иной подход к реализации механизма множественного выбора, где некоторой группе альтернативных действий условного алгоритма следует приписать “коллективное” поведение. Очевидно также, что и case-ветви по умолчанию в такой ситуации не избежать участи в приписывании несвойственной ей роли, несмотря на ее особое положение среди всех ветвей переключателя.

Поначалу представим программу, где будет проиллюстрирован “нетрадиционный” механизм множественного выбора, для реализации которого необходимо отсутствие в case-ветвях переключателя инструкции-перехода ***break,*** чтобы выполнять операцию скользящего суммирования целых чисел для заданного ее верхнего предела:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int m, n(0);
      cout << "Верхний предел суммирования [1, 3]? ";
      cin >> m;
      if (cin.fail())
   cout << "Ошибка формата!\n";
else
   switch (m)
   {
case 3          : n +=   3;
case 2          : n +=   2;
case 1          : n +=   1;
default         : if (n)
  cout << "Сумма = " << n << endl ;
      else
  cout << "Ошибка!" << endl;
   }
return 0;
}
```

Результат работы программы:

```
Целое число [1, 3]? 0 Ошибка!
```

Результат работы программы:

```
Целое число [1, 3]? 1 Сумма = 1
```

Результат работы программы:

```
Целое число [1, 3]? 2 Сумма = 3
```

Результат работы программы:

```
Целое число [1, 3]? 3 Сумма = б
```

Результат работы программы:

```
Целое число [1, 3]? !
Ошибка формата!
```

Как видим, case-ветви переключателя — это помеченные-инструкции, где меткой является ключевое слово ***case*** вместе с константным выражением. Case-метки, как и обычные, могут следовать друг за другом. Исключением является case-ветвь по умолчанию, она может быть помечена лишь одной меткой, составленной только из ключевого слова ***default.*** Механизм передачи управления какой-либо case-ветви основан на последовательном сравнении значения переключателя ***m*** со значениями константных выражений каждой ветви, начиная с первой, до первого совпадения. Если значение переключателя ***m*** не совпадает ни с одним из значений константных выражений case-ветвей переключателя, управление передается его case-ветви по умолчанию. Заметим, что case-метки ветвей переключателя, как и обычные метки помеченных-инструкций, должны отличаться друг от друга, в противном случае это приведет к ошибке времени компиляции.

> **Пример 69**

Продолжая иллюстрацию безопасного потокового ввода в стиле C++, вернемся к уже известной проблеме ввода значения целочисленной переменной для указанного диапазона изменения ее значений и представим программу, где для решения этой проблемы на этот раз будет задействован “традиционный” механизм множественного выбора, для реализации которого в case-ветвях переключателя ***switch*** потребуется наличие инструкции-перехода ***break***:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      cout << "Целое число [1, 3]? ";
      cin >> n;
      if (cin.fail())
      cout << "Ошибка формата!\n";
            else
      switch (n)
   {
case 1 : case 2 : cout << n << endl;
break;
case 3                  : cout << n << endl;
break;
default                 : cout << n << " Ошибка!" << endl;
   }
return 0;
}
```

Результат работы программы:

```
Целое число [1, 3]? 0
0  Ошибка!
```

Результат работы программы:

```
Целое число [1, 3]? 1
1
```

Результат работы программы:

```
Целое число [1, 3]? 2
2
```

Результат работы программы:

```
Целое число [1, 3]? 3
```

3

Результат работы программы:

```
Целое число [1, 3]? 4
3  Ошибка!
4
```

Результат работы программы:

```
Целое число [1, 3]? !
Ошибка формата!
```

Как видим, в case-ветвях переключателя ***switch*** можно обойтись и без блока, так как метки “играют” здесь роль фигурных скобок в составной-инструкции. Однако это не совсем “настоящий” блок, так как он, например, не допускает многократного объявления какого-либо объекта программы в case-ветвях переключателя, которые все вместе располагаются в одной области видимости, а именно в объемлющем блоке самой инструкции-выбора ***switch.*** Можно обойтись и без case-ветви по умолчанию, в этом случае просто не выполнится сама инструкция-выбора ***switch,*** если, например, значение переключателя ***n*** здесь не совпадет ни с одним из значений константных выражений case-ветвей переключателя.

Отметим, что кроме инструкции-перехода ***break*** case-ветви переключателя могут завершаться еще и инструкцией-перехода ***return,*** если, например, не придерживаться рамок структурной парадигмы программирования при проектировании функций, допуская тем самым не одну, как это было принято в C, а сразу несколько точек выхода из функции, как это принято в C++. По-разному можно относиться к такого рода “нарушениям” стиля; зачастую определяющим фактором здесь является не столько стремление следовать определенным правилам, сколько предоставление кому-либо возможности сопровождать без особых усилий код программы.

Отмечается, что помимо улучшения восприятия и понимания кода программы использование инструкции-выбора ***switch*** может также привести и к улучшенному сгенерированному машинному коду.

> Пример 70

Продолжая обзор инструкций-итерации C++, зададимся целью организовать вычисление скользящей целочисленной суммы, прибегая к различным формам циклов, если в качестве одного из слагаемых всякий раз будет использоваться только целочисленная переменная цикла. Здесь важно проиллюстрировать не только сам механизм вычисления с использованием переменной цикла, но еще и стремление к эффективной реализации операции суммирования.

Поначалу представим программу, где для организации цикла с предусловием будет использована инструкция-итерации ***while:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
  using namespace std;
  int sum = 0;
  int i = 0;
  while (i < 5) sum += i++;
  cout << sum << endl;
  return 0;
}
```

Результат работы программы:

```
10
```

Как видим, такая форма организации цикла помимо очевидной начальной инициализации переменной ***sum*** требует также и предварительной инициализации переменной цикла ***i***. Остается признать, что цикл с предусловием — это не самый лучший механизм для вычисления скользящей суммы с использованием переменной цикла и не только из-за “разрыва” в определении верхней и нижней границ диапазона суммирования. Чем меньше “расстояние” между точкой инициализации переменной цикла и самим циклом, тем лучше и для тех, кто пишет код, и для тех, кто его только сопровождает. Очевидно, что без надлежащей инициализации переменной цикла инструкция-итерации ***while*** может и не выполниться, поэтому единственным здесь утешением остается то, что инициализация необходима не просто сама по себе, а именно для задания нижней границы диапазона суммирования.

> **Пример 71**

Представим теперь программу, где для организации цикла с постусловием будет использована инструкция-итерации ***do:***

```
// C++ Инструкции
#include <iostream>
int main ()
{
  using namespace std;
  int sum = 0;
  int i = 0;
  do
      sum += i++;
  while (i < 5);
  cout << sum << endl ;
return 0;
}
```

Результат работы программы:

10

Как видим, и такая форма организации цикла не приводит к эффективной реализации операции суммирования: здесь также необходима предварительная инициализация переменной цикла ***i***, но уже только для задания нижней границы диапазона суммирования, и так же не избежать “разрыва” в определении верхней и нижней границ диапазона суммирования.

Считается, что цикл с постусловием в практике программирования используется неоправданно часто. Огульный “запрет” на использование цикла с постусловием, однако, не стоит принимать во внимание: по крайней мере, здесь можно утверждать, что для вычисления скользящей суммы цикл с постусловием более уместен, чем его антипод — цикл с предусловием. Обрамление тела цикла, состоящего всего из одной единственной инструкции, ключевыми словами ***do*** и ***while*** напоминает блок, что само по себе представляется более важным по сравнению с той мнимой опасностью, которую порой связывают с обязательным выполнением первой итерации.

> **Пример 72**

От циклов с предусловием и с постусловием перейдем теперь к счетному циклу. Если ранее счетный цикл был реализован как “цикл с предусловием” благодаря “способности” инструкции-итерации ***for*** “сыграть” роль инструкции-итерации ***while,*** “лишив” себя своих традиционных трех частей из заголовка, то теперь будем шаг за шагом возвращать эти части на свое место в заголовке, строя всякий раз очередную разновидность счетного цикла.

Поначалу представим программу, где для организации счетного цикла будет использована инструкция-итерации ***for,*** в заголовке которой будут задействованы сразу все три ее традиционные части:

```
// C++ Инструкции
#include <iostream>
int main ()
{
  using namespace std;
  int sum = 0;
  for (int i = 0; i < 5; ++i) sum += i;
  cout << sum << endl;
  return 0;
}
```

Результат работы программы:

10

Как видим, такая форма организации цикла позволяет здесь устранить не только “разрыв” в определении верхней и нижней границ диапазона суммирования, но и обособить операцию приращения переменной цикла ***i*** после выполнения каждой итерации благодаря наличию в заголовке инструкции-итерации ***for*** именно тех его частей, которые всегда связывают с традиционным управлением счетного цикла.

Напомним, что в первой части заголовка инструкции-итерации ***for*** можно записать только одну инструкцию: либо инструкцию-объявление для объявления одной или нескольких локальных переменных, область видимости которых простирается до конца инструкции-итерации ***for,*** либо инструкцию-выражение для инициализации глобальных по отношению к самой инструкции-итерации ***for*** переменных. Вторая часть заголовка связана с так называемым условием в виде логического или арифметического выражения, которое вычисляется перед каждой итерацией, а третья часть заголовка — с инструкцией-выражением, выполняемой после каждой итерации.

Если на первом шаге тело цикла остается на своем месте как обязательный атрибут, то в дальнейшем его легко “отправить” в третью часть заголовка, но об этом уже в следующем примере.

> **Пример 73**

Осознавая важность всех трех частей заголовка инструкции-итерации ***for,*** нельзя не отметить особую роль третьей его части, связанной с вычислением скалярных выражений после каждой итерации. Если первая часть заголовка позволяет объявлять локальную переменную именно в тот момент, когда ей надо присвоить значение, чтобы исключить всякие попытки использования этой переменной до момента ее инициализации, и это воспринимается уже как норма, то в третьей части заголовка можно себе позволить и отступление от нормы, например, “упрятывание” тела цикла.

Итак, продолжая иллюстрацию счетных циклов, теперь представим программу, где в третьей части заголовка инструкции-итерации ***for*** будет “упрятано” тело цикла:

```
// C++ Инструкции
#include <iostream>
int main ()
{
  using namespace std;
  int sum = 0;
  for (int i = 0; i < 5; sum += i, ++i);
  cout << sum << endl;
  return 0;
}
```

Результат работы программы:

```
10
```

Как видим, процедура “упрятывания” тела цикла достаточно проста, необходимо лишь помнить о том, что оператор, позволяет вычислять несколько выражений там, где по синтаксическим правилам возможно только одно. Таким образом, указывая список выражений в третьей части заголовка инструкции-итерации ***for,*** которые вычисляются после каждой итерации, можно либо полностью освободиться от тела счетного цикла, либо “облегчить” его на некоторую разумную долю инструкций.

Памятуя о стремлении к эффективной реализации операции целочисленного суммирования, осталось сделать последний шаг, отказавшись от списка выражений в третьей части заголовка инструкции-итерации ***for*** благодаря применению оператора постфиксного инкремента для переменной цикла. Мотивация такого поступка вполне очевидна, так как оба выражения в этом списке связаны не только самим порядком выполняемых действий, но и одной общей переменной.

> **Пример 74**

Продолжая иллюстрацию механизма для вычисления скользящей целочисленной суммы с использованием переменной счетного цикла, представим теперь программу, где эффективность реализации операции суммирования будет доведена до своего логического предела:

```
// C++ Инструкции
#include <iostream>
int main()
{
  using namespace std;
  int sum = 0;
  for (int i = 0; i < 5; sum += i++);
  cout << sum << endl;
  return 0;
}
```

Результат работы программы:

```
10
```

Как видим, в такой форме организации цикла нет ничего лишнего, что помешало бы увидеть здесь главное: естественное поведение кода, выразительность, строгость и ясность стиля. Считается, что счетным циклом в C++ можно заменить все другие его циклы: конструкция заголовка инструкции-итерации ***for*** настолько универсальна, что может удовлетворить любые изыски как искушенных, так и не очень в непростом ремесле кодирования программ.

Редкий цикл, в том числе и счетный, обходится без инструкции-перехода, будь-то ***break*** или ***continue.*** Если инструкция-перехода ***break*** имеет отношение к выходу из цикла во время выполнения текущей итерации, то инструкция-перехода ***continue*** предназначена для пропуска текущей итерации и перехода к следующей. Очевидно, что в теле цикла всякий раз должно “созревать” некоторое условие, чтобы была причина регулярно пропускать итерации, но об этом уже в следующем примерю.

Из цикла можно выйти также и с помощью инструкции-перехода ***return***, что повлечет за собой и выход из функции. В практике программирования крайне редко встречаются и такие экзотические случаи, когда из цикла выходят даже с помощью инструкции-перехода ***goto.*** Если необходимо сгенерировать исключение, выйти из цикла можно и с помощью ***throw.***

> **Пример 75**

Завершая иллюстрацию механизма для вычисления скользящей целочисленной суммы с использованием переменной счетного цикла, зададимся целью реализовать операцию суммирования с прореживанием, где в поток вывода будут отправляться еще и все ее слагаемые.

Поначалу представим программу, где для реализации операции суммирования с прореживанием в счетном цикле будет использована инструкция-перехода ***continue:***

```
// C++ Инструкции
#include <iostream>
#include <iomanip>
int main()
{
      using namespace std;
      int sum = 0;
      for (int i = 0; i < 10; ++i)
      if (i % 3 == 0)
            continue;
      else {
            cout << setw(2) << i;
            sum += i;
            }
  cout << '\n';
  cout << sum << endl;
  return 0;
}
```

Результат работы программы:

```
 1 2 4 5 7 8
27
```

Как видим, суммированию не подлежат только те слагаемые, которые кратны 3. Благодаря выполнению этого условия как раз и запускается механизм пропуска текущей итерации и перехода к следующей. Перед тем как выполнять операцию суммирования, слагаемые отправляются в поток вывода. Для управления состоянием потока здесь используется манипулятор с аргументом ***setw(),*** предназначенный для управления шириной поля вывода.

Зачастую необходимость в использовании инструкции-перехода ***continue*** отпадает, если удается изменить логику передачи управления в алгоритме выбора, но об этом уже в следующем примере.

> **Пример 76**

В завершение представим теперь и другую версию программы, где для реализации операции суммирования с прореживанием в счетном цикле можно обойтись и без инструкции-перехода ***continue:***

```
// C++ Инструкции
#include <iostream>
#include <iomanip>
int main()
{
      using namespace std;
      int sum = 0;
      for (int i = 0; i < 10; ++i)
      if (i % 3 != 0)
      {
      cout << setw(2) << i;
      sum += i;
      }
cout << '\n';
cout << sum << endl;
return 0;
}
```

Результат работы программы:

```
1 2 4 5 7 8 27
```

Как видим, изменение логики передачи управления для инструкции-выбора ***if*** в теле счетного цикла привело к устранению избыточности кода программы. Однако можно и сохранить ветвь ***else,*** например, так:

```
      for (int i = 0; i < 10; ++i)
      if (i % 3 != 0)
      {
      cout << setw(2) << i;
      sum += i;
      }
      else
            continue;
```

В завершение обзора всех инструкций C++, зададимся целью проиллюстрировать решение задачи, связанной с подсчетом количества повторяющихся цифр в заданном четырехзначном целом числе.

Поначалу представим первую версию программы, где для подсчета будущих цифр четырехзначного числа понадобятся десять счетчиков в виде статических объектов:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      static short int digit0Counter;
      static short int digit1Counter;
      static short int digit2Counter;
      static short int digit3Counter;
      static short int digit4Counter;
      static short int digit5Counter;
      static short int digit6Counter;
      static short int digit7Counter;
      static short int digit8Counter;
      static short int digit9Counter;
      do {
            cout << "Натуральное число [1000,9999]? ";
            cin >> n;
            if (cin.fail())
            {
            cout << "Ошибка формата!\n";
            return -1;
            }
        }
      while (!(n >= 1000 && n <= 9999));
      for (; n != 0; n /= 10)
       switch (n % 10)
            {
            case 0 : ++digit0Counter;
                  break;
            case 1 : ++digit1Counter;
                  break;
            case 2 : ++digit2Counter;
                  break;
            case 3 : ++digit3Counter;
                  break;
            case 4 : ++digit4Counter;
```

```
                break;
            case 5 : ++digit5Counter;
                break;
            case 6 : ++digit6Counter;
                break;
            case 7 : ++digit7Counter;
                break;
            case 8 : ++digit8Counter;
                break;
            case 9 : ++digit9Counter;
        }
if (digitOCounter > 1) cout << "Цифра 0 повторяется "
                            << digitOCounter << " раза\n";
if (digitlCounter > 1) cout << "Цифра 1 повторяется "
                            << digitlCounter << " раза\n";
if (digit2Counter > 1) cout << "Цифра 2 повторяется "
                            << digit2Counter << " раза\n";
if (digit3Counter > 1) cout << "Цифра 3 повторяется "
                            << digit3Counter << " раза\n";
if (digit4Counter > 1) cout << "Цифра 4 повторяется "
                            << digit4Counter << " раза\n";
if (digit5Counter > 1) cout << "Цифра 5 повторяется "
                            << digit5Counter << " раза\n";
if (digit6Counter > 1) cout << "Цифра б повторяется "
                            << digit6Counter << " раза\n";
if (digit7Counter > 1) cout << "Цифра 7 повторяется "
                            << digit7Counter << " раза\n";
if (digit8Counter > 1) cout << "Цифра 8 повторяется "
                            << digit8Counter << " раза\n";
if (digit9Counter > 1) cout << "Цифра 9 повторяется "
                            << digit9Counter << " раза\n";
return 0;
}
```

Результат работы программы:

```
Натуральное число [1000,9999]? 5775
Цифра 5 повторяется 2 раза
Цифра 7 повторяется 2 раза
```

Результат работы программы:

```
Натуральное число [1000,9999]? %775
Ошибка формата!
```

> **Пример 78**

Объявление чрезмерного количества статических переменных для подсчета цифр четырехзначного числа — это неоправданно большие накладные расходы. Одним из приемлемых решений этой проблемы может стать, например, процедура упаковки всех десяти счетчиков в одну переменную типа ***int.***

Теперь представим вторую версию программы, где для подсчета будущих цифр четырехзначного числа понадобится всего один статический объект:

```
// C++ Инструкции
#include <iostream>
int main ()
{
      using namespace std;
      int n;
      int countervalue;
      int mask(07);
      static int digitCounters;
      do
       {
            cout << "Натуральное число [1000,9999]? ";
            cin >> n;
            if (cin.fail())
            {
            cout << "Ошибка формата!\n";
            return -1;
            }
       }
      while (!(n >= 1000 && n <= 9999));
      for (; n != 0; n /= 10)
      switch (n % 10)
            {
            case 0 : digitCounters += 01;
                  break;
            case 1 : digitCounters += 010;
                  break;
            case 2 : digitCounters += 0100;
                  break;
            case 3 : digitCounters += 01000;
                  break;
            case 4 : digitCounters += 010000;
                  break;
            case 5 : digitCounters += 0100000;
                  break;
            case 6 : digitCounters += 01000000;
                  break;
            case 7 : digitCounters += 010000000;
```

```
                break;
            case 8 : digitCounters += 0100000000;
                break;
            case 9 : digitCounters += 01000000000;
            }
        for (int i = 0; i < 10; ++i, mask  <<= 3)
   {
   countervalue = digitCounters & mask;
   countervalue >>= 3 * i;
   if (countervalue > 1) cout << "Цифра "
                              << i
                              << " повторяется "
                              << countervalue << " раза\n";
   }
   return 0;
}
```

Результат работы программы:

Натуральное число [1000,9999]? 5775
Цифра 5 повторяется 2 раза
Цифра 7 повторяется 2 раза

Результат работы программы:

Натуральное число [1000,9999]? %775
Ошибка формата!

Как видим, под каждый счетчик здесь достаточно отвести всего 3 бита, а для чтения счетчика достаточно одной операции маскирования и двух подготовительных операций сдвига: вправо и влево.

Функции

> Пример 79

Одним из средств C++, предназначенных для поддержки императивной парадигмы программирования, являются функции. Недаром императивную парадигму зачастую называют процедурной. В первую очередь процедуры и функции имеют отношение к организации программы. Так, главная функция ***main()*** как непременный атрибут программ на C++ обеспечивает точку входа в программу и поддержку интерфейса командной строки, а например, процедуры и функции как средство задания способа выполнения операций приводят к разбиению программы на составные части. Отказ от повторной компиляции кода при сборке программы приводит, например, к таким формам организации объектного кода процедур и функций, как библиотеки.

Как и ранее, будем строить программу в виде одной единицы компиляции. В этом случае объявлять функции можно и в глобальной, и в локальной областях видимости, а вот определять функции можно только в глобальной области видимости.

Поначалу обратимся к программе в стиле C, где в глобальной области видимости наряду с определением функции ***main()*** будут представлены определения функций ***maximum***0 и ***print(),*** объявление и определение функции ***minimum***0, а в локальной области видимости функции ***main()*** будет представлено объявление функции ***print()***:

```
// C++ Функции
#include <cstdio>
#include <cmath>
int maximum(int m, int n)
{
  return m > n ? m : n;
}
double minimum(double, double);
int main()
{
  void print(char*, int);
  int a, b;
  double c;
  printf("a? ");
  if (!scanf("%d", &a))
  {
  printf("Ошибка формата!\n");
  return -1;
  }
  printf("b? ");
  if (!scanf("%d", &b))
  {
```

```
  printf("Ошибка формата!\n");
  return -1;
   }
print("maximum(a,b) = ", maximum(a, b));
c = minimum((double)a, pow((double)maximum (a, b), 2));
print ("c = ", (int)c);
return 0;
}

void print(char* pointer, int value)
{
   printf("%s%d\n", pointer, value);
}

double minimum(double a, double b)
{
return a < b ? a : b;
}
```

Результат работы программы:

```
a? 1
b? 2
maximum(a,b) = 2
c = 1
```

Напомним, что объявление функции в C++ зачастую называют ее прототипом. Как видим, при объявлении функции имена ее аргументов можно опускать.

При каждом вызове функции выделяется место в автоматической памяти под ее формальные аргументы, и каждому формальному аргументу присваивается значение соответствующего фактического аргумента, называемого также аргументом вызова. Семантика передачи аргументов идентична семантике инициализации. В частности, проверяется соответствие типов формальных и фактических аргументов и при необходимости выполняются либо стандартные, либо определенные пользователем преобразования типов.

Говорят, что аргументы вызова функций ***maximum***0 и ***minimum***(), так же как и второй аргумент вызова функции ***print(),*** передаются по значению. Это означает, что этим функциям доступны не сами аргументы вызова, а их локальные копии. Говорят, что первый аргумент вызова функции ***print()*** передается по адресу. Это означает, что этой функции доступен сам аргумент вызова.

Указание ***void*** в качестве типа возвращаемого значения означает, что функция не возвращает значения. В этом случае инструкцию-перехода ***return*** следует записывать без выражения, однако, как видим, она может и отсутствовать. В остальных случаях наличие инструкции-перехода ***return*** обязательно. Семантика возврата значения из функции идентична семантике инициализации. Возвращаемым значением функции можно и пренебречь, например, как это делалось ранее и делается сейчас здесь для библиотечной функции форматного вывода ***printf().***

> **Пример 80**

Чтобы функции передать аргумент вызова по адресу, в C++ помимо указателя на объект можно воспользоваться также и ссылкой на этот объект. Отметим, что ссылки в C++ были введены в основном для поддержки перегрузки стандартных операторов. Помимо очевидного применения ссылок для передачи аргументов зачастую ссылка используется и как возвращаемое значение. Одним из таких кандидатов на роль функции с аргументами вызова, передаваемых по ссылке, здесь может стать функция ***maximum().*** Что же до ее возвращаемого значения, то ссылка здесь пока неуместна.

Обратимся теперь к программе в стиле C++, где, как и ранее, в глобальной области видимости наряду с определением функции ***main()*** будут представлены определения функций ***maximum()*** и ***print(),*** объявление и определение функции ***minimum(),*** а в локальной области видимости функции ***main()*** будет представлено объявление функции ***print()***:

```
// C++ Функции
#include <iostream>
#include <cmath>
using namespace std;
int maximum(int& m, int& n)
{
  return m > n ? m : n;
}
double minimum(double, double);
int main()
{
  void print(char*, int);
  int a, b;
  double c;
  cout << "a? ";
  cin >> a;
  if (cin.fail())
  {
  cout << "Ошибка формата!\n";
  return -1;
  }
cout << "b? cin >> b;
if (cin.fail())
  {
  cout << "Ошибка формата!\n";
  return -1;
  }
```

```
print("maximum(a,b) = ", maximum(a, b));
c = minimum(double(a), pow(double(maximum(a, b)), 2));
print("c = ", int(c));
return 0;
}
void print(char* pointer, int value)
{
cout << pointer << value << endl;
}
double minimum(double x, double y)
{
return x < y ? x : y;
}
```

Результат работы программы:

```
a? 1
b? 2
maximum(a,b) = 2
c = 1
```

Применение ссылок для передачи аргументов, как правило, требует соблюдения не только известных мер предосторожности, но и самодисциплины. Так, применение ссылок на переменную для передачи аргументов "должно" означать, что функции разрешается модифицировать передаваемые ей аргументы, а вот применение ссылок на константу для передачи аргументов означает, что функции теперь запрещено модифицировать передаваемые ей аргументы. При попытке выстраивания подобной "защиты" не стоит, однако, удивляться тому, что теперь придется менять не только сигнатуру функции, но также и другие ее атрибуты. Для функции ***maximum()*** при сохранении типа возвращаемого значения это, например, может выглядеть так:

```
int maximum(const int& m, const int& n)
{
return const_cast<int5>(m > n ? m : n);
}
```

В противном случае это может выглядеть, например, и так:

```
const int& maximum(const int& m, const int& n)
{
return m > n ? m : n;
}
```

Напротив, применение ссылок на константу для передачи аргументов означает, что аргументы вызова теперь могут быть и константными выражениями, чего нельзя было допустить в случае ссылок на переменную.

> **Пример 81**

Оглядываясь назад, отметим, что, строя программу из одной единицы компиляции, пользователь предоставлял компилятору только исходный файл. Результат обработки препроцессором С исходного файла принято называть единицей трансляции. Все то, чего недоставало программе, было взято из заголовочных файлов стандартной библиотеки. Если строить программу в виде нескольких единиц компиляции, то одним из простых методов достижения согласованности объявлений в различных единицах трансляции является включение заголовочных файлов пользователя, содержащих информацию об интерфейсе, в исходные файлы с исполняемым кодом. Напомним, что для именования заголовочных файлов пользователя в директиве препроцессора С ***#include*** вместо угловых скобок следует использовать кавычки.

Поначалу обратимся к программе в стиле С, собираемой всего из двух единиц компиляции: заголовочного файла, где будут представлены определения функций ***maximum()*** и ***print(),*** и исходного включающего файла с исполняемым кодом, к которому препроцессором С будет добавлен код из заголовочного файла.

Представим первую единицу компиляции — заголовочный файл ***c_header.h:***

```
// C++ Заголовочный файл – c_header.h
#include <cstdio>
int maximum(int m, int n)
{
  return m > n ? m : n;
}
void print(char* pointer, int value)
{
  printf("%s%d\n", pointer, value);
}
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
# include "c_header.h"
# int main ()
{
  int a, b;
  printf("a? ");
  if (!scanf("%d", &a))
  {
  printf("Ошибка формата!\n");
  return -1;
  }
  printf("b? ");
```

```
   if (!scanf("%d", &b))
   {
   printf("Ошибка формата!\n");
   return -1;
   }
   print("maximum(a,b) = ", maximum(a, b));
   return 0;
}
```

Результат работы программы:

```
a? 1
b? 2
maximum(a,b) = 2
```

Отметим, что организацию программы в виде набора исходных файлов принято называть физической структурой программы. Считается, что физическое разбиение программы на различные файлы должно определяться в соответствии с ее логической структурой. Следует признать, что подход к физическому разбиению программы с единственным заголовочным файлом наиболее полезен, когда она имеет небольшой размер и не требуется отдельного использования ее частей.

Отметим также, что перекомпиляция заголовочного файла всякий раз при его включении рано или поздно может стать тем ощутимым препятствием на пути разработки и сопровождения программы, которое приведет к пересмотру ее физической структуры. Одним из приемлемых решений может стать стремление к соблюдению традиций, связанных с организацией заголовочных файлов стандартной библиотеки C++, в которых, как правило, содержатся только объявления, а не исполняемый код. Такой подход неизбежно приводит к раздельной компиляции частей программы, называемых по традиции модулями, что, в свою очередь, требует от пользователя усилий по организации защиты с помощью директив условной трансляции препроцессора C от повторного включения кода заголовочных файлов в исходные включающие файлы с исполняемым кодом.

Чтобы сделать возможной раздельную компиляцию модулей, пользователь должен предоставить объявления, дающие информацию о типах, необходимую для анализа единицы трансляции, отдельно от остальных частей программы. Все объявления должны быть согласованы между собой так же, как и в программе, состоящей из одного единственного исходного файла. Все несогласованности различных типов, которые не смог обнаружить компилятор в каждой отдельной единице компиляции, разрешаются компоновщиком на этапе связывания раздельно откомпилированных частей в одно целое — загружаемый код программы. Тем самым наряду с ошибками времени компиляции и ошибками времени выполнения, как видим, имеют место и ошибки времени компоновки.

> **Пример 82**

Обратимся теперь к программе в стиле C++, собираемой, как и ранее, всего из двух единиц компиляции: заголовочного файла, где будут представлены определения функций ***maximum()*** и ***print(),*** и исходного включающего файла с исполняемым кодом, к которому препроцессором C будет добавлен код из заголовочного файла. Памятуя о "традиции" применения ссылок пока только для передачи аргументов и о "запрете" применения ссылки как возвращаемого значения, объявим здесь аргументы функции ***maximum(),*** как и прежде, в виде ссылок на переменную и сохраним тип возвращаемого значения неизменным.

Представим первую единицу компиляции — заголовочный файл ***c++_header.h:***

```
// C++ Заголовочный файл – c++_header.h
#include <iostream>
using namespace std;
int maximum(int& m, int& n)
{
   return m > n ? m : n;
}
void print(char* pointer, int value)
{
   cout << pointer << value << endl;
}
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include "c++_header.h" int main ()
{
int a, b;
cout << "a? ";
cin >> a;
if (cin.fail())
   {
   cout << "Ошибка формата!\n";
   return -1;
   }
cout << "b? ";
cin >> b;
if (cin.fail())
   {
```

```
  cout << "Ошибка формата!\n";
  return -1;
  }
  print("maximum(a,b) = ", maximum(a, b));
  return 0;
}
```

Результат работы программы:

```
a? 1
b? 2
maximum(a,b) = 2
```

Чтобы перейти к раздельной компиляции, эту программу, даже невзирая на ее небольшой размер, следует разбить уже на три части: заголовочный файл, в котором будут содержаться объявления функций ***maximum()*** и ***print(),*** файл реализации, в котором будут содержаться определения функций ***maximum()*** и ***print(),*** и исходный файл с исполняемым кодом.

Заголовочный файл представляет собой интерфейс для пользователей, который будет включен в исходный файл с исполняемым кодом. Файл реализации — это независимый модуль с исполняемым кодом, для раздельной компиляции которого требуется, чтобы все объявления в нем были согласованы между собой. Как правило, каждому заголовочному файлу соответствует свой файл реализации. Для того чтобы в будущем повторно включаемый заголовочный файл не вызывал перекомпиляций своего файла реализации, в заголовочные файлы принято вставлять так называемые стражи включения (от слов ***include guards***), которые создаются при помощи директив условной трансляции препроцессора C.

В результате такого разбиения программы на составные части, как видим, каждый логический модуль представляет собой согласованный и вполне самодостаточный фрагмент. Первую единицу трансляции — файл реализации — можно компилировать независимо от второй единицы трансляции — исходного включающего файла вкупе с его заголовочным файлом. Полученные два объектных файла могут быть связаны компоновщиком в одно целое — загружаемый файл. Достраивая исходный файл с исполняемым кодом и не “ломая” интерфейс файла реализации, можно всякий раз компоновать новый загружаемый файл уже без перекомпиляций файла реализации. Представим первую единицу компиляции — заголовочный файл ***header.h:***

```
// C++ Заголовочный файл – header.h
#ifndef HEADER_H
#define HEADER_H // страж включения
int maximum(int&, int&);
void print(char*, int);
#endif // HEADER H
```

Представим вторую единицу компиляции — файл реализации ***implements.cpp:***

```
// C++ Файл реализации – implements.cpp
#include <iostream>
using namespace std;
int maximum(int&.m, int& n)
{
return m > n ? m : n;
}
void print(char* pointer, int value)
{
cout << pointer << value << endl;
}
```

Представим третью единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include "header.h"
#include <iostream>
int main()
{
using namespace std;
int a, b;
cout << "a? ";
cin >> a;
if (cin.fail())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
cout << "b? ";
cin >> b;
if (cin.fail ())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
print("maximum(a,b) = ", maximum(a, b));
return 0;
}
```

Все остальное определяется уже той средой, где строится сама программа.

> **Пример 83**

Одной из ярких иллюстраций механизма передачи аргументов по адресу является алгоритм перестановки двух объектов программы местами, применяемый, например, в алгоритмах сортировки. Нелишним будет напомнить, что перестановка на самом деле осуществляется на основе механизма обменивания значениями. Остановим свой выбор на той его версии, где результат достигается за три операции присваивания благодаря привлечению вспомогательной переменной, в роли которой, как правило, выступает временный объект.

Поначалу представим программу, где аргументы функции ***swap()*** будут объявлены как указатели на ***int:***

```
// C++ Функции
#include <iostream>
void swap(int* a, int* b)
{
  int c = *a;
  *a = *b;
  *b = c;
}
int main()
{
using namespace std;
int a(5), b(7);
cout << a << '\t' << b << endl;
swap(&a, &b);
cout << a << '\t' << b << endl;
return 0;
}
```

Результат работы программы:

```
5          7
7          5
8
```

Следует отметить, что “упрятывание” первой операции присваивания в операцию инициализации обусловлено самой синтаксической формой объявления временного объекта, поэтому здесь нельзя считать, что она “выпадает” из списка операций присваивания.

Четыре операции разыменования указателя в одном алгоритме из трех инструкций следует воспринимать как неизбежные накладные расходы, с которыми необходимо смириться, — другой альтернативы, увы, нет.

> **Пример 84**

Теперь представим программу, где аргументы функции ***swap()*** будут объявлены как ссылки на ***int.***

```
// C++ Функции
#include <iostream>
void swap(int& a, int& b)
{
int c = a;
a = b;
b = c;
}
int main()
{
using namespace std;
int a(5), b(7);
cout << a << '\t' << b << endl;
swap(a, b);
cout << a << '\t' << b << endl;
return 0;
}
```

Результат работы программы:

```
5          7
7          5
8
```

Отметим здесь, что за кажущейся простотой этого алгоритма в его инструкциях могут “скрываться” те же четыре, но неявные операции разыменования указателя, на этот раз константного. К такому осторожному высказыванию приходится прибегать из-за особенностей деталей реализации ссылки. Напомним, что ссылка, в отличие от указателя, не является объектом, над которым можно выполнять операции. Так, несмотря на форму записи операции над ссылкой, ни один оператор на самом деле не выполняет каких-либо действий над ней. С одной стороны, можно считать, что реализацией ссылки, например, может являться константный указатель, а с другой стороны, компилятор может оптимизировать ссылку таким образом, что во время исполнения вообще не будет объекта, представляющего ссылку.

> **Пример 85**

В алгоритмах обменивания значениями наряду с обычными объектами программы зачастую выступают и указатели на объекты. В этом случае аргументы функции, обменивающей друг с другом значения двух указателей, объявляются как ссылки на указатели.

Итак, в завершение представим программу, где аргументы функции ***swap()*** будут объявлены как ссылки на указатели на ***int***:

```
// C++ Функции
#include <iostream>
void swap(int*& a, int*& b)
{
int* c = a;
a = b;
b = c;
}
int main()
{
using namespace std;
int a(5), b(7);
int* p = &a;
int* q = &b;
cout << *p << '\t' << *q << endl;
swap (p, q);
cout << *p << '\t' << *q << endl;
return 0;
}
```

Результат работы программы:

```
5          7
7          5
8
```

Как видим, последние два примера наглядно демонстрируют, что для операций обменивания значениями применение ссылок по сравнению с указателями является оправданным и естественным актом. Использование ссылок в качестве аргументов здесь просто и интуитивно понятно, а использование с этой же целью указателей, напротив, воспринимается уже как анахронизм, если только не считать его данью традиции, связанной со стилем C.

> **Пример 86**

Тип ***enum*** — перечисление — один из двух типов C++, которые определяются пользователем. Другим таким типом, как известно, является класс. Перечисление является одним из удобных средств для объявления набора именованных целых констант. Если при объявлении элементов перечисления не указаны целочисленные инициализаторы, то значения именованных констант определяются по умолчанию порядком перечисления, начинаясь с 0 для первой и увеличиваясь на 1 для каждой следующей. Если для какого-либо элемента задается инициализатор, он всего лишь определяет начало упорядоченной по возрастанию последовательности значений для остальных элементов перечисления.

Представим программу в стиле C, где результат стрельбы по мишени в виде круга заданного радиуса и с заданными координатами его центра будет определяться по координатам выстрела с помощью функции ***shoot(),*** возвращаемое значение которой инициализируется одним из двух элементов перечисления — ***no*** или ***yes:***

```
// C++ Функции
#include <cstdio>
#include <cmath>
enum { no, yes };
int shoot(double x, double y, double x0, double y0, double r)
{
  return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? no : yes;
}
int main()
{
      if (shoot(0.5, 0.5, 0.5, 0.5, 2))
            printf("Да!\n");
      else
           printf("Нет!\n");
      return 0;
}
```

Результат работы программы:

Да!

Напомним, что в C отсутствует логический тип, поэтому, как видим, перечисление помогает восполнить этот “пробел” — именованные константы ***no*** и ***yes*** выступают в роли литералов ***false*** и ***true,*** являющихся значениями логического типа C++ ***bool.***

> **Пример 87**

Представим программу в стиле C++, где результат стрельбы по мишени в виде круга заданного радиуса и с заданными координатами его центра будет определяться по координатам выстрела с помощью функции ***shoot(),*** тип возвращаемого значения которой будет объявлен как ***bool,*** а последние три аргумента будут объявлены как аргументы по умолчанию:

```
// C++ Функции
#include <iostream>
#include <cmath>
bool shoot(double x, double y, double x0=0.0, double y0=0.0, double
r=1.0);
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false
: true;
}
int main()
{
using namespace std;
   if (shoot(0.5, 0.5))
      cout << "Да!” << endl;
   else
      cout << "Нет!" << endl;
   if (shoot(0.5, 0.5, 0.5))
      cout << "Да!" << endl;
   else
      cout << "Нет!" << endl;
   if (shoot(0.5, 0.5, 0.5, 0.5))
      cout << "Да!" << endl;
   else
      cout << "Нет!" << endl;
   if (shoot(0.5, 0.5, 0.5, 0.5, 2))
      cout << "Да!" << endl;
   else
      cout << "Нет!" << endl;
return 0;
}
```

Результат работы программы:

```
Да!
Да!
Да!
Да!
```

> **Пример 88**

Аргументы по умолчанию, как видим, предоставляют пользователю возможность вызова одной и той же функции с различным набором аргументов. Очевидно, что для поддержки такой гибкой формы вызова функции ее аргументы по умолчанию могут быть объявлены только в конце списка аргументов. Поскольку тип аргумента по умолчанию проверяется в месте объявления функции, а вычисляется в момент ее вызова, считается, что аргумент по умолчанию не может быть повторен или изменен в последующих объявлениях в одной и той же области видимости.

Продолжая иллюстрацию механизма объявления аргументов по умолчанию, представим теперь программу, где будет показано, каким же образом в глобальной области видимости следует записывать прототип и определение функции ***shoot(),*** позволяющей по координатам выстрела определять результат стрельбы по мишени в виде круга заданного радиуса и с заданными координатами его центра:

```
// C++ Функции
#include <iostream>
#include <cmath>
bool shoot(double x, double y, double x0=0.0, double y0=0.0, double
r=l. 0);
int main()
{
using namespace std;
 if (shoot(0.5, 0.5))
     cout << "Да!" << endl;
 else
     cout << "Нет! " << endl;
if (shoot(0.5, 0.5, 0.5, 0.5, 2))
     cout << "Да!" << endl;
 else
     cout << "Нет!" << endl;
return 0;
}
bool shoot(double x, double y, double x0, double y0, double r)
{
return pow((x — x0), 2) + pow((y — y0), 2) > pow(r, 2) ? false : true;
}
```

Результат работы программы:

```
Да!
Да!
```

Если программа собирается из нескольких единиц компиляции, аргументы по умолчанию, как правило, объявляются в прототипах функций, которые размещаются в общедоступном заголовочном файле. В каждом исходном файле с исполняемым кодом теперь могут быть представлены как обычные определения функции, так и те, где допускается объявление аргументов по умолчанию.

Поначалу рассмотрим случай, когда в исходном файле с исполняемым кодом будет представлено обычное определение функции ***shoot().***

Представим первую единицу компиляции — заголовочный файл ***header.h.***

```
// C++ Заголовочный файл – header.h
#ifndef HEADER_H
fdefine HEADER_H // страж включения
bool shoot(double x, double y, double x0=0.0, double y0=0.0, double
r=1.0);
#endif // HEADER_H
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include <iostream>
#include <cmath>
#include "header.h"
bool shoot(double x, double y, double x0, double y0, double r)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false : true;
}
int main()
{
using namespace std;
      if (shoot(0.5, 0.5))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
      if (shoot(0.5, 0.5, 0.5, 0.5, 2))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
return 0;
}
```

Как видим, в такой организации программы нет ничего необычного.

Рассмотрим теперь случай, когда в исходном файле с исполняемым кодом будет представлено определение функции ***shoot(),*** где допускается объявление аргументов по умолчанию.

Представим первую единицу компиляции — заголовочный файл ***header.h:***

```
// C++ Заголовочный файл – header.h #ifndef HEADER_H
#define HEADER_H // страж включения
bool shoot(double x, double y, double x0, double y0 double r=1.0);
#endif // HEADER_H
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include <iostream>
#include <cmath>
# include "header.h"
bool shoot(double x, double y, double x0=1.0, double y0=1.0, double r)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false : true;
}
int main()
{
using namespace std;
  if (shoot(0.5, 0.5))
      cout << "Да!" << endl;
  else
      cout << "Нет!" << endl;
  if (shoot(0.5, 0.5, 0.5, 0.5, 2))
      cout << "Да!" << endl;
  else
  cout << "Нет!" << endl;
return 0;
}
```

Как видим, в той области видимости, где определяется функция ***shoot(),*** все три ее аргумента по умолчанию — ***x0, y0*** и ***r*** — расположены правее тех, которые не имеют значений по умолчанию, поэтому требование присваивать такие значения справа налево для аргументов ***x0 и y0*** здесь не нарушается.

Однако возможны и отступления от общепринятого правила, поэтому поначалу рассмотрим случай, когда в заголовочном файле, как и ранее, будут представлены определения функций, но на этот раз с аргументами по умолчанию.

Представим первую единицу компиляции — заголовочный файл ***header.h:***

```
// C++ Заголовочный файл – header.h
#ifndef HEADER_H
#define HEADER_H // страж включения
bool shoot(double x, double y, double x0=0.0, double y0=0.0, double
r=1.0)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false : true;
}
#endif // HEADER_H
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include <iostream>
#include <cmath>
#include "header.h"
int main()
{
using namespace std;
      if (shoot(0.5, 0.5))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
      if (shoot(0.5, 0.5, 0.5, 0.5, 2))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
return 0;
}
```

Зачастую требуется, чтобы в конкретных приложениях пользователь мог бы по своему усмотрению выполнять различные настройки одной и той же универсальной функции с аргументами по умолчанию, прототип которой, как это принято, должен размещаться в общедоступном заголовочном файле. Очевидно, что для поддержки такого механизма тоже следует отказываться от правила объявлять аргументы по умолчанию только в прототипах функций. Примерами подобного рода изобилует и сама стандартная библиотека C++.

Итак, в завершение рассмотрим случай, когда в исходном файле с исполняемым кодом будет выполняться настройка аргументов по умолчанию для “универсальной” функции ***shoot()*.**

Представим первую единицу компиляции — заголовочный файл ***header.h***:

```
// C++ Заголовочный файл – header.h
#ifndef HEADER_H
#define HEADER_H // страж включения
bool shoot(double x, double y, double x0, double y0, double r); #endif
// HEADER_H
```

Представим вторую единицу компиляции — исходный файл с исполняемым кодом:

```
// C++ Функции
#include <iostream>
#include <cmath>
#include "header.h"
bool shoot(double x, double y, double x0=0.0, double y0=0.0, double
r=1.0)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false : true;
}
int main()
{
using namespace std;
      if (shoot(0.5, 0.5))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
      if (shoot(0.5, 0.5, 0.5, 0.5, 2))
            cout << "Да!" << endl;
      else
            cout << "Нет!" << endl;
return 0;
}
```

Далее будет сказано, как в одной области видимости подобную настройку можно будет осуществлять и для перегруженных функций.

> **Пример 89**

Функцию можно определить со спецификатором ***inline.*** Такие функции получили название встроенных. Спецификатор ***inline*** указывает компилятору, что он должен пытаться всякий раз в месте вызова функции генерировать ее код, а не вызывать функцию посредством обычного механизма вызова. Встраиваемая функция должна быть небольшой по размеру, должна быть часто вызываемой, не должна содержать цикла и не должна быть рекурсивной. Чтобы сделать возможным встраивание кода функции, ее определение должно находиться в каждой единице трансляции, где она используется. Поэтому рекомендуется помещать определение встраиваемой функции в заголовочный файл и включать его во все файлы, где есть обращения к ней.

Представим программу, где уже знакомая по предыдущим примерам функция ***maximum()*** может стать встроенной:

```
// C++ Функции
#include <iostream>
inline int maximum(int m, int n)
{
return m > n ? m : n;
}
int main()
{
using namespace std;
int a, b, c, d;
cout << "Четыре целых числа? ";
cin >> a >> b >> c >> d;
      if (cin.fail())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
cout << maximum(a, b) << '\t' << maximum(c, d) << endl;
return 0;
}
```

Результат работы программы:

```
Четыре целых числа? 1  2 3 4
2          4
```

Считается, что хороший компилятор может в месте вызова сгенерировать две инструкции: ***a > b ? a : b и c > d ? c : d.*** Однако не стоит удивляться, что он может этого и не сделать.

> Пример 90

Функция, которая явно (прямо) или неявно (косвенно) вызывает сама себя, называется рекурсивной. Такая функция обязательно должна определять условие окончания, в противном случае рекурсия будет продолжаться до тех пор, пока не исчерпается стек. Одним из ярких классических примеров прямой рекурсии является алгоритм вычисления факториала.

Представим программу, где для вычисления факториала неотрицательного целого числа будет использоваться рекурсивная функция ***factorial()***:

```
// C++ Функции
#include <iostream>
long int factorial(int n)
{
if (n == 0)
      return 1;
else
      return n * factorial(n — 1);
}
int main()
{
using namespace std;
int n;
do
 {
      cout << "n? ";
      cin >> n;
      if (cin. fail ())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
}
while (!(n >= 1 && n <= 12));
cout << "n! = " << factorial(n) << endl;
return 0;
}
```

Результат работы программы:

```
n? 12
n! = 479001600
```

Как видим, здесь для функции ***factorial***0 условием окончания является равенство нулю аргумента вызова.

> **Пример 91**

Обратимся теперь к другому, не менее яркому, классическому примеру прямой рекурсии, а именно к эвристическому алгоритму отыскания действительных корней нелинейных уравнений одного переменного методом дихотомии. Напомним, что в основе этого метода лежит известная эвристика: если на границах некоторого отрезка функция одного переменного меняет знак, то существует, по крайней мере, одна точка на этом отрезке, в которой функция обращается в нуль.

Представим программу, где для отыскания действительного корня нелинейного уравнения ***f(x)*** = ***0*** будет использоваться рекурсивная функция ***root():***

```
// C++ Функции
#include <iostream>
#include <cmath>
#include <iomanip>
double root(double f(double), double x_left, double x_right, double
tolerance)
{
double f_left = f(x_left), f_right = f(x_right), x_root, f_root;
x_root = (x_left + x_right) / 2.0;
f_root = f (x_root);
if (fabs (f_root) > tolerance)
      return f_left * f_root <0.0 ? root(f, x_left, x_root, tolerance)
      : root(f, x_root, x_right, tolerance);
else
      return x_root;
}
double f(double x)
{
return exp(x) – exp(-x) – 2.0;
}
int main()
{
using namespace std;
double x_left = 0.0, x_right = 1.0, tolerance = 1e-6;
if (f(x_left)* f(x_right) > 0.0)
      cout << "На границах отрезка функция не меняет знак!" << endl;
else
      cout << setprecision(7) << "Корень = "
 << root(f, x_left, x_right, tolerance) << endl;
return 0;
}
```

Результат работы программы:

```
Корень = 0.8813734
```

Точность отыскания действительного корня нелинейного уравнения ***f(x) = 0*** здесь определяется значением параметра ***tolerance*** — так называемого “нуля”. Как видим, для функции ***root()*** условием окончания является результат операции “не больше нуля” для модуля функции ***f(x_root),*** где ***x root*** всякий раз вычисляется как середина отрезка локализации корня. Благодаря тому, что отрезок локализации корня после каждой очередной итерации уменьшается вдвое, метод дихотомии в вычислительной математике иногда называют также методом деления отрезка пополам или методом бисекций.

Зачастую рекурсивные алгоритмы в качестве одного из своих параметров требуют привлечения функций, поэтому в императивных языках программирования весьма востребованы специальные средства для поддержки такого механизма. Не являются исключением и такие языки, как C и C++, где в качестве такого средства выступают указатели на функцию. Далее указателям на функцию будет посвящен свой раздел, а пока необходимо отметить, что функцию можно вызывать не только с помощью ее имени, но также и с помощью указателя на нее.

Как видим, первый аргумент рекурсивной функции ***root()*** объявлен как функция. Такие аргументы иногда называют аргументами-функциями, подчеркивая их особую роль среди остальных аргументов в сигнатуре функции. Кстати, подобным образом зачастую выделяют также аргументы-ссылки и аргументы-указатели. Считается, что формальный аргумент-функцию следует трактовать как указатель на функцию, такая трансформация типа всегда происходит автоматически.

А вот при вызове функции ***root()*** ее первый аргумент, как видим, является просто именем функции ***f.*** В соответствии с принципами строгой типизации, считается, что механизм точного соответствия формальных аргументов функции ее фактическим аргументам опирается на следующее правило: фактический аргумент типа “функция” автоматически приводится к типу “указатель на функцию”. Иными словами, при вызове функции ***root()*** применяется тривиальное преобразование имени функции в указатель на функцию, в результате чего аргумент вызова / типа “функция” будет приведен к типу “указатель на функцию”.

Итак, при вызове функции ***root()*** между формальным аргументом типа “функция”, ставшим указателем на функцию, и фактическим аргументом, приведенным к типу “указатель на функцию”, будет установлено точное соответствие.

> **Пример 92**

Когда функции выполняют схожие операции над объектами различных типов, может оказаться удобным присвоить им одно и то же имя. Использование одного имени для операции, выполняемой с различными типами, называется перегрузкой.

Две функции в C++ называются перегруженными, если они имеют одинаковое имя, объявлены в одной и той же области видимости, но имеют разную сигнатуру. Процесс поиска подходящей функции из множества перегруженных заключается в нахождении наилучшего соответствия типов формальных и фактических аргументов.

Поначалу представим программу, где механизм перегрузки будет основываться на проверке точного соответствия типов аргументов функций ***abs Value(),*** вызываемых для объектов арифметических типов, чтобы вычислить модуль числа:

```
// C++ Функции
#include <iostream>
using namespace std;
short int absValue(short int x)
{
cout << "short int\t";
return x < 0 ? -x : x;
}
int absValue(int x)
{
cout << "int\t\t";
return x < 0 ? -x : x;
}
long int absValue(long int x)
{
cout << "long int\t";
return x < 0 ? -x : x;
}
float absValue(float x)
{
cout << "float\t\t";
return x < 0 ? -x : x;
}
double absValue(double x)
{
cout << "double'st\t";
return x < 0 ? -x : x;
}
```

```
long double absValue(long double x)
{
cout << "long double\t";
return x < 0 ? -x : x;
}
int main()
{
cout << absValue((short int)10) << endl;
cout << absValue(10) << endl;
cout << absValue(10L) << endl;
cout << absValue((float)10.0) << endl;
cout << absValue(10.0) << endl;
cout << absValue ((long double)10.0) << endl;
return 0;
}
```

Результат работы программы:

```
short int           10
int                 10
long int            10
float               10
double              10
long double         10
```

Отметим, что критерий точного соответствия типов является первым по порядку из набора критериев, заложенных в механизм перегрузки для проверки наилучшего соответствия типов формальных и фактических аргументов функций. Критерий точного соответствия типов также включает в себя и соответствие, достигаемое тривиальными преобразованиями типов, одно из которых уже было представлено на примере имени функции и указателя на функцию. К другим примерам тривиальных преобразований относятся: ***lvalue*** и ***rvalue,*** имя массива и указатель, ***имя типа*** и ***const имя типа.*** Напомним, что ***rvalue*** (от слов ***right value)*** — это все те выражения, что не являются ***lvalue.***

Как видим, для каждого вызова ***absValue***0 нашлась единственная перегруженная функция из множества предложенных благодаря проверке самого первого критерия из ранжированного набора — достижения точного соответствия типов.

Отметим также, что для компоновщика все перегруженные функции на этапе компоновки кода “выглядят” как функции с уникальными именами. Декорирование имен перегруженных функций в уникальные внутренние имена в преддверии этапа их компоновки — это забота компилятора по обеспечению безопасного связывания.

> **Пример 93**

Перейдем ко второму критерию наилучшего соответствия типов формальных и фактических аргументов перегруженных функций, связанному с “продвижением” типов фактических аргументов. “Продвижение” типа означает его расширение, это одно из неявных преобразований, сохраняющих значение арифметических типов, которые помимо механизма перегрузки выполняются также и в арифметических и логических операциях, чтобы привести операнды к “естественным” размерам.

Теперь представим программу, где механизм перегрузки будет основываться на проверке соответствия, достигаемого “продвижением” типов аргументов функций ***absValue***0, вызываемых, как и ранее, для объектов арифметических типов, чтобы вычислить модуль числа:

```
// C++ Функции #include <iostream> using namespace std;
int absValue(int x)
{
cout << "int\t\t";
return x < 0 ? -x : x;
}
long int absValue(long int x)
{
cout << "long int\t";
return x < 0 ? -x : x;
}
double absValue(double x)
{
cout << "double\t\t";
return x < 0 ? -x : x;
}
long double absValue(long double x)
{
cout << "long double\t";
return x < 0 ? -x : x;
}
int main()
{
cout << absValue((short int)10) << endl;
cout << absValue (10) << endl;
cout << absValue(10L) << endl;
```

```
cout << absValue((float)10.0) << endl;
cout << absValue (10.0) << endl;
cout << absValue((long double)10.0) << endl;
return 0;
}
```

Результат работы программы:

```
int                10
int                10
long int           10
double             10
double             10
long double        10
```

Здесь, как видим, тип ***short int*** "продвигается" в тип ***int,*** а тип ***float*** — в тип ***double.*** К другим примерам "продвижения" арифметических типов относятся: ***bool*** в ***int, char, signed char, unsigned char, unsigned short int*** расширяются в ***int,*** если ***int*** может представить все значения исходных типов, и в ***unsigned int*** в противном случае; ***double*** в ***long double.*** Тип ***wcharjt*** или типы перечислений "продвигаются" в первый из следующих типов, который может представить все значения из базовых типов: ***int, unsigned int, long int*** или ***unsigned long int.***

Следует отметить, что "продвижения" составляют неотъемлемую часть обычных арифметических преобразований, которые выполняются над операндами бинарного оператора, чтобы привести их к общему типу, который потом используется как тип результата.

Последующие два критерия связаны с преобразованиями типов аргументов вызова перегруженных функций, отличных от тривиальных. Так, третий критерий основан на проверке соответствия, достигаемого путем стандартных преобразований, а вот четвертый — на проверке соответствия, достигаемого при помощи преобразований, определяемых пользователем. И, наконец, последний критерий основан на проверке соответствия за счет многоточия ... в объявлении функций.

Прежде чем перейти к третьему критерию, отметим, что для преобразований, определяемых пользователем, можно воспользоваться операторами преобразования в виде операторных функций-членов класса или их "аналогами" в виде обычных именованных функций-членов класса. И говорить об этом можно только в контексте объектно-ориентированной парадигмы программирования, но уже не здесь. Что же касается той роли, которую отводят лексеме ... в сигнатуре функций, то об этом будет сказано уже в следующем разделе, посвященном функциям с неуказанным количеством аргументов.

> **Пример 94**

Итак, в завершение этого обзора представим программу, где механизм перегрузки будет основываться на проверке соответствия, достигаемого путем стандартных преобразований типов аргументов функций ***absValue(),*** вызываемых, как и ранее, для объектов арифметических типов, чтобы вычислить модуль числа:

```
// C++ Функции
#include <iostream>
using namespace std;
double absValue(double x)
{
cout << "double\t\t";
return x < 0 ? -x : x;
}
int main()
{
cout << absValue((short int)10) << endl;
cout << absValue(10) << endl;
cout << absValue(10L) << endl;
cout << absValue((float)10.0) << endl;
return 0;
}
```

Результат работы программы:

```
double          10
double          10
double          10
double          10
```

Помимо арифметических типов стандартным преобразованиям могут подвергаться и другие типы: например, указатели на произвольные типы в тип ***void*,*** указатели на производные типы в указатели на базовые типы. Здесь же имеют место стандартные преобразования знаковых интегральных типов в беззнаковые, целого значения 0 к типу указателя, а также перечислимого типа в целый или логический типы.

Если соответствие типов формальных и фактических аргументов может быть получено двумя способами на одном и том же уровне критериев, вызов функции считается неоднозначным и отвергается.

> **Пример 95**

Оглядываясь назад, еще раз вернемся к тем программам, где результат стрельбы по мишени в виде круга заданного радиуса и с заданными координатами его центра определялся по координатам выстрела с помощью функции ***shoot().***

Представим программу, где результат стрельбы по мишени будет определяться с помощью перегруженных функций ***shoot(),*** отбор устоявшей функции среди которых будет завершен после проверки первого критерия разрешения механизма перегрузки:

```
// C++ Функции
#include <iostream>
#include <cmath>
bool shoot(double x, double y)
{
return pow(x, 2) + pow(y, 2) > pow(1.0, 2) ? false : true;
}
bool shoot(double x, double y, double x0, double y0)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(1.0, 2) ? false : true;
}
bool shoot(double x, double y, double x0, double y0, double r)
{
return pow((x – x0), 2) + pow((y – y0), 2) > pow(r, 2) ? false : true;
}
int main()
{
using namespace std;
   if (shoot(0.5, 0.5))
      cout << "Да!" << endl;
   else
      cout << "Нет!" << endl;
   if (shoot(0.5, 0.5, 0.5, 0.5))
      cout << "Да!" << endl;
   else
      cout << "Нет!" << endl;
   if (shoot(0.5, 0.5, 0.5, 0.5, 2.0))
      cout << "Да!" << endl;
   else
   cout << "Нет!" << endl;
return 0;
}
```

Результат работы программы:

```
Да!
Да!
Да!
```

Как видим, такой подход к “выстраиванию” сценария стрельбы по мишени при помощи перегруженных функций требует от пользователя большей прозорливости по сравнению с функцией с аргументами по умолчанию.

Очевидно, что наряду с проверкой точного соответствия типов аргументов здесь легко “переключиться” и на проверку соответствия, достигаемого путем стандартных преобразований типов аргументов. Например, такой вызов функции ***shoot():***

```
if (shoot(0.5, 0.5, 0.5, 0.5, 2))
    cout << "Да!" << endl;
else
    cout << "Нет!" << endl;
```

позволит “переключиться” при отборе устоявшей функции с первого критерия на третий. Стандартные преобразования для арифметических типов тем и хороши, что они естественны по своей природе, а потому и “незаметны”. Было бы неоправданным стремление забывать об этом их свойстве. В практике программирования нередки случаи, когда стандартные преобразования сознательно игнорируются в угоду стилю, как, например, в этом учебном примере.

Очевидно также, что инициировать проверку еще и для второго критерия здесь не следует, достаточно ограничиться первым и третьим.

> **Пример 96**

Для некоторых функций в сигнатуре невозможно указать количество и типы всех аргументов. В этом случае прибегают к лексеме ... (многоточие), записывая ее либо единственным, либо последним элементом списка аргументов, что означает “и может быть еще несколько аргументов”. Считается, что сам разделитель запятая перед многоточием необязателен. О таких функциях говорят как о функциях с неуказанным количеством аргументов. Зачастую такие функции получают информацию о типах и количестве фактических аргументов по значению явно объявленных аргументов. Типичным применением многоточий является создание интерфейса для стандартных библиотечных функций С, например, из заголовочных файлов ***<cstdio>*** и ***<stdio.h>.***

В заголовочных файлах ***<cstdarg>*** и ***<stdarg.h>*** представлен набор стандартных макросов для доступа к неспецифицированным аргументам в таких функциях. Ключевая идея заключается в том, чтобы с помощью некоторого “умного” указателя “извлекать” из стека аргументы вызова, обладая информацией о типе каждого из них. Настройка указателя на тип каждого ожидаемого аргумента вызова выполняется во времени компиляции, а его извлечение — во времени выполнения. Поскольку компилятор “выпадает” из процесса стандартной проверки и преобразований типа, то теперь вся ответственность за поведение кода перекладывается на пользователя.

Зададимся целью организовать поиск минимума среди аргументов вызова для функций с неуказанным количеством аргументов, воспользовавшись поддержкой макросов ***valist, vastart, vaarg*** и ***vaend*** из ***<cstdarg>*** или ***<stdarg.h>.***

Поначалу представим программу, где каждая функция с неуказанным количеством аргументов будет настроена на обработку заданного количества своих аргументов вызова, тип которых либо точно соответствует предполагаемому, либо может быть преобразован путем расширения:

```
// C++ Функции
#include <iostream>
#include <cstdarg>
using namespace std;
int minlnt(int argCounter, ...)
{
  if (argCounter < 1)
  {
  cout << "Недопустимое значение первого аргумента?" << endl;
  return 0;
  }
  else
  {
  int nextArg;
  va_list pointer;
  va_start(pointer, argCounter);
  int min = va arg(pointer, int);
```

```
  --argCounter;
      for (; argCounter; —argCounter)
      {
      nextArg = va_arg(pointer, int);
      min = min < nextArg ? min : nextArg;
      }
va_end(pointer);
return min;
   }
}

long int minLonglnt(int argCounter, ...)
{
if (argCounter < 1)
      {
      cout << "Недопустимое значение первого аргумента!" << endl;
      return 0L;
      }
else
      {
long int nextArg;
va_list pointer;
va_start(pointer, argCounter);
long int min = va_arg(pointer, long int);
--argCounter;
for (; argCounter; —argCounter)
      {
      nextArg = va_arg(pointer, long int);
      min = min < nextArg ? min : nextArg;
      }
va_end(pointer);
return min;
      }
}

double minDouble(int argCounter, ...)
{
if (argCounter < 1)
{
cout << "Недопустимое значение первого аргумента!" << endl;
return 0.0;
}
      else
      {
double nextArg;
va_list pointer;
va_start(pointer, argCounter);
dovible min = va_arg (pointer, double);
—argCovinter;
```

```
for (; argCounter; —argCounter)
      {
      nextArg = va_arg(pointer, double);
      min = min < nextArg ? min : nextArg;
      }
va_end(pointer);
return min;
      }
}

int main()
{
  cout  <<  minInt(1, 1) << endl;
  cout  <<  minInt(2, 2,   3)  << endl;
  cout << minInt(2, (short int)3, (short int)4) << endl;
  cout  <<  minLongInt(3, 10L,   20L, 30L)  << endl;
  cout  <<  minDouble(3,    10.0, -20.0,30.0)    << endl;
  cout << minDouble(2, (float)10.0, -(float)30.0) << endl;
return 0;
  }
```

Результат работы программы:

```
1
2
3
10
-20
-30
```

Итак, чтобы приступить к обработке списка неуказанных аргументов, необходимо выполнить две подготовительные процедуры: с помощью макроса ***valist*** объявить указатель ***pointer*** для извлечения аргументов вызова из стека, а затем уже с помощью макроса ***vastart*** “подготовиться” к последующим операциям извлечения, настроив указатель ***pointer*** на первый неуказанный аргумент. Каждый шаг по извлечению аргумента вызова, осуществляемый с помощью макроса ***vaarg,*** как видим, требует явного указания типа. После завершения процедуры извлечения следует осуществить вызов макроса ***va_end,*** для того чтобы привести стек в то состояние, которое было до его модификации благодаря вызову макроса ***va start.*** Считается, что модификация стека может быть выполнена таким образом, что станет невозможен нормальный выход из функции.

От семейства функций, настроенных на обработку своих неуказанных аргументов, тип которых лег в основу именования каждой из них, теперь легко перейти к одной единственной функции, если добавить в ее сигнатуру еще один параметр, который будет указывать на тип аргументов вызова, но об этом уже в следующем примере.

> **Пример 97**

Итак, в завершение иллюстрации идеи обработки списка неуказанных аргументов для поиска минимума среди аргументов вызова представим программу с функцией ***minimum(),*** настроенной на распознавание типа неуказанных аргументов:

```
// C++ Функции
#include <iostream>
#include <cstdarg>
void* minimum(char sign, int argCounter ...)
{
   if (argCounter < 1)
return 0; // Недопустимое значение второго аргумента!
   else {
   va_list pointer;
   va_start(pointer, argCounter);
   if (sign == 'I')
            {
            int nextArg;
            int* pointerOnMin = new int;
            *pointerOnMin = va_arg(pointer, int);
            --argCounter;
            for (; argCounter; —argCounter)
                  {
                  nextArg = va_arg(pointer, int);
                  *pointerOnMin = *pointerOnMin < nextArg ?
            *pointerOnMin : nextArg;
                  }
   return pointerOnMin;
            }
   if (sign = 'L')
   {
   long int nextArg;
   long int* pointerOnMin = new long int;
   *pointerOnMin = va_arg (pointer, long int);
   --argCounter;
   for (; argCounter; —argCounter)
      {
      nextArg = va_arg(pointer, long int);
      *pointerOnMin = *pointerOnMin < nextArg ? *pointerOnMin
: nextArg;
      }
   return pointerOnMin;
   {
   if (sign = 'D’)
   {
```

```
   double nextArg;
   double* pointerOnMin = new double;
   *pointerOnMin = va_arg(pointer, double);
   --argCounter;
   for (; argCounter; --argCounter)
      {
      nextArg = va_arg(pointer, double);
      *pointerOnMin = *pointerOnMin < nextArg ? *pointerOnMin
      : nextArg;
      }
   return pointerOnMin;
      }
      va_end(pointer);
      }
}

int main()
{
using namespace std;
if (int* p = (int*)minimum('I', 1, 1)) cout << *p << endl;
if (long int* p = (long int*)minimum('L', 2, 10L, 20L) )
      cout << *p << endl;
if (double* p = (double*) minimum ('D' , 3, 10.0, -20.0, 30.0))
      cout << *p << endl;
return 0;
}
```

Результат работы программы:

```
1
10
-20
```

Как видим, чтобы воспользоваться ***void*** в качестве базового типа для указателей на динамические объекты неизвестного типа, необходимо применять операции явного преобразования типа указателя.

> **Пример 98**

Функция, как и переменная, может стать тем объектом программы, к которому можно “привязать” указатель. В этом случае наряду с непосредственным вызовом возможен и косвенный вызов функции. Объявление указателя на функцию сочетает в себе правила объявления встроенных указателей и функций. Аргументы указателей на функции объявляются точно так же, как и аргументы самих функций. Правила передачи аргументов при вызове функций через указатели те же самые, что и при непосредственном вызове функции. Отличие лишь в том, что объявляющая часть, состоящая из имени и префиксного оператора объявления *, заключается в скобки О, при инициализации указателя на функцию необязательно пользоваться оператором & для получения адреса функции, необязательно также и разыменование указателя на функцию с помощью оператора * при косвенном вызове функции, во всех операциях присваивания требуется точное соответствие типов и сигнатур.

Обратимся к алгоритму вычисления факториала и представим программу, где вызовы функций ***recursiveFactorial()*** и ***iterative Factorial()*** будут осуществляться при помощи указателя на функцию:

```
// C++ Функции
#include <iostream>
long int recursiveFactorial(short int n)
{
if (n = 0)
      return 1;
else
      return n * recursiveFactorial(n — 1);
}

long int iterativeFactorial(short int n)
{
if (n = 0) return 1;
      else
      {
      long int p = 1;
      for (int k = 1; k < n; p *= ++k);
return p;
}
}

int main()
{
using namespace std;
short int n;
long int(*functionPointer)(short int) = recursiveFactorial;
```

```
do
      {
      cout << "n? ";
      cin >> n;
      if (cin.fail())
            {
            cout << "Ошибка формата!\n";
            return -1;
            }
      }
while (!(n >= 1 && n <= 12));
cout << "n! = " << functionPointer(n) << endl;
functionPointer = iterativeFactorial;
cout << "n! = " << functionPointer(n) << endl;
return 0;
}
```

Результат работы программы:

```
n? 12
n! = 479001600
n! = 479001600
```

Следует отметить, что C, а вместе с ним и C++, иногда “ругают” за синтаксис объявлений, особенно тех, которые содержат в себе указатели на функции. Недаром такого рода объявления, как и сами типы, называют сложными. Далее будет сказано об одном из приемлемых способов определения синонимов сложных типов при помощи директивы ***typedef***

В практике императивного программирования указатели на функции составляют основу многих инструментальных средств, ярким примером среди которых являются реализации систем меню или полиморфных процедур.

> **Пример 99**

Указатели на функции зачастую выступают в качестве аргументов функций, если, например, необходимо выполнить указанную операцию над данными. Функцию, для косвенного вызова которой объявляется аргумент-указатель, называют функцией обратного вызова (от слов ***callback function).*** Функции обратного вызова — это одно из средств, используемых для многократного выполнения некоторой операции и не только над данными. Например, в алгоритмах сортировки стандартной библиотеки C функции обратного вызова выполняют операцию сравнения данных, тип которых им заранее неизвестен. Функции обратного вызова могут использоваться, например, для уведомления приложений с графическим интерфейсом о действиях пользователя или
о наступлении каких-либо событий.

Продолжая иллюстрацию косвенного вызова функций, представим программу, где указатель на функцию станет одним из аргументов функции ***result(),*** позволяющей осуществлять поиск экстремума среди целочисленных данных:

```
// C++ Функции
#include <iostream>
int result(int m, int n, int(*functionPointer)(int, int))
{
   return functionPointer(m, n);
}

int min(int m, int n)
{
   return m < n ? m : n;
}

int max(int m, int n)
{
   return m > n ? m : n;
}

int main()
{
using namespace std;
int m, n;
cout << "m? ";
cin >> m;
if (cin.fail())
   {
   cout << "Ошибка формата!\n";
   return -1;
   }
cout << "n? ";
cin >> n;
```

```
if (cin.fail())
  {
cout << "Ошибка формата!\n";
return -1;
  }
cout << "min(m,n) = " << result(m, n, min) << endl;
cout << "max(m,n) = " << result(m, n, max) << endl;
return 0;
}
```

Результат работы программы:

```
m? 1
n? 2
min(m,n) = 1
max(m,n) = 2
```

Как видим, если ранее объявление указателя на функцию в локальной области видимости функции ***main()*** из-за сложности синтаксиса воспринималось как еще не совсем привычная процедура, то его объявление как аргумента функции вызывает здесь уже другие эмоции.

Для задания синонима типа C++ в наследство от C досталась директива ***typedef*** с помощью которой можно “преодолевать” излишнюю громоздкость выражений при объявлении объектов сложных типов. В практике императивного программирования на C для задания синонимов характерных разновидностей сложных типов с помощью директивы ***typedef*** даже был выработан свой уникальный стиль именования, который используется до сих пор уже не одним поколением программистов. Остается лишь сожалеть, что в учебных примерах не всегда удается следовать этому стилю.

> **Пример 100**

Определяя синоним типа “указатель на функцию” при помощи директивы ***typedef,*** можно еще на один шаг приблизиться к привычной для функций форме объявления ее аргументов — ***имя типа имя аргумента функции.*** Аргументы типа “указатель на функцию”, как и сами функции, своим именем обычно всегда “указывают” на операцию, для выполнения которой их и объявляют, в противном случае такого рода код программы вызывает немалые затруднения при его сопровождении. По-разному можно относиться к такой традиции, однако желательно ее соблюдать.

Итак, завершая иллюстрацию косвенного вызова функций, представим программу, где тип аргумента ***compare*** функции ***result(),*** позволяющей осуществлять поиск экстремума среди целочисленных данных, будет объявлен как указатель на функцию при помощи директивы ***typedef:***

```
// C++ Функции
#include <iostream>
typedef int(*functionPointer)(int, int);
int result(int m, int n, functionPointer compare)
{
  return compare(m, n);
}

int min(int m, int n)
{
  return m < n ? m : n;
}

int max(int m, int n)
{
  return m > n ? m : n;
}

int main()
{
using namespace std;
int m, n;
cout << "m? ";
cin >> m;
if (cin.fail())
  {
  cout << "Ошибка формата!\n";
  return -1;
  }
cout << "n? ";
cin >> n;
```

```
if (cin.fail())
  {
cout << "Ошибка формата!\n";
return -1;
  }
cout << "min(m,n) = " << result(m, n, min) << endl;
cout << "max(m,n) = " << result(m, n, max) << endl;
return 0;
}
```

Результат работы программы:

```
m? 1
n? 2
min(m,n) = 1 max (m,n) = 2
```

Как видим, благодаря использованию директивы ***typedef*** здесь удалось добиться не только ясности кода, но также и строгости стиля в интерфейсе функции ***result().*** “Маскируя” указатель на функцию, можно добиваться и большего, подобными примерами изобилуют не только стандартные библиотеки C и C++.

Необходимо отметить, что указатели на функцию имеют большое значение в C, а вот в C++ они используются реже. Например, в обобщенных алгоритмах стандартной библиотеки C++ вместо указателей на функцию используются объекты-функции (или функторы) — объекты классов, которые благодаря определению в них оператора вызова функции () ведут себя в некотором смысле как функции. Объекты-функции, как стандартные, так и определяемые пользователем, позволяют записывать код с использованием нетривиальных операций в качестве параметра.

> **Пример 101**

Получив представление о функциях как о таких частях императивной программы, которым предписывается выполнять определенные действия над данными, в общем случае определяемыми параметрами, вернемся к уже известной проблеме, связанной с подсчетом количества повторяющихся цифр в заданном четырехзначном целом числе. Очевидно, что первым кандидатом на роль функции с параметрами должен стать именованный алгоритм вывода количества повторяющихся цифр.

Представим программу, где для операции вывода количества повторяющихся цифр будет использоваться функция ***print(),*** параметрами которой станут сама цифра ***digit*** и ее счетчик ***digitCounter.***

```
// C++ Функции
#include <iostream>
using namespace std;
void print(char digit, short int digitCounter)
{
  cout << "Цифра " << digit
 << " повторяется " << digitCounter << " pasa\n";
}

int main()
{
  int n;
static short int digitOCounter;
static short int digitlCounter;
static short int digit2Counter;
static short int digit3Counter;
static short int digit4Counter;
static short int digit5Counter;
static short int digit6Counter;
static short int digit7Counter;
static short int digit8Counter;
static short int digit9Counter;
do
 {
      cout << "Натуральное число [1000,9999]? ";
      cin >> n;
      if (cin.fail())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
  }
while (!(n >= 1000 && n <= 9999));
```

```
for (; n != 0; n /= 10)
      switch (n % 10)
      {
      case 0 : ++digitOCounter;
            break;
      case 1 : ++digitlCounter;
            break;
      case 2 : ++digit2Counter;
            break;
      case 3 : ++digit3Counter;
            break;
      case 4 : ++digit4Counter;
            break;
      case 5 : ++digit5Counter;
            break;
      case 6 : ++digit6Counter;
            break;
      case 7 : ++digit7Counter;
            break;
      case 8 : ++digit8Counter;
            break;
      case 9 : ++digit9Counter;
      }
  if   (digitOCounter  >  1) print('0', digitOCounter);
  if   (digitlCounter  >  1) print('1',  digitlCounter);
  if   (digit2Counter  >  1) print('2',  digit2Counter);
  if   (digit3Counter  >  1) print('3',  digit3Counter);
  if   (digit4Counter  >  1) print('4',  digit4Counter);
  if   (digit5Counter  >  1) print('5',  digit5Counter);
  if   (digit6Counter  >  1) print('6',  digit6Counter);
  if   (digit7Counter  >  1) print('7',  digit7Counter);
  if   (digit8Counter  >  1) print('8',  digit8Counter);
  if   (digit9Counter  >  1) print('9',  digit9Counter);
return 0;
}
```

Результат работы программы:

```
Натуральное число [1000,9999]? 5775
Цифра 5 повторяется 2 раза
Цифра 7 повторяется 2 раза
```

Очевидно также, что в процессе декомпозиции алгоритма программы здесь можно отобрать и других кандидатов на роль подпрограммы в виде функции. Напомним, что проектирование императивной программы и ее составных частей основывается на применении методов нисходящего проектирования программ, т.е. на реализации принципа “от сложного — к простому”.

> **Пример 102**

Завершая обзор функций, обратимся к известной проблеме порождения конфликта имен, если в одной и той же области видимости “неосмотрительно” обращаются к именам из пространства имен ***std.*** Подобного рода конфликт, например, может быть вызван “пристрастием” пользователя к имени функции ***abs()*** для вычисления модуля числа, если “забыть” о существовании ***stdv.abs().*** Для разрешения такого конфликта следует объявить собственное пространство имен, а затем обратиться к механизму ***using-объявлений,*** чтобы ввести указанное имя в другую область видимости.

Представим программу, где в область видимости функции ***main()*** к именам из пространства имен ***std*** с помощью ***using-объявления*** будет добавлено имя функции ***abs()*** из пространства имен ***absValue***:

```
// C++ Функции
#include <iostream>
namespace AbsValue
{
double abs(double x)
      {
      printf("double\t");
      return x < 0 ? -x : x;
      }
}

int main()
{
using namespace std;
using AbsValue::abs;
cout << abs(10.0) << endl;
return 0;
}
```

Результат работы программы:

```
double 10
```

Такого же результата можно добиться, если воспользоваться квалифицированным именем функции ***abs(),*** например:

```
int main ()
{
using namespace std;
cout << AbsValue::abs(10.0) << endl;
return 0;
}
```

Массивы

Одномерные массивы

> Пример 103

Массивы представляют собой такие структуры данных, которые состоят из упорядоченного набора фиксированного количества однотипных компонентов, причем индивидуальное имя получает только весь набор, а для компонентов этого набора определяется лишь порядок следования и общее их количество. Каждый элемент массива, представляющий собой минимальную структурную единицу, имеет однозначно определенное множество индексов, которые определяют его положение в упорядоченном наборе, т.е. задают правило вычисления его номера. Отметим, что в C++ элементы массивов индексируются, начиная с 0.

Каждый элемент массива может быть явно обозначен посредством указания имени массива и множества индексов. Доступ к элементу массива в C++ возможен двумя способами: либо с помощью индексной арифметики, либо с помощью адресной арифметики в соответствии с соглашением, что имя массива является указателем на его первый элемент. Необходимо только помнить, что индексную арифметику для элементов встроенных массивов компилятор всегда заменит адресной.

Размерность массива определяется числом его индексов. Так, одномерный массив (или вектор) представляет собой список элементов, обозначаемых одиночными индексами. Для заданного типа ***имя типа*** выражение ***имя_типа***[] — это встроенный тип “одномерный массив”.

Поначалу обратимся к программе, где будут объявлены одномерные символьные массивы, для инициализации которых будут использоваться строковые литералы:

```
// C++ Одномерные массивы
#include <iostream>
#include <iomanip>
int main O
{
using namespace std;
int i;
char line[] = "Привет!";
int n = (sizeof line) / sizeof(char);
cout << sizeof line << '\t' << n << endl;
cout << line << '\t';
while (line[i]) cout << line[i++];
cout << endl;
for (i = 0; i < n — 1; ++i) cout << setw(4) << line[i];
cout << endl;
for (i = 0; i < n; ++i)
      if ((int)line[i] < 0)
            cout << setw(4) << ((int)line[i] + 256);
```

```
      else
       cout << setw(4) << (int)line[i];
       cout << endl;
char string[15] = "Ты вся из огня";
cout << sizeof string << endl;
cout << string << '\t';
i = 0;
while (string[i]) cout << string[i++];
cout << endl;
return 0;
}
```

Результат работы программы:

```
8           8
Привет!     Привет!
П     р     и     в     е     т     !
143   224   168   162   165   226   33    0
15
Ты вся из огня Ты вся  из огня
```

Если массив объявляется без указания размера, наличие инициализатора является обязательным. Как видим, с помощью оператора ***sizeof*** можно выяснить размер блока памяти, выделенной под массив. Напомним, что в C++ размеры объектов или типов выражаются в единицах размера ***char.***

Если массив объявляется с указанием размера, наличие инициализатора является необязательным, однако его наличие — это еще не признак “благополучия”, так как его размер не должен превышать размера массива, в противном случае имеет место ошибка времени компиляции. Если размер инициализатора окажется меньше размера массива, последние его элементы инициализируются по умолчанию значением 0 соответствующего типа, что рано или поздно может привести к побочным эффектам. Именно поэтому при объявлении массива размер инициализатора всегда должен соответствовать размеру массива.

Чтобы вывести в десятичном виде ASCII-коды символьных литералов, как видим, понадобилась операция смещения для порядкового номера символа, принадлежащего в таблице ASCII набору символов кириллицы.

> **Пример 104**

Теперь перейдем к программе, где при объявлении одномерных символьных массивов будут использоваться списки инициализации в стиле С, элементами которых являются символьные литералы:

```
// C++ Одномерные массивы
#include <iostream>
int main ()
{
using namespace std;
 int i;
 char line[] ={ 'П','р','и','в','е','т','\0'};
 cout << sizeof line << endl;
 cout << line << '\t';
 while (line[i]) cout << line[i++];
 cout << endl;
 const int n = 8;
 char string[n] = { 'П','р','и','в','е','т','\0'};
 cout << sizeof string << endl;
 cout << string << '\t';
 i  = 0;
while (string[i]) cout << string[i++];
cout << endl;
for (i = 0; i < n – 1; ++i) cout << string[i];
cout << endl;
return 0;
}
```

Результат работы программы:

```
8
Привет! Привет!
8
Привет! Привет!
Привет!
```

Как видим, если для инициализации символьных массивов используются списки инициализации, не следует забывать, что последним элементом списка должен быть нулевой символ — символьный литерал ***'\0'.*** Если массив объявляется без указания размера, то его фактический размер определяется количеством элементов списка инициализации. В противном случае размер массива указывается либо только с помощью константного выражения, если массив размещается в статической или автоматической памяти, либо, как об этом будет сказано далее, еще и с помощью неконстантного выражения, если массив размещается в свободной памяти.

> **Пример 105**

Объявляя статический массив объектов встроенного типа, в C++ всегда можно положиться на механизм инициализации его элементов по умолчанию значением 0 соответствующего типа. Если это по какой-либо причине неприемлемо, “настоящую” инициализацию либо выполняют при объявлении массива, либо откладывают ее на потом. Зачастую динамическая инициализация элементов массива предпочтительней статической инициализации.

Представим теперь программу, где будет объявлен статический одномерный целочисленный массив, для доступа к элементам которого наряду с “традиционной” индексной арифметикой будет использоваться и адресная арифметика:

```
// C++ Одномерные массивы
#include <iostream>
#include <iomanip>
int main()
{
 using namespace std;
 int i, n;
 static int a [5];
 n = (sizeof a) / sizeof(int);
 cout << sizeof a << '\t' << n << endl;
 cout << a << '\t' << &a[0] << endl;
 for (i = 0; i < n; ++i) cout << setw(2) << a[i];
 cout << endl;
 for (i = 0; i < n; ++i) a[i] = n — 1 — i;
 i = 0;
 while (a[i]) cout << setw(2) << a[i++];
 cout << endl;
 i = 0;
 while (*(a + i)) cout << setw(2) << *(a + i++);
 cout << endl;
 i = 0;
 while (i[a]) cout << setw(2) << i++[a];
 cout << endl;
 i = 0;
 while (*(i + a)) cout << setw(2) << *(i++ + a);
 cout << endl;
 int* p = &a[4];
 cout << p[0] << setw(2) << *p
 <<         setw(3)  << p[-1]  << setw(2)   << * (p - 1)
 <<         setw(3)  << p[-2]  << setw(2)   << * (p - 2)
 <<         setw(3)  << p[-3]  << setw(2)   << * (p - 3)
 <<         setw(3)  << p[-4]  << setw (2)  << * (p - 4) << endl;
return 0;
}
```

Результат работы программы:

```
20        5
0x437020            0x437020
0 0 0 0 0
4  3  2  1
4  3  2  1
4  3  2  1
4  3  2  1
0         0    1     1        2        2      3      3      4      4
```

Напомним здесь, что обращение к имени массива ***a*** влечет за собой тривиальное преобразование имени массива в указатель на ***int,*** значением которого является адрес первого элемента массива — ***&a[0].*** Напомним также, что адресная арифметика — это арифметика с указателями. Операция индексирования ***i***-го элемента массива ***a*** влечет за собой операцию разыменования указателя, значение которого теперь связано с адресом ***i***-го элемента массива благодаря выполнению операции сложения ***&a\0***] и смещения ***i*sizeof(int).** Иными словами, выражение ***a[i]*** в индексной арифметике эквивалентно выражению ****(a+i)*** в адресной арифметике.

Если в адресной арифметике не так важно, в каком порядке записывать операнды для операции сложения, то в индексной арифметике этот порядок "важен" только с позиции здравого смысла. Причина в том, что недоумение по поводу непривычного порядка операндов для оператора индексации [] может быть вызвано непониманием природы перехода от ***имя_массива[выражение***] к ****(имя_массива+выражение),*** выполняемого компилятором автоматически для встроенных массивов.

Как в C, так и в C++ во время выполнения программы можно выйти за границы диапазона встроенного массива без генерации сообщений об ошибках времени выполнения. Устранение такого рода проблемы возможно, но только не в рамках императивной парадигмы программирования, поскольку для создания защищенного массива потребуется поддержка со стороны объектно-ориентированной парадигмы в виде класса и перегруженного оператора индексации [].

> **Пример 106**

C++ унаследовал от C так называемые C-строки — символьные массивы, последним элементом которых является нулевой символ — символьный литерал ***'\0'.*** Нулевой символ играет роль ограничителя, позволяющего указать на конец C-строки. Такой подход является следствием той причины, которую связывают с утратой информации о размере символьного массива благодаря неявному преобразованию типа из ***char[]*** в ***char*,*** если массив передается в качестве аргумента функциям. Тем самым отпадает необходимость в передаче этим функциям еще одного дополнительного аргумента, указывающего на размер передаваемой C-строки. Так, в стандартных заголовочных файлах <***cstring***> и <***string.h***> представлены объявления подобного рода функций, аргументами которых являются указатели на ***char,*** например, ***strlen()*** для вычисления длины и ***strcpy()*** для копирования C-строк.

Представим программу, где для операции ввода C-строки будет использоваться функция-член ***getline()*** класса ***basic_istream,*** а для операций вычисления длины и копирования C-строки наряду с обычными средствами, в основе которых лежит адресная арифметика, будут использоваться также и функции ***strlen()*** и ***strcpy()***:

```
// C++ Одномерные массивы
#include <iostream>
#include <iomanip>
#include <cstring>
int main()
{
using namespace std;
char string[20];
char* stringPointer = string;
cout << "? ";
repeat : cin.getline(string, 20);
if (cin.fail())
      {
      cin.clear();
      goto repeat;
      }
else
      cout << sizeof string << '\t' << string << endl;
      for (int i = 0;; ++i)
      {
      if ((int)string[i] < 0)
            cout << setw(4) << ((int)string[i] + 256);
      else
            cout << setw(4) << (int)string[i];
            if ((int)string[i] = 0) break;
      }
cout << endl;
int length = 0;
```

```
while (*stringPointer++) ++length;
cout << length << '\t' << strlen(string) << endl;
stringPointer = string;
char newString[20];
char* newStringPointer = newString;
do
  *newStringPointer = *stringPointer++;
while (*newStringPointer++);
cout << sizeof newString << '\t' << newString << endl;
cout << "? ";
yet : cin.getline(string, 20);
if (cin.fail())
   {
   cin.clear();
   goto yet;
   }
else
      cout << strcpy(newString, string) << '\t' << newString << endl;
return 0;
}
```

Результат работы программы:

```
? Привет!
20       Привет!
143 224 168 162 165 226 33 0
7        7
20       Привет!
? Ты вся из огня
Ты вся из огня Ты вся из огня
```

Функция ***basic_istream::getline()*** предназначена для таких операций ввода, где заранее не предполагается, что означают введенные символы. Символы-разделители ***getline()*** считывает так же, как и обычные символы, а когда в потоке ввода встречает символ-ограничитель, она считывает его и затем удаляет из потока. Второй аргумент функции ***getline()*** определяет количество считываемых символов из потока.

Как видим, здесь реализован еще один вариант механизма защиты операции ввода, где уже нет необходимости извещать об ошибке формата, так как символьный массив может быть использован также и для считывания из потока строки с “мусором”.

Цикл с постусловием для операции копирования С-строки может оказаться и с пустым телом:

```
   do
   ;
   while (*newStringPointer++ = *stringPointer++);
```

> **Пример 107**

Как и любые другие переменные, указатели могут стать элементами массивов. Инициализация указателей может быть только явной, ее выполняют либо при объявлении массива указателей, либо откладывают на потом, главное — не забыть ее выполнить: адресная арифметика ошибок не “прощает”. Напомним, что значение указателя, равное 0, означает, что указатель ни на что не ссылается, поэтому попытка его разыменования приводит к аварийному завершению программы.

Обратимся к программе, где будет объявлен одномерный массив указателей на ***int*** для иллюстрации механизма косвенного доступа к переменным ***a*** и ***b,*** если их адреса станут инициализаторами указателей:

```
// C++ Одномерные массивы
#include <iostream>
int: main ()
{
 using namespace std;
 const int n = 2;
 int a(5), b(7);
 int* pointersOnlntArraylD[n];
 cout << pointersOnlntArraylD << endl;
 pointersOnlntArraylD[0] = &a;
 pointersOnlntArraylD[1] = &b;
 cout << pointersOnlntArraylD[0] << '\t'
      << *pointersOnIntArraylD << ' \t'
      << *pointersOnIntArraylD[0] << '\t'
      <<**pointersOnIntArraylD << endl;
cout << pointersOnlntArraylD[1] << '\t'
      << *(pointersOnlntArraylD + 1) << '\t*
     << *pointersOnIntArraylD[1] << '\t'
      << ** (pointersOnlntArraylD + 1) << endl;
return 0;
}
```

Результат работы программы:

```
0x76fdc8
0x76fde8            0x7 6fde8          5         5
0x76fde4            0x76fde4           7         7
```

> **Пример 108**

Объявляя динамический массив объектов встроенного типа, для доступа к его элементам наряду с традиционным указателем, связанным с блоком выделенной памяти, можно воспользоваться и указателем на указатель.

Представим теперь программу, где будет объявлен динамический одномерный целочисленный массив, для доступа к элементам которого будут использоваться указатель на ***int*** и указатель на указатель на ***int.***

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
int main{)
{
 const int n = 2;
 int* pointerOnlnt = new int[n];
 int** pointerOnPointerOnlnt = SpointerOnlnt;
 pointerOnlnt[0] =5;
 pointerOnlnt[1] = 7;
 cout << pointerOnlnt << endl;
 cout << fipointerOnlnt[0] << '\t' << pointerOnlnt[0] << '\t'
      << *pointerOnInt << endl;
 cout << SpointerOnlnt[1] << '\t' << pointerOnlnt[1] << '\t'
      << *(pointerOnlnt +1) << endl;
 cout << pointerOnPointerOnlnt << endl;
 cout << *pointerOnPointerOnInt << '\t'
      << **pointerOnPointerOnInt << endl;
 cout << *pointerOnPointerOnInt + 1 << '\t'
      << *(*pointerOnPointerOnInt + 1) << endl;
delete[] pointerOnlnt;
return 0;
}
```

Результат работы программы:

```
0x8905a0
0x8905a0            5         5
0x8905a4            7         7
0x7 6fdf4
0x8905a0            5
0x8905a4            7
```

Напомним, что для удаления динамических массивов следует воспользоваться оператором ***delete*[]**.

> **Пример 109**

От автоматического массива указателей перейдем к динамическому и представим программу, где будет объявлен динамический одномерный массив указателей на ***int*** для иллюстрации механизма косвенного доступа к переменным ***a*** и ***b,*** если их адреса, как и ранее, станут инициализаторами указателей:

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
int main ()
{
const int n = 2;
int a(5), b(7);
int** pointerOnPointerOnlnt = new int*[n];
cout << pointerOnPointerOnlnt << endl;
pointerOnPointerOnlnt[0] = &a;
pointerOnPointerOnlnt[1] = &b;
cout << pointerOnPointerOnlnt[0] << '\t'
     << *pointerOnPointerOnInt << '\t'
     << *pointerOnPointerOnInt[0] << '\t'
      <<**pointerOnPointerOnInt << endl;
cout << pointerOnPointerOnlnt[1] << '\t'
     << *(pointerOnPointerOnlnt + 1) << '\t'
     << *pointerOnPointerOnInt[1] << '\t'
      << **(pointerOnPointerOnlnt + 1) << endl;
delete[] pointerOnPointerOnlnt;
return 0;
}
```

Результат работы программы:

```
0x8905a0
0x7 6fdf0              0x76fdf0          5          5
0x76fdec               0x76fdec          7          7
```

> **Пример 110**

Когда имя массива передается функции, она получает в качестве аргумента адрес первого элемента массива. Иными словами, массив нельзя передать по значению. Напомним, что благодаря тривиальному преобразованию имени массива в указатель теряется информация о размере массива. Чтобы добиться точного соответствия типов формальных и фактических аргументов при передаче одномерных массивов, аргумент вызываемой функции может быть объявлен одним из трех способов: либо как массив с указанием его размера, либо как массив без указания его размера, либо как указатель. Очевидно, что для последних двух случаев понадобится еще один дополнительный аргумент, связанный с размером массива.

Поначалу представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** аргумент которой будет объявлен как одномерный целочисленный массив с указанием его размера:

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
const int n = 3;
void print(int array[n])
{
for (int i = 0; i < n; ++i) cout << array[i] << endl;
}

int main()
{
int a[n] ={1,2,3};
print(a);
return 0;
}
```

Результат работы программы:

```
1
2
3
```

> **Пример 111**

Теперь представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как одномерный целочисленный массив без указания его размера:

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
void print(int array[], int n)
{
for (int i = 0; i < n; ++i) cout << array[i] << endl;
}

int main()
{
  const int n = 3;
  int a[n] ={1,2,3};
  print(a, 1);
  cout << endl;
  print(a, 2);
  cout << endl;
  print(a, n);
  return 0;
}
```

Результат работы программы:

```
1

1
2

1
2
3
```

> **Пример 112**

В завершение обзора указанных способов объявления аргумента при передаче массивов представим программу, где для визуализации элементов одномерного целочисленного массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как указатель на ***int:***

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
void print(int* pointer, int n)
{
for (int i = 0; i < n; ++i) cout << pointer[i] << endl;
}

int main()
{
const int n = 3;
int a[n] ={1,2,3};
print(a, 1);
cout << endl;
print(a, 2);
cout << endl;
print(a, n);
cout << endl;
print(&a[1], 2);
return 0;
}
```

Результат работы программы:

```
1

1
2

1
2
3

2
3
```

> **Пример 113**

С помощью директивы ***typedef,*** доставшейся C++ в наследство от C, можно определить такой тип, как “одномерный массив заданного размера из элементов указанного типа”. По-разному можно относиться к такого рода синонимам сложных типов, бесспорно одно — ясность кода от этого только улучшается.

Продолжая иллюстрацию механизма передачи массивов функциям, обратимся к программе, где для визуализации элементов одномерного целочисленного массива будет использоваться функция ***print(),*** тип аргумента которой объявлен с помощью директивы ***typedef*** как “одномерный массив из трех элементов типа ***int:***

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
const int n = 3;
typedef int ArraylD[n];
void print(ArraylD array)
{
for (int i = 0; i < n; ++i) cout << array[i] << endl;
}

int main()
{
ArraylD a = { 1, 2, 3 };
print(a);
return 0;
}
```

Результат работы программы:

```
1
2
3
```

Такой подход, как видим, позволяет избавиться и от “лишнего” дополнительного аргумента, связанного с размером массива.

> **Пример 114**

Оглядываясь назад, от автоматического одномерного целочисленного массива теперь перейдем к динамическому и представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как указатель на ***int:***

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
void print(int* pointer, int n)
{
for (int i = 0; i < n; ++i) cout << pointer[i] << endl;
}

int main()
{
  int n = 3;
  int* p = new int[n];
  for (int i = 0; i < n; p[i] = ++i);
  print(p, 1);
  cout << endl;
  print(p, 2);
  cout << endl;
  print (p, n);
  cout << endl;
  print(&p[1], 2);
  return 0;
}
```

Результат работы программы:

```
1

1
2

1
2
3

2
3
```

> **Пример 115**

Для выделения массивам свободной памяти можно воспользоваться функциями ***malloc()*** и ***calloc()*** стандартной библиотеки C, а для ее освобождения — функцией ***free()***. В случае успешного выделения свободной памяти функции ***malloc()*** и ***calloc()*** возвращают ненулевой указатель на ***void,*** в противном случае — 0. Функция ***malloc()*** выделяемую память не инициализирует, а вот ***calloc(),*** напротив, инициализирует ее значением 0 соответствующего типа.

Обратимся к программе, где для выделения свободной памяти одномерному целочисленному массиву будет использоваться функция ***malloc(),*** а для визуализации его элементов будет использоваться уже знакомая функция ***print(),*** первый аргумент которой объявлен как указатель на ***int***:

```
// C++ Одномерные массивы
#include <iostream>
#include <cstdlib>
using namespace std;
void print(int* pointer, int n)
{
for (int i = 0; i < n; ++i) cout << pointer[i] << endl;
}
int main()
{
int n = 3 ;
int* p = (int*)malloc(n*sizeof(int));
for (int i = 0; i < n; p[i] = ++i);
print(p, n);
free(p);
return 0;
}
```

Результат работы программы:

```
1
2
3
```

Представим прототипы функций ***malloc()*** и ***free()*** из стандартных заголовочных файлов <***cstdlib***> и <***stdlib.h***>***:***

void* malloc(size_t);

void free(void*);

Как видим, функция ***malloc()*** выделяет блок памяти размером ***n*sizeof (int)*** байт.

> **Пример 116**

Теперь обратимся к программе, где для выделения свободной памяти одномерному целочисленному массиву будет использоваться функция ***calloc()***, а для визуализации его элементов будет использоваться та же функция ***print(),*** первый аргумент которой объявлен как указатель на ***int.***

```
// C++ Одномерные массивы
#include <iostream>
#include <cstdlib>
using namespace std;
void print(int* pointer, int n)
{
for (int i = 0; i < n; ++i) cout << pointer[i] << endl;
}
int main()
{
  int n = 3;
  int* p = (int*)calloc(n, sizeof(int));
  cout  <<  &p[0]  << '\t'  <<p[0]  <<  endl;
  cout  <<  &p[1]  << '\t'  <<p[1]  <<  endl;
  cout  <<  &p[2]  << '\t'  <<p[2]  <<  endl;
for (int i = 0; i < n; p[i] = ++i);
print (p, n);
free(p);
return 0;
}
```

Результат работы программы:

```
0x8905a0    0
0x8905a4    0
0x8905a8    0
1
2
3
```

Представим прототип функции ***calloc()*** из стандартных заголовочных файлов <***cstdlib***> и <***stdlib.h***>:

void* calloc{size_t, sizet);

Как видим, функция ***calloc()*** выделяет **n** блоков памяти, размер каждого из которых ***sizeof(int)*** байт, и инициализирует их значением 0 типа ***int.***

```
0x8905a4    0
0x8905a8    0
```

> **Пример 117**

Одной из ярких иллюстраций передачи массивов функциям являются алгоритмы сортировки, позволяющие упорядочить множество данных в возрастающем или убывающем порядке. Одним из самых известных (и самых скверных) алгоритмов сортировки является метод “пузырьковой” сортировки, который иногда называют “злым духом перестановок”.

Обратимся к программе, позволяющей упорядочить в возрастающем порядке одномерный целочисленный массив методом “пузырьковой” сортировки:

```
// C++ Одномерные массивы
#include <iostream>
#include <iomanip>
void bubble(int* pointer, int counter)
{
register int i, j;
int element;
for (i = 0; i < counter – 1; ++i)
for (j = i + 1; j < counter; ++j)
      if (pointer[j] < pointer[i])
      {
      element = pointer[i];
      pointer[ij = pointer[j];
      pointer[j] = element;
      }
}

int main()
{
using namespace std;
const int n = 10;
int a[n] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
int i;
for (i = 0; i < n; ++i) cout << setw(2) << a[i];
cout << endl;
bubble(a, n);
for (i = 0; i < n; ++i) cout << setw(2) << a[i];
cout << endl;
return 0;
}
```

Результат работы программы:

```
9876543210
0123456789
```

> **Пример 118**

Объявив с помощью директивы ***typedef*** тип “комплексное число” как “одномерный массив из двух элементов типа ***double*** для представления вещественной и мнимой частей комплексного числа, можно реализовать комплексную арифметику даже в рамках императивной парадигмы программирования. Обычно для такой арифметики требуется поддержка со стороны объектно-ориентированной парадигмы в виде абстрактного типа данных — класса ***complex.*** Ограничимся минимальным набором операций для комплексной арифметики: например, для инициализации — это функция ***define()*** для визуализации — это функция ***print()*** а для доступа к вещественной и мнимой частям комплексного числа — это функции ***get_re()*** и ***get_im().***

Продолжая иллюстрацию механизма передачи массивов функциям, обратимся к программе, где будет реализована комплексная арифметика для операции сложения:

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
typedef double Complex[2];
void define(Complex c, double r = 0.0, double i = 0.0)
{
c[0] = r;
c[1] = i;
}

void print(Complex c)
{
cout << '(' << c[0] << ", " << c[1] << ')' << endl;
}

double& get_re(Complex c)
{
return c[0];
}

doubles get_im(Complex c)
{
return c[1];
}

int main()
{
Complex a, b;
define(a, -1, 5);
define(b);
```

```
print(a);
print(b);
get_re(b) += 10;
get_im(b) +=7;
print(b);
get_re(a) += get_re(b);
get_im(a) += get_im(b);
print(a);
return 0;
}
```

Результат работы программы:

```
(-1, 5)
(0, 0)
(10, 7)
(9, 12)
```

Как видим, определения синонимов сложных типов с помощью директивы ***typedef*** не только приводят к улучшению ясности интерфейса всех функций из указанного набора, “маскируя” передачу им одномерного массива заданного размера, но также и “приподнимают” их до уровня, свойственного объектно-ориентированной парадигме программирования.

Заметим, что в смешанной арифметике, как правило, полагаются на стандартные преобразования типов, поэтому вместо чисел с плавающей точкой можно указывать и целые числа. Чтобы выполнить операцию сложения числа с вещественной или мнимой частями комплексного числа, равно как и операцию сложения вещественных и мнимых частей двух комплексных чисел, функции ***get_re()*** и ***get_im()*** должны возвращать ссылку на ***double.***

> Пример 119

Объявив с помощью директивы ***typedef*** тип “комплексное число”, теперь можно объявлять и такие структуры данных, как одномерные массивы комплексных чисел. Отметим, что такого рода структуры данных принято называть массивами массивов.

Обратимся к программе, позволяющей отыскивать заданное комплексное число в одномерном массиве комплексных чисел:

```
// C++ Одномерные массивы
#include <iostream>
using namespace std;
typedef double Complex[2];
bool input(Complex c)
{
cout << "(re,im)" << '\t';
cout << "re? ";
cin >> c[0];
if (cin.fail())
   {
   cout << "Ошибка формата!\n";
   return false;
   }
cout << '\t' << "im? ";
cin >> c[1];
if (cin.fail())
      {
      cout << "Ошибка формата!\n";
      return false;
      }
return true;
}

void print(Complex c)
{
cout << '(' << c[0] << ", " << c[1] << ')' << endl;
}

Complex* find(Complexfi c, Complex key)
{
if (c[0] = key[0] && c[1] = key[1])
   return &c;
else
   return (Complex*)0;
}

int main()
```

```
{
const int n = 3; int i;
Complex a[n], key;
Complex* p;
for (i = 0; i < n; ++i)
   if (!input(a[i])) return -1;
for (i = 0; i < n; ++i) print(a[i]);
cout << "Ключ поиска" << endl;
if (!input(key)) return -1;
for (i = 0; i < n; ++i)
   if (p = find(a[i], key)) print(*p);
return 0;
}
```

Результат работы программы:

```
(re,im) re? -1 im? 5
(re,im) re? 10 im? 7
(re,im) re? 9 im? 12
(-1, 5)
(10, 7)
(9, 12)
Ключ поиска
(re,im) re? 10 im? 7
(10, 7)
```

Как правило, результат операции поиска указанного объекта в массиве объектов возвращается в виде указателя на объект.

А вот такое определение функции ***find()***

```
bool find(Complex c, Complex key)
{
if (c[0] = key[0] && c[1] == key[1])
   return true;
else
   return false;
}
```

приводит лишь к фиксации состояния “нашел — не нашел”:

```
for (i = 0; x < n; ++i)
if (find(a[i], key)) print(a[i]);
```

> Пример 120

В завершение обзора одномерных массивов вернемся к уже известной проблеме, связанной с подсчетом количества повторяющихся цифр в заданном четырехзначном целом числе. Памятуя о неоправданно больших накладных расходах, от счетчиков для каждой цифры перейдем теперь к массиву счетчиков.

Представим программу, где для счетчиков цифр будет объявлен статический одномерный целочисленный массив:

```
// C++ Одномерные массивы
#include <iostream>
int main()
{
using namespace std;
int n;
static int digitsCounters[10];
do
 {
      cout << "Натуральное число [1000,9999]? ";
      cin >> n;
      if (cin.fail())
      {
      cout << "Ошибка формата!\n";
      return -1;
      }
}
while (!(n >= 1000 && n <= 9999));
for (; n != 0; digitsCounters[n % 10]++, n /= 10);
for (int i = 0; i < 10; ++i)
if (digitsCounters[i] > 1) cout << "Цифра "
 << i
 << " повторяется "
 << digitsCounters[i]
 << " pasa\n";
return 0;
}
```

Результат работы программы:

```
Натуральное число [1000,9999]? 5775
Цифра 5 повторяется 2 раза
Цифра 7 повторяется 2 раза
```

Как видим, вся процедура подсчета повторяющихся цифр может быть “упрятана” в заголовок инструкции-итерации ***for.***

Двумерные массивы

> Пример 121

Двумерный массив (или матрица) представляет собой таблицу элементов, которая состоит из фиксированного числа строк и столбцов. Каждый элемент матрицы обозначается двумя индексами. Первый индекс указывает строку, а второй — столбец матрицы. Для заданного типа ***имя_типа*** выражение ***имя_типа[\][\]*** — это встроенный тип "двумерный массив". Напомним, что в C++ многомерные массивы трактуются как массивы массивов. Двумерный массив — это массив одномерных массивов или, как говорят, список векторов.

Поначалу обратимся к программе, где при объявлении двумерного целочисленного массива будут использоваться списки инициализации в стиле C, а для доступа к его элементам наряду с "традиционной" индексной арифметикой будет использоваться и адресная арифметика:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
int main()
{
using namespace std;
const int columnSize(2), rowSize(3);
int a[columnSize][rowSize] ={ {1,2,3}, {4,5, 6} };
int i, j , n;
n = (sizeof a) / sizeof(int);
cout  <<   sizeof a << '\t' << n << endl;
cout  <<   a << '\t' << &a[0][0] << '\t' << a[0][0] << endl;
cout  <<   a + 1 << '\t' << &a[1][0] << '\t' << a[1][0]  <<     endl;
for (i = 0; i < columnSize; ++i)
      {
for (j = 0; j < rowSize; ++j) cout << setw(2) << a[i][j]; cout <<
endl;
      }
cout  <<   a[0][0] << ' \t' << **a << endl;
cout  <<   a[1][2] << '\t' << *(*(a +1) +2) << endl;
int* p = &a[0][0];
for (i = 0; i < columnSize; ++i)
      {
      for (j = 0; j < rowSize; ++j)
      cout << setw(2) << p[i*rowSize+j]; cout << endl;
      }
return 0;
}
```

Результат работы программы:

```
24          6
0x76fdc0          0x76fdc0    1
0x76fdcc          0x76fdcc    4
1 2 3
4 5 6
1      1
6      6
1 2 3
4 5 6
```

Как видим, список инициализации двумерного массива в стиле С представляет собой список списков инициализации одномерных массивов.

Напомним здесь, что обращение к имени массива ***a*** влечет за собой тривиальное преобразование имени массива в указатель на ***int,*** значением которого является адрес первого элемента массива — ***&a[0][0].*** Операция индексирования элемента ***i***-ой строки и ***j***-го столбца массива ***a*** влечет за собой операцию разыменования указателя, значение которого теперь связано с адресом элемента ***i***-ой строки и ***j***-го столбца благодаря выполнению операции сложения ***&a[0][0]*** и смещения (***i*3+j)*sizeof(int***). Если "знать", как вычисляется смещение, становится понятно, почему многомерные массивы в С++ следует трактовать как массивы массивов.

Объявление ***int a[][rowSize]*** эквивалентно объявлению ***int***(**a)[rowSize],* которое означает "указатель на массив из ***rowSize*** элементов типа ***int,*** поэтому, например, выражение &a[1] эквивалентно выражению ***(a+1),*** а выражение ***a[1][0]*** в индексной арифметике эквивалентно выражению ****(a+1)** в адресной арифметике.

Итак, выражение a [i][j] в индексной арифметике эквивалентно выражению ***(*(a+i)+j)*** в адресной арифметике.

> Пример 122

Представим программу, где с помощью динамического одномерного массива указателей на ***int*** и динамических одномерных целочисленных массивов будет создан динамический двумерный целочисленный массив, для доступа к элементам которого будет использоваться указатель на указатель на ***int***:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
int main()
{
using namespace std;
int i, j, m(2), n(3), value(0);
int** pointerOnPointerOnlnt = new int*[m];
for (i = 0; i < m; ++i) pointerOnPointerOnlnt[i] = new int[n];
for (i = 0; i < m; ++i)
for (j = 0; j < n; ++j) pointerOnPointerOnlnt[i][j] = ++value; cout <<
pointerOnPointerOnlnt[0] << '\t'
       << SpointerOnPointerOnlnt[0][0] << '\t'
       << *pointerOnPointerOnInt[0] << '\t'
       << pointerOnPointerOnlnt[0][0] << endl;
cout << pointerOnPointerOnlnt[1] << '\t'
       << SpointerOnPointerOnlnt[1][0] << '\t'
       << *pointerOnPointerOnInt[1] << '\t'
       << pointerOnPointerOnlnt[1][0] << endl;
for (i = 0; i < m; ++i)
      {
for (j = 0; j < n; ++j)
    cout << setw(2) << pointerOnPointerOnlnt[i][j];
    cout << endl;
    }
cout << **pointerOnPointerOnInt << '\t'
       << *(*(pointerOnPointerOnlnt +1) +2) << endl;
for (i = 0; i < m; ++i) delete[] pointerOnPointerOnlnt[i];
delete[] pointerOnPointerOnlnt;
return 0;
}
```

Результат работы программы:

```
0x8905b0            0x8905b0          1        1
0x8905c0            0x8905c0          4        4
1 2 3
4 5 6
1     6
```

> Пример 123

От динамических двумерных целочисленных массивов перейдем к вещественным и представим программу, где для доступа к элементам массива будет использоваться “традиционная” индексная арифметика:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
int main ()
{
using namespace std;
const int columnSize(5), rowSize(4);
int i, j;
double** base = new double*[columnSize];
for (i =0; i < columnSize; ++i) base[i] = new double[rowSize];
cout.precision(1);
for (i = 0; i < columnSize; ++i)
      {
      for (j = 0; j < rowSize; ++j)
     {
     base[i] [j] = (i + 1) * (j + 1);
     cout << setw(5) << fixed << base[i][j];
     }
cout << endl;
      }
double maximum = base[0][0];
for (i = 0; i < columnSize; ++i)
for (j = 0; j < rowSize; ++j)
if (base[i] [j] > maximum) maximum = base[i] [j];
cout << "maximum = " << maximum << endl;
for (i =0; i < columnSize; ++i) delete[] base[i];
delete[] base;
return 0;
}
```

Результат работы программы:

```
1.0        2.0      3.0      4.0
2.0        4.0      6.0      8.0
3.0        6.0      9.0      12.0
4.0        8.0     12.0      16.0
5.0        10.0    15.0      20.0
maximum = 20.0
```

> **Пример 124**

Итак, когда имя массива передается функции, она получает в качестве аргумента адрес первого элемента массива. Чтобы добиться точного соответствия типов формальных и фактических аргументов при передаче двумерных массивов, аргумент вызываемой функции может быть объявлен одним из пяти способов: либо как массив с указанием его размера, либо как массив без указания его первой размерности, либо как указатель на одномерный массив, либо как указатель, либо как указатель на указатель. Для второго и третьего случаев понадобится всего один дополнительный аргумент, связанный с первой размерностью массива, а вот для последних двух случаев понадобятся уже два дополнительных аргумента, связанных с размером массива.

Поначалу представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** аргумент которой будет объявлен как двумерный целочисленный массив с указанием его размера:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
const int columnSize(2), rowSize(3);
void print(int array[columnSize][rowSize])
{
   for (int i = 0; i < columnSize; ++i)
   {
for (int j = 0; j < rowSize; ++j) cout << setw(2) << array[i] [j];
cout << endl;
   }
}

int main()
{
int a[columnSize][rowSize] ={ { 1, 2, 3>, {4, 5, 6} };
print(a);
return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

> **Пример 125**

Теперь представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как двумерный целочисленный массив без указания его первой размерности:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
const int rowSize = 3;
void print(int array[][rowSize], int columnSize)
{
  for (int i = 0; i < columnSize; ++i)
  {
for (int j = 0; j < rowSize; ++j) cout << setw(2) << array[i][j];
cout << endl;
  }
}

int main()
{
const int columnSize = 2;
int a[columnSize][rowSize] = { { 1, 2, 3 }, { 4, 5, 6 > };
print(a, columnSize);
return 0;
}
```

Результат работы программы:

1 2 3

4 5 6

> **Пример 126**

Продолжая обзор указанных способов объявления аргумента при передаче массивов, теперь представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как указатель на одномерный массив из трех элементов типа ***int:***

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
const int rowSize =3;
void print(int (*pointerOnArraylD)[rowSize], int columnSize)
{
   for (int i = 0; i < columnSize; ++i)
   {
   for (int j = 0; j < rowSize; ++j)
    cout << setw(2) << pointerOnArraylD[i][j];
    cout << endl;
   }
}

int main()
{
const int columnSize = 2;
int a[columnSize][rowSize] ={ { 1, 2, 3}, {4, 5, 6} };
print(a, columnSize);
return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

Объявив с помощью директивы ***typedef*** тип “указатель на одномерный массив заданного размера из элементов указанного типа”, теперь можно объявлять и такие структуры данных, как двумерные массивы, но об этом уже в следующем примере.

> **Пример 127**

Продолжая обзор, теперь представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** тип первого аргумента которой с помощью директивы ***typedef*** объявлен как “указатель на одномерный массив из трех элементов типа ***int:***

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
const int rowSize =3;
typedef int (*pointerOnArraylD)[rowSize];
void print(pointerOnArraylD pointer, int columnSize)
{
  for (int i = 0; i < columnSize; ++i)
  {
for (int j =0; j < rowSize; ++j) cout << setw(2) << pointer[i][j];
cout << endl;
  }
}

int main()
{
const int columnSize = 2;
int a[columnSize][rowSize] = { { 1, 2, 3 ), { 4, 5, 6 } };
print(a, columnSize);
return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

> **Пример 128**

Продолжая обзор указанных способов объявления аргумента при передаче массивов, представим программу, где для визуализации элементов двумерного целочисленного массива будет использоваться функция ***print()***, первый аргумент которой будет объявлен как указатель на ***int.***

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
void print(int* pointer, int columnSize, int rowSize)
{
   for (int i = 0; i < columnSize; ++i)
   {
   for (int j = 0; j < rowSize; ++j)
    cout << setw(2) << pointer[i*rowSize+j];
    cout << endl;
   }
}

int main()
{
const int columnSize(2), rowSize (3);
int a[columnSize][rowSize] = { { 1, 2, 3 } , { 4, 5, 6 } };
print(&a[0][0], columnSize, rowSize);
return 0;
}
```

Результат работы программы:

1 2 3

4 5 6

> **Пример 129**

Объявляя с помощью директивы ***typedef*** синонимы сложных типов, следует стремиться и к таким их именованиям, которые “говорили” бы сами за себя иными “словами”, чем те громоздкие по своей сути имена, “приставленные служить” своим учебным примерам.

Продолжая обзор, представим программу, где для визуализации элементов массива будет использоваться функция ***print(),*** тип аргумента которой с помощью директивы ***typedef*** объявлен как “двумерный целочисленный массив заданного размера”:

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
const int columnSize(2), rowSize(3);
typedef int Array2D[columnSize][rowSize];
void print(Array2D array)
{
   for (int i = 0; i < columnSize; ++i)
   {
   for (int j = 0; j < rowSize; ++j) cout << setw(2) << array[i][j];
   cout << endl;
   }
}

int main()
{
Array2D a={ {1,2,3}, {4,5, 6 } };
print(a);
return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

> **Пример 130**

В завершение обзора указанных способов объявления аргумента при передаче массивов теперь от автоматических двумерных массивов перейдем к динамическим.

Поначалу обратимся к программе, где для визуализации элементов динамического двумерного целочисленного массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как указатель на указатель на ***int:***

```
// C++ Двумерные массивы
#include <iostream>
#include <iomanip>
using namespace std;
void print(int** pointer, int columnSize, int rowSize)
{
  for (int i = 0; i < columnSize; ++i)
  {
for (int j = 0; j < rowSize; ++j) cout << setw(2) << pointer[i][j];
cout << endl;
  }

}
int main()
{
  int columnSize(2), rowSize(3);
  int i, j, value(0);
  int** p = new int*[columnSize];
  for (i =0; i < columnSize; ++i) p[i]= newint[rowSize];
  for (i = 0; i < columnSize; ++i)
  for (j = 0; j < rowSize; ++j) p[i][j] = ++value;
  print(p, columnSize, rowSize);
  for (i = 0; i < columnSize; ++i) delete[] p[i];
  delete[] p;
  return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

Напомним, что динамический двумерный массив создается на основе массива указателей, каждый из которых связан со своим одномерным массивом, поэтому здесь не может быть и речи о непрерывном блоке памяти, выделенной под массив.

> **Пример 131**

Памятуя о том, что двумерный массив представляет собой массив одномерных массивов, можно воспользоваться библиотечной функцией ***malloc()*** из стандартных заголовочных файлов **<*cstdlib*>** и **<*stdlib.h*>,** чтобы выделить ему непрерывный блок динамической памяти.

Теперь обратимся к программе, где для визуализации элементов “двумерного” целочисленного массива будет использоваться функция ***print(),*** первый аргумент которой будет объявлен как указатель на ***int:***

```
// C++ Двумерные массивы
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;
void print(int* pointer, int columnSize, int rowSize)
{
  for (int i = 0; i < columnSize; ++i)
  {
  for (int j = 0; j < rowSize; ++j)
  cout << setw(2) << pointer[i*rowSize+j];
  cout << endl;
  }
}

int main()
{
  int columnSize(2), rowSize(3), value(0);
  int n = columnSize * rowSize;
  int* p = (int*)malloc(n*sizeof(int));
  for (int i = 0; i < n; ++i) p[i] = ++value;
  print(p, columnSize, rowSize);
  free(p);
  return 0;
}
```

Результат работы программы:

1 2 3
4 5 6

Приложение

Алфавит языка C++

Набор символов или составных символов, рассматриваемых как единое целое, с помощью которых составляется текст программы, принято называть ***алфавитом*** языка программирования. В C++ элементы алфавита строятся из упорядоченного набора символов, каждому из которых соответствует индивидуальный числовой код от 0 до 127 основной таблицы ASCII-кодов персонального компьютера.

Текст программы — это последовательность строк, состоящих из элементов алфавита языка. Текст программы, представляемый в виде текстового файла, является входным текстом или исходным кодом для компилятора языка C++.

К элементам алфавита языка C++ относятся:

- буквы:

прописные латинские буквы
ABCDEFGHIJKLMNOPQRSTUVWXYZ
строчные латинские буквы **abcdefghijklmnopqrstuvwxyz**
символ подчеркивания _

- арабские цифры:

0123456789

- специальные символы:

+ - * / % & ^ | < > = . , ; : '()[]{}#!?\ ~

составные символы, рассматриваемые как единое целое:

::	**->**	**++**	**—**	**.***	**->***	**<<**	**>>**
<=	**>=**	**=**	**!=**	**&&**	**||**	***=**	**/=**
%=	**+=**	**-=**	**<<=**	**>>=**	**&=**	**| =**	**^=**
//	**/***	***/**	**...**	**"**			

Лексическая структура языка C++

Элементы алфавита языка используются для построения базовых смысловых единиц, называемых ***лексемами,*** или ***токенами*** (от слова ***token).*** Лексический анализатор компилятора языка C++ разделяет входной текст, представляемый в виде потока символов, на лексемы.

К лексемам относятся:

- идентификаторы (имена);
- разделители;
- литералы;
- операторы (знаки операций);
- ключевые слова;
- директивы препроцессора C.

Идентификаторы

Идентификаторы служат для именования таких объектов программы, как типы, константы, переменные, метки и функции. В качестве идентификаторов можно использовать любые последовательности символов, которые удовлетворяют следующим ограничениям:

- □ идентификатор может состоять из букв и цифр;
- □ идентификатор не может начинаться с цифры;
- □ идентификатор не может совпадать ни с одним из ключевых слов.

В C++ нет ограничений на количество символов в идентификаторе, однако некоторые части реализации иногда накладывают такие ограничения.

Разделители

К разделителям относятся:

- □ символ пробел (ASCII-код 32);
- □ управляющие символы (ASCII-коды от 0 до 31);
- □ комментарий.

Основное назначение разделителей — разделять в тексте программы ключевые слова и идентификаторы, разделять слова в строках текстовых файлов, а также замыкать эти строки или сами эти файлы.

Комментарии либо заключаются в скобки вида /* */ и могут быть как однострочными, так и многострочными, либо начинаются с двойной наклонной черты //, в этом случае комментарий будет однострочным.

Литералы

Литералы используются для обозначения в тексте программы чисел, символов и символьных строк, а также ASCII-кодов.

Например:

1	***-1l***	***1L***	десятичные литералы;
lu	***IV***	***1UL***	десятичные литералы;
0x0f	***0xlF***		шестнадцатеричные литералы;
00	***077***		восьмеричные литералы;
0.001	***—0.1***		литералы с фиксированной точкой;
0.00lf	***-0.1F***		литералы с фиксированной точкой;
1.2e+2	***-1E-3***		литералы с плавающей точкой;
'A '			символьный литерал (символ A);
'\n '	'1/2'	'\xa'	символьные литералы (ASCII-код 12);
"Hello"			строковый литерал;

Операторы

В языках программирования операторы или так называемые знаки операций используются для указания операций над объектами программы. К сожалению, в русскоязычной литературе по программированию термин “оператор” связывали и даже сейчас иногда продолжают связывать не с операциями, а с синтаксическими единицами языка — предложениями (от слова ***statement***), которые теперь в

соответствии с их первоначальным смыслом при переводе стандарта современных языков программирования с английского на русский язык уже стали называть инструкциями. Причину такого положения можно объяснить тем, что для языков низкого уровня термин “инструкция”, как и его синоним “команда”, ранее связывали с командой ассемблера, поэтому термин “statement” для языков высокого уровня стали переводить как “оператор”, а не как “инструкцию”.

В современных языках программирования большинство операторов допускает перегрузку, синтаксически и семантически напоминающую (пере)определение функций, поэтому автор оставляет за собой право все, что имеет отношение к термину “statement” в языке C++, называть инструкциями, а не операторами.

В C++ представлены три вида операторов: унарные, бинарные и один тернарный оператор.

Приведем таблицу стандартных операторов языка C++ в порядке убывания их приоритета:

разрешение области видимости	*class name* :: *member*
разрешение области видимости	*namespace name* :: *member*
глобально	:: *name*
глобально	:: *qualifiedname*
выбор члена	*object* . *member*
выбор члена	*pointer* -> *member*
доступ по индексу	*pointer* [***expr***]
вызов функции	***expr*** (***expr_list***)
конструирование значения	***type*** (*expr_list****)***
постфиксный инкремент	*lvalue* ++
постфиксный декремент	*lvalue* --
идентификация типа	***typeid*** (***type***)
идентификация типа во времени выполнения	***typeid*** (***expr***)
преобразование с проверкой во времени выполнения	***dynamic_cast<type>*** (***expr***)
преобразование с проверкой во времени компиляции	***static_cast<type>*** (***expr***)
преобразование без проверки	***reinterpret_cast<type>*** (***expr)***
константное преобразование	***const_cast<type>*** (***expr***)
размер объекта	***sizeof*** *expr*
размер типа	***sizeof (*** *type*)
префиксный инкремент	++ *lvalue*
префиксный декремент	-- *lvalue*
дополнение	~ *expr*
отрицание	! *expr*
унарный минус	- *expr*
унарный плюс	+ *expr*
адрес	& *lvalue*
разыменование	* *expr*

создать (выделить память)	***new*** *type*
создать (выделить память и инициализировать)	***new*** *type* (*expr_list**)***
создать (разместить)	***new*** (*expr_list*) *type*
создать (разместить и инициализировать)	***new*** (*expr_list*) *type*(*exprjist*)
уничтожить (освободить память)	***delete*** *pointer*
уничтожить массив	***delete[]*** *pointer*
приведение (преобразование типа)	(***type)*** *expr*
выбор члена	*object* . * *pointer_to_member*
выбор члена	*pointer* ->* *pointer_to_member*
умножение	*expr* * *expr*
деление	*expr* / *expr*
остаток от деления (деление по модулю)	*expr* % *expr*
сложение	*expr* + *expr*
вычитание	*expr* — *expr*
сдвиг влево	*expr* << *expr*
сдвиг вправо	*expr* >> *expr*
меньше	*expr* < *expr*
меньше или равно	*expr* <= *expr*
больше	*expr* > *expr*
больше или равно	*expr* >= *expr*
равно	*expr* == *expr*
не равно	*expr* ! = *expr*
побитовое И (AND)	*expr* & *expr*
побитовое исключающее ИЛИ (XOR)	*expr* ^ *expr*
побитовое ИЛИ (OR)	*expr* | *expr*
логическое И (AND)	*expr* **&&** *expr*
логическое ИЛИ (OR)	*expr* || *expr*
условное выражение	*expr* ? *expr* : *expr*
присваивание	*lvalue* = *expr*
умножение и присваивание	*lvalue* ***= *expr*

деление и присваивание	*lvalue* **/=** *expr*
остаток от деления и присваивание	*lvalue* %= *expr*
сложение и присваивание	*lvalue* **+=** *expr*
вычитание и присваивание	*lvalue* **-=** *expr*
сдвиг влево и присваивание	*lvalue* **<<-** *expr*
сдвиг вправо и присваивание	*lvalue* >>= *expr*
побитовое И и присваивание	*lvalue* **&=** *expr*
побитовое ИЛИ и присваивание	*lvalue* |= *expr*
побитовое исключающее ИЛИ и присваивание	*lvalue* ^= *expr*
генерация исключения	***throw*** ***expr***
запятая (последовательность)	***expr , expr***

В каждом блоке таблицы расположены операторы с одинаковым приоритетом. Унарные операторы и операторы присваивания правоассоциативны, а все остальные операторы левоассоциативны.

В таблице *class_name* означает имя класса, *namespace_name* — имя пространства имен, *qualified_name* — квалифицированное имя, *name* — имя, *member* — имя члена, *object* — выражение, дающее объект класса, *pointer* — выражение, дающее указатель, *pointer to member* — выражение, дающее указатель на член, *expr* — выражение, *expr_list* — список выражений, *lvalue* — выражение, обозначающее неконстантный объект, *type* — имя типа.

Ключевые слова

Приведем список ключевых слов языка C++:

and	***default***	***friend***	***register***	***true***
andeq	***delete***	***goto***	***reinterpret_cast***	***try***
asm	***do***	***if***	***return***	***typedef***
auto	***double***	***inline***	***short***	***typeid***
bool	***dynamiccast***	***int***	***signed***	***typename***
break	***else***	***long***	***sizeof***	***union***
case	***enum***	***mutable***	***static***	***unsigned***
catch	***explicit***	***namespace***	***staticcast***	***using***
char	***export***	***new***	***struct***	***virtual***
class	***extern***	***operator***	***switch***	***void***
const	***false***	***private***	***template***	***volatile***
constcast	***float***	***protected***	***this***	***wchart***
continue	***for***	***public***	***throw***	***while***

Ключевые слова можно использовать только по своему прямому назначению и их нельзя переопределять, т.е. использовать в качестве идентификаторов.

Директивы препроцессора С

Препроцессор С — это относительно простой обработчик макросов. В дополнение к возможности определять и использовать макросы препроцессор предоставляет механизмы для включения текстовых и стандартных заголовочных файлов, а также для условной компиляции, основываясь на макросах. Все директивы препроцессора начинаются с символа #.

Приведем список директив препроцессора С:

> #***include*** копирует исходный текст из другого файла

#include позволяет:

□ сделать доступным определение интерфейса

□ составить исходный текст из различных частей

> ***Udejine*** определяет макрос (с аргументами или без)

Mefine позволяет:

□ определить:

- символические константы
- открытые подпрограммы
- обобщенные подпрограммы
- обобщенные типы

□ переименовать

□ связать строки

□ определить специализированный синтаксис

□ сделать общую макрообработку

> ***#ifdef*** включает последующие строки в зависимости от условия

ttifdef позволяет:

□ осуществить контроль версий

□ произвести комментирование кода

> ***#endif*** означает конец условного блока, открытого ***ttifdef*** или ***#if***

> ***#undef*** сообщает об отмене всех предыдущих определений идентификатора

> ***#ifndef*** сообщает об отсутствии идентификатора

> ***#if*** включает последующие строки в зависимости от условия ***defined()***

> ***#else*** означает ***else***-часть условного блока, открытого ***#ifdef*** или ***Щ***

> ***#elif*** означает ***"else if'*** и используется при построении вложенных ***#if***

> ***#line*** отменяет автоматическую нумерацию строк

> ***#error*** заставляет реализацию генерировать диагностическое сообщение

> ***#pragma*** влияет на поведение компилятора в зависимости от реализации

ttpragma позволяет:

□ осуществить управление размещением структур данных в памяти

□ информировать компилятор о необычном потоке управления

Наряду с директивами существуют три операции, которые можно использовать только в директивах препроцессора С: подстановка строки (#), конкатенация (##) и подстановка символа **(#@)**. С помощью этих операций осуществляется подстановка имен параметров макросов и отдельных символов, а также динамическое создание имен переменных и макроопределений.

Программа C++ — это набор единиц трансляции, комбинируемых посредством компоновки. Единица трансляции, которую часто называют исходным файлом, — это последовательность объявлений. В общем случае программа C++ состоит из двух частей: раздела объявлений и объявления блока программы. Объявление — это один из видов инструкций языка C++.

Типы

В C++ имеется набор фундаментальных типов, отражающих характерные особенности организации памяти компьютера и наиболее распространенные способы хранения данных.

Приведем список фундаментальных типов языка C++:

- логический тип

bool

- символьные типы

char, signed char, unsigned char

- целые типы

short int {short), signed short int, unsigned short int int, signed int {signed), unsigned int (unsigned) long int {long), signed long int, unsigned long int

- типы с плавающей точкой

float, double, long double

- перечислимые типы
- тип ***void***
- указатели
- массивы
- ссылки
- классы

Перечисления и классы называются ***типами, определяемыми пользователем,*** или ***пользовательскими типами.*** Остальные типы называются ***встроенными типами.*** Логический тип, символьные и целые типы вместе называются ***интегральными типами.*** Интегральные типы вместе с типами с плавающей точкой называются ***арифметическими типами.***

Размер типов определяется реализацией.

Инструкции

К инструкциям языка C++относятся:

- ***помеченная-инструкция***
  идентификатор: инструкция
  case константное выражение : ***инструкция***
  default: инструкция

- ***инструкция-выражение***
  выражение $_{optional}$*;*
- ***составная-инструкция***
  { последовательность инструкций $_{oplional}$ }
- ***инструкция-выбора***
  if (*условие) инструкция*
  if (*условие*) *инструкция* ***else*** *инструкция*
  switch (*условие*) *инструкция*
- ***инструкция-итерации***
  while (*условие) инструкция*
  do *инструкция* ***while*** (*выражение*);
  for { *инструкция инициализации условие* $_{optional}$*; выражение* $_{oplional}$ } *инструкция*
- ***инструкция-перехода***
  break; ***continue;***
  return *выражение* $_{optbmat}$;
  goto *идентификатор;*
- ***инструкция-объявление***
- ***блок-try***
  try { *последовательность инструкций* $_{oplional}$ } *списокобработчиков*

Здесь приняты следующие обозначения:
последовательность инструкций:
инструкция последовательность инструкций $_{optiona1}$

условие:
выражение
спецификатор_типа объявитель = выражение

инструкция инициализации:
инструкция-выражение

простое_объявление
простое_объявление:
спецификатор простого типа идентификатор инициализатор-,

список обработчиков :
catch (*объявление исключения*) { *последовательность_инструкций* $_{optiona1}$ }
список_обработчиков список_обработчиков$_{optiona1}$

Напомним, что *"optional"* означает “необязательно”.

БИБЛИОГРАФИЯ

1. *Шилдт Г.* Теория и практика C++: пер. с англ. СПб.: BHV-Санкт-Петербург, 1996.
2. *Паппас К., Мюррей У.* Руководство программиста по C/C++. В 2 кн. Кн. I. М.: “СК Пресс”,1997.
3. *Паппас К, Мюррей У.* Руководство программиста по C/C++. В 2 кн. Кн. II. М.: “СК Пресс”, 1997.
4. *Сэвитч* У. C++ в примерах: пер. с англ. М.: ЭКОМ, 1997.
5. *Шилдт Г.* Самоучитель C++: пер. с англ. СПб.: BHV-Санкт-Петербург, 1997.
6. *Дейтел Х., Дейтел П.* Как программировать на C++: пер. с англ. М.: Издательство БИНОМ, 1998.
7. *Страуструп Б.* Язык программирования C++: пер. с англ. 3-е изд. СПб.; М.: “Невский Диалект”: БИНОМ, 1999.
8. *Топп У., Форд У.* Структуры данных в C++: пер. с англ. М.: БИНОМ, 1999.
9. *Страуструп Б.* Дизайн и эволюция C++: пер. с англ. М.: ДМК Пресс, 2000.
10. *Вандевурд Д., Джосаттис Н.М.* Шаблоны C++: справ. разработчика: пер. с англ. М.: Вильямс, 2003.
11. *Дёмкин В.М.* Основы объектно-ориентированного программирования в примерах на C++: учеб. пособие. Н. Новгород: НФ ГУ ВШЭ, 2005.
12. *Дёмкин В.М.* Практикум: Объектно-ориентированное программирование в примерах на C++: учеб. пособие. Н. Новгород: НФ ГУ ВШЭ, 2005.

Содержание

УДК 004.438
ББК 32.97-018.1
Д30

Издание осуществлено в рамках
Инновационной образовательной программы ГУ ВШЭ
<<Формирование системы аналитических компетенций
для инноваций в бизнесе и государственном управлении>>

Дёмкин, В. М. Императивное программирование в примерах на С++: прак- ДЗО тикум [Текст] / В. М. Демкин ; Гос. ун-т — Высшая школа экономики. — М. : Изд. дом ГУ ВШЭ, 2007. — 193, [3] с. — 1000 экз. — ISBN 978-5-7598-0487-1 (в обл.).

Практикум предназначен для самостоятельного изучения одного из разделов курса <<Информатика>> — императивного программирования на языке С++. Обсуждаются вопросы программирования классических типов алгоритмов—линейных, условных, циклических, рекурсивных и эвристических. Учебные примеры в виде программных реализаций с комментариями и протоколами результатов работы программы иллюстрируют принципы императивной парадигмы программирования с помощью управляющих структур и структур данных. Программный код апробирован на современных компиляторах платформ Windows и Linux. Практикум подготовлен при поддержке Научного фонда ГУ ВШЭ в рамках Инновационной образовательной программы.

Рецензент— доктор физико-математических наук профессор *С.Н. Митяков*

Зав. редакцией *О А. Шестопалова*
Редактор *И.В. Башнина*
Художественный редактор *А. М. Павлов*
Компьютерная верстка: *В. М. Дёмкин*
Корректор *А А. Папушина*
OCR by Palek

Подписано в печать 14.11.2007. Формат 60x84 1/8. Гарнитура Times New Roman.
Печать офсетная. Бумага офсетная № 1. Усл. печ. л. 11,39. Уч.-изд. л. 14,13. Тираж 1000 экз. Заказ № 7654. Изд. № 752.
ГУ ВШЭ. 125319, Москва, Кочновский проезд, д. 3 Тел./факс: (495) 772-95-71
Отпечатано в соответствии с качеством предоставленных диапозитивов в ФГУП <<Производственно-издательский комбинат ВИНИТИ>>, 140010, г. Люберцы Московской обл., Октябрьский пр-т, 403 при содействии ООО <<МАКС Пресс>>
105066, г. Москва, Елоховский пр., д. 3, стр. 2. Тел.: 939-38-90,939-38-93. Тел./факс: 939-38-91.