Е.К. Липачёв

Технология программирования. Базовые конструкции C/C++

Учебно – справочное пособие

Казанский университет

2012

УДК 004.43

ББК 32.973.26–018

Печатается по решению Редакционно-издательского совета ФГАОУВПО «Казанский (Приволжский) федеральный университет» методической комиссии Института математики и механики им. Н.И.Лобачевского от 19 апреля 2012 г.

Научный редактор –
д-р физ.-мат. наук **Ф.Г. Авхадиев**

Рецензенты:

канд. физ.-мат. наук **А.Ф. Галимянов**
канд. физ.-мат. наук **М.Ф. Насрутдинов**

Липачёв Е.К.

Технология программирования. Базовые конструкции C/C++: учебно–справочное пособие / Е.К. Липачёв. – Казань: Казан. ун-т, 2012. - 142 с.

Предназначено студентам, изучающим программирование на естественно-научных факультетах высших учебных заведений.
За основу взяты лекции по курсу «Технология программирования и работа на ЭВМ» по направлению подготовки 010800 Механика и математическое моделирование, прочитанные автором студентам механико-математическом факультета (Институт математики и механики им. Н.И. Лобачевского) Казанского (Приволжского) федерального университета.

УДК 004.43
ББК 32.973.26–018

Оглавление

Введение

В этом руководстве два языка программирования C и C++ рассматриваются в одной связке. Это нужно пояснить. Оба языка имеют общие конструкции, язык C++ создан на основе языка C и является его расширением. Программы, написанные на “чистом” языке C обрабатываются компиляторами C++. Сложившаяся практика преподавания компьютерных технологий в Вузе в основной части ориентирована на работу в операционных системах MS Windows и, как правило, программирование осваивается в средах MS Visual Studio или C++ Builder. При программировании в этих средах потребуется кропотливая настройка параметров проекта, чтобы провести различие языков C и C++.

Руководство предназначено студентам, изучающим программирование. По языкам C и C++ имеется достаточно обширная литература, более того, большинство изданий, как на русском, так и на английском языках, легко найти в сети. Кроме книг, с приемами программирования можно познакомиться на сайтах, посвященных языкам C и C++, – в разделе “Электронные ресурсы” приводится список некоторых из них. Подробная справочная книга по языку программирования, вряд ли, нужна сейчас. В среде программирования имеется многоуровневая справочная служба, а при написании кода на помощь приходят технологии автодополнения (IntelliSense, Code Completion). Поэтому в руководстве акцент сделан на использование базовых конструкций языка и на их взаимодействие. Примеры, сопровождающие описание элементов языка, носят не только иллюстративный характер, – во многих из них вводятся дополнительные конструкции.

Структура программы

Программа строится из отдельных блоков, называемых функциями. Каждая функция, в свою очередь, составлена из операторов, которые являются инструкциями для компьютера.

В любой программе на языках C и C++ должна быть функция `main()`.

Функция `main()`

```
int main()
{
…
return 0;
}
```

Структуру программы, содержащей функции, см. в разделе “Функции”.

Комментарии

Комментарии – это строки с текстом, поясняющим логику работы программы. В языках C и C++ используются два типа комментариев. Многострочный комментарий начинается с символов “`/*`”, за которыми записывается комментирующий текст, символы “`*/`” обозначают завершение комментария. В первоначальных версиях языка C использовался только этот тип комментирования. Комментарии этого типа можно поставить всюду, где разрешен пробел.

Второй тип комментариев – однострочный. Он начинается последовательностью символов “`//`” и заканчивается концом строки.

Пример.

```
/* cascade.cpp:
  Реализация каскадного алгоритма.
  См. И.Добеши, Десять лекций по вейвлетам.
*/
#include <iostream>
#include <cmath>
using namespace std;
const int n=2; // Порядок вейвлета Добеши
int main()
{
     int m; // Глубина каскадного алгоритма
     cin >> m;
     …   ….   ….  …
   while ((r<3/*пр.граница носителя*/)&&(r>0))
     return 0;
}
```

Правила комментирования – вопрос стиля программирования, которого придерживается программист или команда разработчиков. С рекомендациями по правилам программирования и, в частности, документированию кода, можно познакомиться в книге Голуб А. *Правила программирования на Си и Си++*. Отличающиеся подходы к вопросу документирования сформулированы в концепциях ***Literate Programming*** (***Грамотное программирование***) и ***Extreme Programming*** (также ***XP*** и ***Экстремальное программирование***). Ссылки можно найти в сети Интернет.

Отметим также, что комментирование активно используется при отладке программ – часть операторов можно просто “закомментировать” и, тем самым, скрыть от компилятора, что, возможно, поможет локализовать ошибку.

Директивы препроцессора

Каждая программа на C/C++, как правило, начинается с директив препроцессора – их можно узнать по символу “#”.

Например, если в программе присутствуют операторы ввода и вывода, то необходимо добавить инструкции

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

или

```
#include <iostream.h>
```

Препроцессор обрабатывает код программы на первом шаге компиляции. Его задача – выполнение текстовых преобразований в соответствии с директивами. Встретив директиву

`#include "имя-файла"` или `#include <имя-файла>`,

препроцессор заменит такую строку на код, записанный в указанном файле.

Разница в использовании кавычек (" ") и угловых скобок (< >) заключена в пути поиска: файл, обрамленный угловыми скобками, ищется в системных каталогах среды программирования, а файл в кавычках сначала ищется в рабочем каталоге программы.

Программы C/C++ обычно имеют несколько директив `#include`. Файлы, добавляемые препроцессором, называют заголовочными, и они традиционно имеют расширение `.h`.

Директива макроподстановки:

```
#define имя    текст
```

используется для замены: всюду в программе, где будет найдено **`имя`**, будет произведена замена на **`текст`**.

Директиву `#define` в языке ранних версиях С использовали для создания констант

```
#define  PI   3.1415
```

Встретив такую инструкцию, препроцессор заменит в тексте программы все вхождения `PI` на `3.1415`.

Отметим, что в файле `math.h` для числа π определена константа

```
#define M_PI        3.14159265358979323846
```

В файле `limits.h` (каталог `/include` в установочном каталоге среды разработки) с помощью `#define` заданы минимальные и максимальные значения числовых типов, например,

```
#define INT_MAX  2147483647 /* maximum (signed) int value */
```

Директивы можно `#define` также используется для создания макросов.

Пример.

```
   #define max(a,b)    (((a) > (b)) ? (a) : (b))
… ….          ….          ….              …
   float x;
   cin>>x;
   float y = max(sin(x), cos(x));
```

Имеется несколько управляющих инструкций выборочного включения.

```
#if условие
.....            строки
#endif
```

`строки` выполняются только в том случае, когда **`условие`** истинно.

```
#if условие
....             строки-1
#else
...              строки-2
#endif
```

если **`условие`** истинно, выполняются **`строки-1`**, если же ложно – **`строки-2`**.

```
#ifdef идентификатор
.....            строки
#endif
```

строки выполняются только в случае, если **идентификатор** ранее определен директивой `#define`.

```
#ifndef идентификатор
.....              строки
#endif
```

наоборот, **строки** выполняются только в случае, если **идентификатор** еще не был определен директивой `#define`.

Эти директивы исключают повторение одних и тех инструкций при сборке многофайлового проекта – сразу несколько файлов проекта могут содержать одинаковые препроцессорные директивы.

Пример. Фрагмент кода из файла `\include\minmax.h`

```
#ifndef max
#define max(a,b)  (((a) > (b)) ? (a) : (b))
#endif
```

Пример. Фрагмент кода из файла `\include\stdio.h`

```
#ifndef _INC_STDIO
#define _INC_STDIO
#include <crtdefs.h>
#endif
```

– содержимое файла `crtdefs.h` будет включено только однажды, а именно, при самой первой обработке инструкции `#include <stdio.h>` в процессе сборки файлов проекта.

Блок

```
#ifndef идентификатор
#define идентификатор
…   …   …   …
#endif
```

называют ***стражем включения*** (include guard).

Подробное изложение возможностей препроцессора можно найти в электронной книге Stallman R.M., Weinberg Z.. *The C Preprocessor*. - Free Software Foundation, Inc., 2011. - 83 p. - http://gcc.gnu.org/onlinedocs/cpp.pdf

Представление данных

Переменная – это именованная область памяти, к которой можно обращаться из программы, записывая и извлекая из нее данные. Каждая переменная относится к определенному типу, задающему размер памяти,

диапазон значений хранимых данных, а также возможных набор операций с данными.

Имя переменной, или *идентификатор*, можно образовывать из латинских букв, цифр и символа подчеркивания. Идентификатор не может начинаться с цифры. ***Прописные и строчные буквы в именах различаются.***

В современных версиях языка нет ограничений на длину идентификатора. Но, все же, ограничение есть – на длину значащих для компилятора символов. Так, стандарт C99 языка C (см. http://gcc.gnu.org/c99status.html; http://www.open-std.org/JTC1/SC22/WG14/www/docs/ n1336.pdf) устанавливает ограничение в 31 начальных символа для внешнего идентификатора и до 63 символов – для внутреннего. Такое же ограничение и в "свежем" стандарте C11 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf). Едва ли нужно запоминать эти числа – вряд ли в программе будут использоваться идентификаторы с таким большим числом совпадающих символов в начале.

Следует учитывать, что в C/C++ имеются ключевые слова (напр., названия типов, операторов) и они не могут быть использованы в качестве идентификаторов. Эти ключевые слова указаны в стандарте.

При составлении имен желательно придерживаться какого-либо стиля, например, стиля ***CamelCase***, согласно которому идентификатор образуется из нескольких слов, отражающих назначение идентификатора, при этом, слова пишутся слитно без пробелов и каждое слово пишется с заглавной буквы. Различают два варианта этого стиля: ***UpperCamelCase*** (или ***PascalCase***) и ***lowerCamelCase***. Различие стилей – в выборе регистра начальной буквы идентификатора. Стиль UpperCamelCase используют для наименования типов (в том числе классов), а lowerCamelCase – для переменных, экземпляров классов, функций и методов.

Распространенным является также стиль, основанный на использовании символа подчеркивания в именах.

Отметим также "***венгерскую нотацию***" (Hungarian Notation) программиста-космонавта Чарльза Симони (Charles Simonyi) – см., напр., http://msdn.microsoft.com/en-us/library/Aa260976.

Примеры.

```
class SomeClass; // тип
SomeClass oneClass; // объект
int oneNumber; // переменная
int one_Number; // переменная
getName(); // функция
```

Несколько слов о стиле программирования

Стиль программирования – набор рекомендаций написания программного кода. Стиль, в частности, предлагает систему образования имен переменных, типов и функций, правила расстановки скобок и использования пробельных отступов (создание “лесенки”). В предыдущих разделах уже отмечены стиль написания идентификаторов CamelCase и правила комментирования. Фигурные скобки можно расставить несколькими способами – например, в книге Хэзфилд Р., Кирби Л., Корбит Д. и др. *Искусство программирования на C* выделено 4 стиля расстановки скобок.

Пример. Стили расстановки скобок.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void func();

int _tmain(int argc, _TCHAR* argv[])
{
    const int N=10;
    int i,j,k,m;
    // стиль 1TBS:
    for (i=0;i<N;i++)
    {
        func();
    }
    //  стиль  BSB:
    for (j=0;j<N;j++) {
        func();
    }
    //  стиль Whitesmith
    for (k=0;k<N;k++)
        {
        func();
        }
    //  стиль GNU
    for (m=0;m<N;m++)
        {
            func();
        }

    return 0;
}

    void func()
```

```
{
    cout<<"\nStyle";
}
```

Развернутое изложение основ стиля программирования можно найти в книге Саттер Г., Александреску А. *Стандарты программирования на C++. 101 правило и рекомендация*, а также "*C++ Programming Style Guidelines*" – http://geosoft.no/development/cppstyle.html и "*C++ Coding Conventions*" – http://www.c-xx.com/ccc/ccc.php

Встроенные типы

К базовым типам в C/C++ относятся: тип `char` — для хранения отдельных символов и небольших целых чисел, тип `int` — для работы с целыми числами, тип `float` для представления чисел с плавающей точкой, тип `double` — для чисел с двойной точностью, и тип `void` — без значения.

Пример.

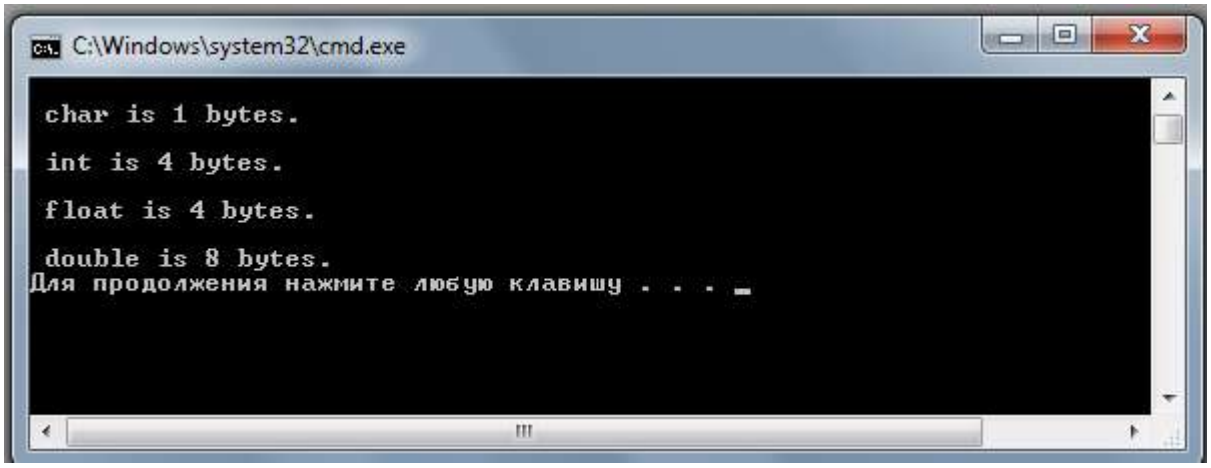
```
char ch='Ё';
char ch2=127;
int i=10; // десятичная запись
int j=0x16; // шестнадцатеричная запись
float eps=1.e-8; // =0.00000001
double pi=3.141592654;
float _pi=3.14159;
```

Размер памяти, выделяемый для хранения данных определенного типа, зависит от среды программирования, только у типа `char` – это 1 байт.

С помощью оператора `sizeof()` можно узнать размер памяти (в байтах), отведенный для указанного типа.

Пример.

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

int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"\n char is "<<sizeof(char)<< " bytes.\n";
    cout<<"\n int is "<<sizeof(int)<< " bytes.\n";
    cout<<"\n float is "<<sizeof(float)<<" bytes.\n";
    cout<<"\n double is "<<sizeof(double)<<" bytes\n";
    return 0;
}
```



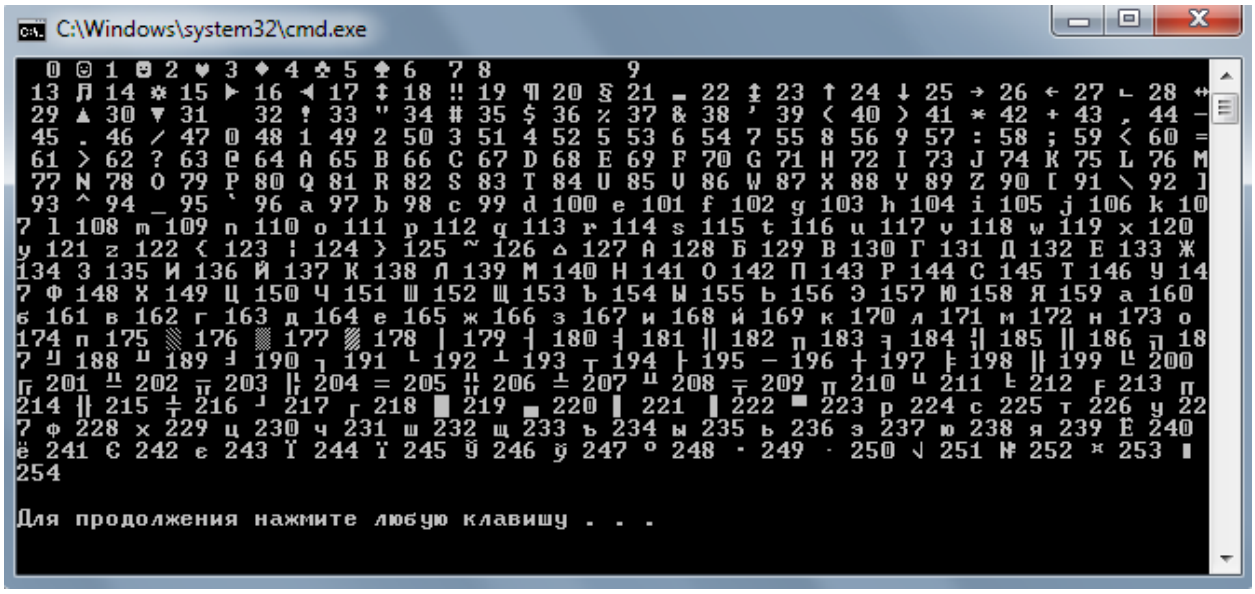
Помимо базового типа `int` для целочисленных данных используются типы: `unsigned char`, `signed char`, `unsigned int`, `short int`, `long int`, `long long int`, `signed long int`, `unsigned long int`, `unsigned long long int`.

Эти типы различаются только диапазоном значений, которые могут принимать числа, а объем памяти зависят от среды программирования – определить размер выделяемой памяти можно с помощью `sizeof()` также как в приведенном примере.

Замечание. В большинстве компиляторов `unsigned char` и `char` совпадают. Однако, при обработке текстов, содержащих кириллицу, лучше не полагаться на правила умолчания и указывать тип `unsigned char`. В следующем примере выводится таблица символов, из которой видно, что все символы кириллицы имеют код, больший, чем `125` (предельное значение для `signed char`).

Пример. Программа печатаем символы и их коды. Так, например, символ ‘ё’ имеет код `241`.

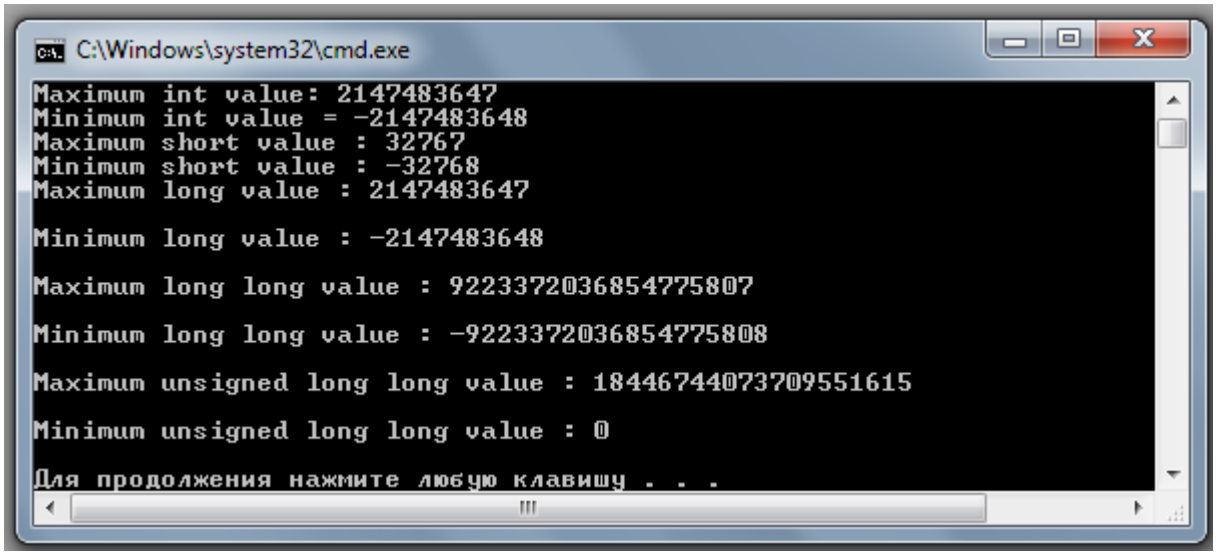
```
// таблица символов
unsigned char ch; //
int i;
for (i=0;i<255;i++) {ch=i; cout<<ch<<' '<<i<<' '; }
cout<<"\n\n";
```



В заголовочном файле /include /climits содержится информация о предельных значениях целочисленных данных. Можно заглянуть в этот файл или вывести с помощью программы

Пример.

```
#include <iostream>
#include <climits> // не обязательно
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
     cout << "Maximum int value: " << INT_MAX << '\n';
     cout << "Minimum int value = " << INT_MIN << '\n';
     cout << "Maximum short value : " << SHRT_MAX << '\n';
     cout << "Minimum short value : " << SHRT_MIN << '\n';
     cout << "Maximum long value : " << LONG_MAX << "\n";
     cout << "Minimum long value : " << LONG_MIN << "\n";
     cout << "Maximum long long value:" << LLONG_MAX << "\n";
     cout << "Minimum long long value : " << LLONG_MIN << "\n";

     cout << "Maximum unsigned long long value : "
               << ULLONG_MAX << "\n\n";
     cout << "Minimum unsigned long long value:"<< 0 << "\n";
    // Аналогично остальные типы целых
     return 0;
}
```



Для поддержки операций с плавающей точкой помимо `float` и `double` имеется тип расширенной точности `long double`.

Операции

Основные классы операций в C/C++: арифметические, логические, поразрядные и операции сравнения.

Операции обозначаются специальными знаками (см. далее).

Операции, применяемые к одному операнду, называются *унарными* (например, операция определения адреса (`&`)), а применяемые к двум операндам – *бинарными* (например, операция сложения чисел).

Выражение состоит из одного или нескольких операндов, в простейшем случае – это имя переменной.

Арифметические операции

Символ операции	Значение	Использование
*	Умножение	`expr * expr`
/	Деление	`expr / expr`
%	Остаток от деления	`expr % expr`
+	Сложение	`expr + expr`
-	Вычитание	`expr - expr`

Пример.

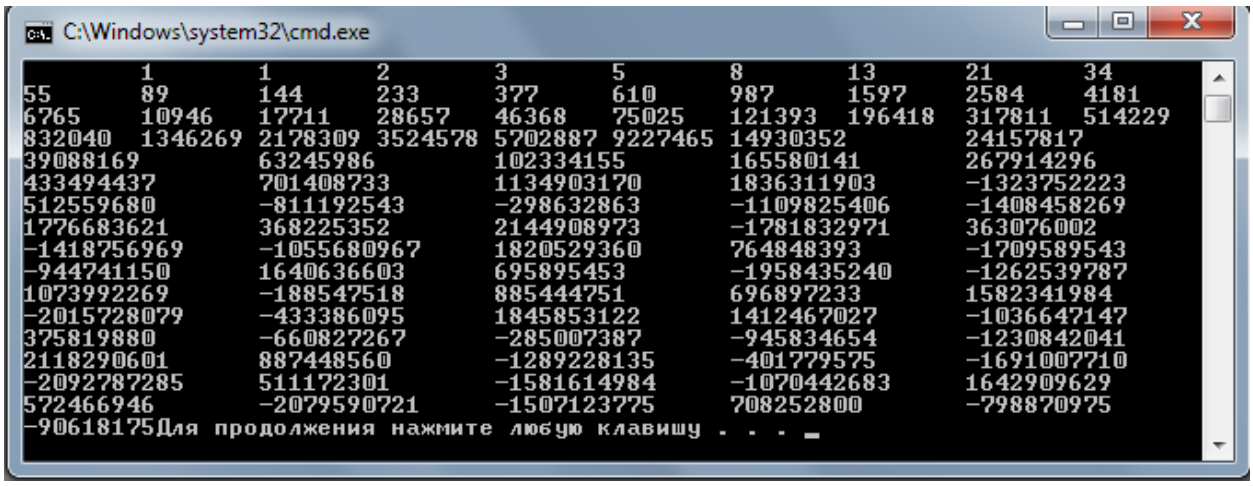
```
//    Программа определяет четность целого
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{   int num;
    cout << "Vvesti chislo:";
```

```
    cin >> num;     // чтение числа
    // проверка на четность
    if((num%2)==0) cout << "Even \n";
    else cout << "Odd \n";
    return 0;
}
```

Результат вычисления арифметического выражения может оказаться неопределенным. Это может возникнуть, например, из-за *переполнения* – значение превысит допустимый предел значений данного типа. В этих случаях говорят об арифметических исключениях, хотя программа продолжит свою работу и не последует никаких сообщений об ошибке.

Пример. Программа вычислений чисел Фибоначчи. Хотя алгоритм реализован правильно, и программа успешно завершается, но результаты, начиная с 47-го числа, неверные – мы видим даже отрицательные числа. Причина – тип `int` не может хранить большие числа. Использование `long long int` способно исправить ситуацию всего лишь для первых 92 чисел, а с помощью `unsigned long long int` получим 93 правильных числа (можно сравнить со значениями, приведенными на странице http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibtable.html), см. также пример в разделе “Цикл `for`”.

```
// fibb.cpp: Вычисление чисел Фибоначчи
//
#include "stdafx.h"
#include <iostream>
using namespace std;
const int n=100;
int _tmain(int argc, _TCHAR* argv[])
{
    int f,g,h;
    g=0; h=1;
    for (int i=2;i<n;i++){
        f=g+h;
        cout<<'\t'<<f;
        h=g;
        g=f;
    }
    return 0;
}
```



Замечание. Решить эту задачу, т.е. получить все значащие цифры большого числа, можно с привлечением длинной арифметики (см., напр. Кнут Д. *Искусство программирования, том 2. Получисленные алгоритмы*.).

Операции сравнения и логические операции

Символ операции	Значение	Использование
!	Логическое НЕ	`!expr`
<	Меньше	`expr1 < expr2`
<=	Меньше или равно	`expr1 <= expr2`
>	Больше	`expr1 > expr2`
>=	Больше или равно	`expr1 >= expr2`
==	Равно	`expr1 == expr2`
!=	Не равно	`expr1 != expr2`
&&	Логическое И	`expr1 && expr2`
||	Логическое ИЛИ	`expr1 || expr2`

Пример.

```
int x;
cin>>x;
bool bx=(x>=0) && (x<=100); // bx=1 (true), если 0<=x<=100
bool bbx=(x<0) || (x>100); // bbx =0 (false), если 0<=x<=100
bool bbbx=!((x>=0) && (x<=100));// bbbx =0, если 0<=x<=100
```

Поразрядные операции

Поразрядные (побитовые) операции — это операции над отдельными битами данных и могут применяться только к данным, имеющим тип `char` или тип `int`. В поразрядных операциях не могут участвовать данные других типов.

Побитовые логические операции “`&`” (побитовое И), “`|`” (побитовое ИЛИ), “`^`” (побитовое исключающее ИЛИ). “`~`” (двоичное дополнение)

позволяют выполнить установку значений битов. Операции производятся по всем известным таблицам истинности.

Пример. Проверка значения бита.

```
char bb=0x64; // = 100 в дес.с.с. и 1100100 - в дв.с.с.
if (bb & 4) cout<<"\nТретий бит равен 1";
else cout<<"\nТретий бит равен 0";
if (bb & 8) cout<<"\nЧетвертый бит равен 1";
else cout<<"\nЧетвертый бит равен 0";
if (bb & 32) cout<<"\nШестой бит равен 1";
else cout<<"\nШестой бит равен 0";
```

Побитовые операции сдвига “>>” и “<<” сдвигают все биты переменной, соответственно, вправо или влево. Общая форма операторов сдвига:

```
переменная >> количество_разрядов
переменная << количество_разрядов
```

При сдвиге битов в один конец числа, другой конец заполняется нулями.

Операции сдвига можно использовать для быстрого умножения (и деления) на степени числа 2.

Пример. Умножение и деление на степени числа 2.

```
int a,b;
a=3; b=2048;
int n=a<<5; // n= a*32
int m=b>>5; // m=b/32
int x= (a<<3) + (a<<2); // x=a*12 = a*8 + a*4
```

Замечание. В последнем операторе примера показано, как применить операции сдвига при умножении на числа, отличные от степеней 2, в данном случае – умножение на 12. Скобки в операторе `x= (a<<3) + (a<<2)` необходимы, – это связано с тем, что сложение имеет более высокий приоритет, чем операции сдвига.

Операции присваивания

В результате операции присваивания переменная получает новое значение.

Общая форма оператора присваивания:

```
идентификатор = выражение;
```

Оператор присваивания может присутствовать в любом выражении языка.

Пример. Оператор присваивания в арифметическом выражении.

```
int a, b, c;
```

```
c=(a=2)+(b=3);
```

Несколько операций присваивания могут быть объединены, например,

```
int i, j;
i = j = 0; //присваивание 0 обеим переменным

i = (j = 3)*2 +7;
```

Такое присваивание называют ***множественным***.

При множественном присваивании вычисления производятся справа налево:

```
i = j= k = m=1;
```

Сначала переменная `m` получает значение `1`, затем `k` получает значение m, `j` – значение результата этого присваивания, и в завершении – `i` получает значение `j`.

В языках C и C++ используются также ***составные операции присваивания***.

Общий синтаксис составного оператора присваивания:

`a op= b;`

где `op=` является одним из операторов: `+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=.`

Запись `a op= b` эквивалентна записи `a = a op b`.

Операции инкремента и декремента

Операции инкремента или, по-русски, увеличения (++) и декремента, т.е. уменьшения (--) дают возможность компактной записи для изменения значения переменной на единицу.

Выражение `n++` является *постфиксной* формой оператора инкремента. Значение переменной `n` увеличивается *после того*, как ее текущее значение употреблено в арифметическом выражении. Аналогично, выражение `m--` является *постфиксной* формой оператора декремента.

Существует и *префиксная* форма этих операторов: `++n`, `--m`. При использовании такой формы текущее значение переменной сначала увеличивается или уменьшается и только потом используется в арифметическом выражении.

Пример.

```
int n, m,k,j;
n=m=2;
k=++n * 2; // k=6
j=2 * m++; // j=4
```

Пример. Возможная неоднозначность совместного использования операторов в префиксной и постфиксной формах.

```
#include <iostream>
using namespace std;
int main()
{    int i=2, j=2;
     int k=(i++) * (i++) * i;//k=8
     int m=(++j) * (j++) * j;//m=?
     cout<<”k= ”<<k<<” i= ”<<i<<” m= ”<<m<<” j= ”<<j;
     return 0;
}
```

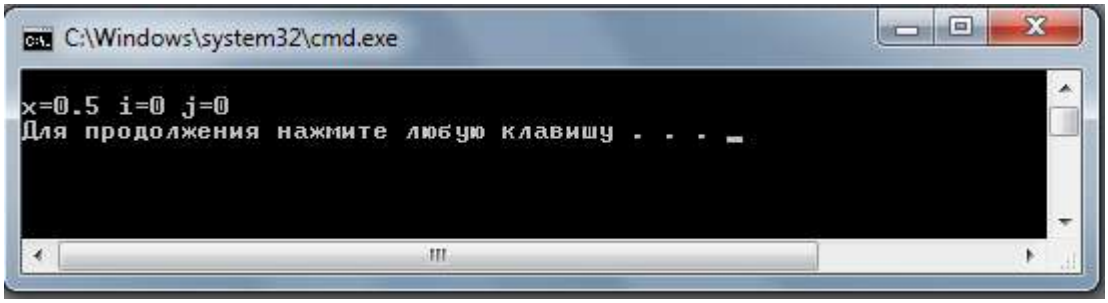
Стандартная рекомендация – вместо эквивалентных (по значению) операторов присваивания использовать операторы инкремента и декремента. Это связано с тем, что компиляторы создают для них более эффективный код.

Преобразование типов в операции присваивания

Если в операции присутствуют переменные разных типов, компилятор производит, если это возможно, преобразование типов. Значение выражения в правой части оператора присваивания преобразуется к типу переменной в левой части.

Пример. Потеря информации при преобразовании.

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
int i, j;
double x=0.5;
i=x;
j=1/2+1/2;
return 0;
}
```



Операция “запятая”

Одно выражение может состоять из набора выражений, разделенных запятыми; например,

```
d=(b=a+1),c=a+2;
a=1,2,3;
```

такие выражения вычисляются слева направо. Результатом всего выражения будет результат самого правого выражения списка. Эту операцию, её ещё называют ***операцией последовательного вычисления,*** используют в тех случаях, когда нужно вычислить несколько выражений, а по правилам синтаксиса допускается только одно выражение, как, например, в операторе `for`.

Пример. Несколько выражений в выражении инициализации оператора цикла `for`.

```
for (x = 4,u = a[n],k=n-1;k>=0;k--) u = u*x + a[k];
```

в данном случае, группа выражений `x = 4,u = a[n],k=n-1` рассматривается как одно и имеет значение `k=n-1`.

Операция “запятая” обладает самым низким приоритетом, – это означает, что оператор,

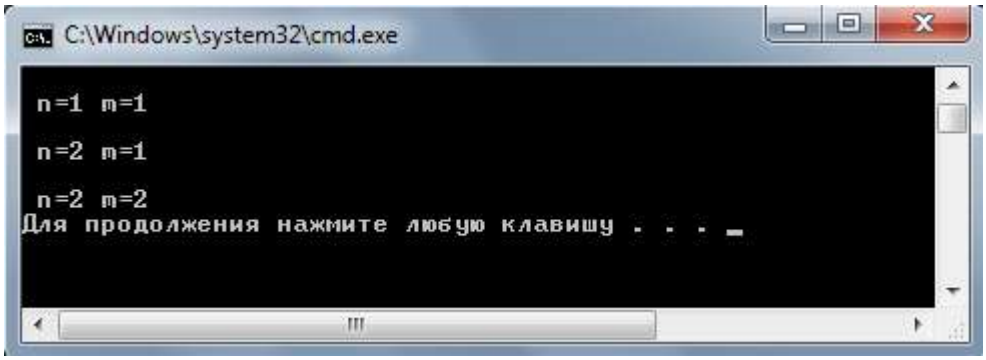
```
y=1,2;
```

воспринимается как

```
(y=1),2;
```

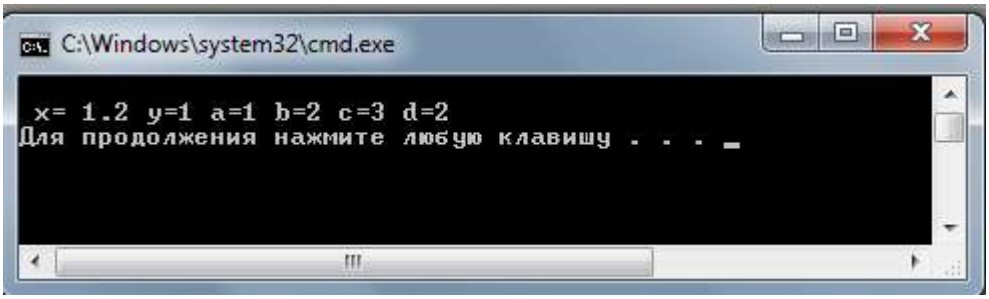
Пример. Операция запятая – расстановка скобок влияет на результат.

```
using namespace std;
int main()
{
 int n,m;
 n=m=1,2;
 cout<<"\n n="<<n<<" m="<<m<<'\n';
 n=(m=1,2);
 cout<<"\n n="<<n<<" m="<<m<<'\n';
 n=m=(1,2);
 cout<<"\n n="<<n<<" m="<<m<<'\n';
 return 0;
}
```



Пример. Варианты использования операции “запятая”

```
double x,y;
x=1.2;
y=1,2;
int a,b,c,d;
a=1,2,3;
d=(b=a+1),c=a+2;
```



Приоритеты операций

Приоритеты операций задают последовательность вычислений в сложном выражении.

В C/C++ умножение и деление имеют более высокий приоритет, чем сложение, поэтому они будут вычислены раньше. Их собственные приоритеты равны, поэтому умножение и деление будут вычисляться слева направо. Самый низкий приоритет у операции “запятая”.

Далее приведен список операций (с рядом из них предстоит еще познакомиться) в порядке убывания приоритета: {“`( )`” (скобки функций), “`[ ]`” (скобки для индексов массивов), “`->`” , “`.`” (операции доступа к элементам структур)}, {“`!`”, “`~`”, “`++`”, “`--`“, “`+`” (унарный плюс), “`-`“ (унарный минус), “`(type)`”, “`*`” (операция указателя – “разыменование”), “`&`” (операция адреса), “`sizeof`”} , {“`*`”, “`/`”, “`%`”}, {“`+`”, “`-`”}, {“`<<`”, “`>>`”}, {“`<`”, “`<=`”, “`>`”, “ `>=`”}, {“`==`”, “`!=`”}, “`&`”, “`^`”, “`|`”, “`&&`”, “`||`”, “`?:`”, {“`=`”, “`+=`”, “`-=`”, “`*=`”, “`/=`”, “`%=`”}, “`,`”. Фигурные скобки объединяют группы операций с одинаковым приоритетом.

Преобразование типов

В арифметических операциях все операнды предварительно приводятся к одному типу. Чтобы избежать потери точности, используется принцип перехода от операнда меньшего типа к большему, например, если встречаются переменные, принадлежащие к типам `int`, `float` и `double`, то производится преобразование к типу `double`. Этот процесс преобразования типов называется *продвижением типов* (*type promotion*) (подробнее, см., напр., Шилдт Г. *Полный справочник по C* и *Программирование на C и C++*. – http://cpp.com.ru/).

В C/C++ имеется операция *явного приведения типа*, т.е. можно указать, к какому типу необходимо преобразовать значение выражения. Общая форма оператора явного приведения типа:

(Тип) выражение

Пример. Явное приведение типа в арифметическом выражении.

```
double  x;
int m;
cin>>m;
x= (double) m/2 + (double)7/3;
```

В языке C++ используется также приведение типа в форме вызова функции

Тип (выражение)

Пример.

```
double  x;
int m;
cin>>m;
x= double(m/2) + double(7)/3;
```

Инструкции

Выражение, составленное по правилам C/C++ (например, `i + 5`), становится *простой инструкцией*, если после него поставлена точка с запятой. *Составная инструкция* – это последовательность простых инструкций, заключенная в фигурные скобки. Инструкции выполняются в порядке их записи.

Простейшей формой является пустая инструкция, состоящая только из точки с запятой:

```
; // пустая инструкция
```

Пример.

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

```
cin>>i;
j = i * 2; // простая инструкция
for (; // пустая инструкция
    i<30; )
 {
   // составная инструкция
   k=i+j;
   i=j;
   j=k;
   cout<<"\n "<<k;
 }
```

Инструкция `if`

Инструкция **`if`** обеспечивает выполнение или пропуск инструкции в зависимости от условия

`if (проверяемое условие) оператор`

условие заключается в круглые скобки и образуется с помощью логических операций.

Проверяемое условие может быть выражением и должно иметь скалярный результат, т.е. быть целым числом, числом с плавающей точкой, символом или указателем. Массивы или структуры не могут являться значениями проверяемого условия. Любое ненулевое значение проверяемого условия рассматривается как `true` (истина), а нулевое — `false` (ложь).

Пример.

```
/*  сравнение символов */
char ch='\n';
if (ch=='\n') cout <<"\n ch= \\n";

// сравнение целых
int n=5;
if (n==5) cout<<"\n n = 5";

/* сравнение чисел с плавающей точкой: */
double d=0.00001;
if (d==1.e-5) cout<<"\n d = 0.00001";
if (fabs(d-1.e-5)<=1.e-6) cout<<"\n d = 0.00001";/*
числа равны с точностью до 6 знач. цифры*/

/* определение диапазона: */
int t=25;
if ((t>10) && (t<30)) cout<<"\n 10 < "<<t <<" <30";
```

```
// арифметическое выражение как условие
int x;
cin>>x;
if (x*x) cout<<"\n true ";
else cout<<"\n false ";

if (x)
{
     // составная инструкция
     x--;
     cout<<"\n x= "<< x;
}
```

Инструкция **if-else** содержит уже два оператора, один из которых из которых выполняется

if (проверяемое условие)
оператор1
else оператор2

если **проверяемое условие** имеет ненулевое значение, выполняется **оператор1**, а при нулевом значении – **оператор2**.

Пример.

```
double x,f;
cout<<"\n x= "; cin>>x;
if ((x>=0) && (x<1)) f=1; // Функция Хаара
else f=0;
```

Операторы `if` могут содержать другие операторы `if`. Такие операторы называют ***вложенными условными операторами***. Во вложенном условном операторе конструкция `else` относится к ближайшему оператору `if`, – однако, этот порядок можно изменить расстановкой фигурных скобок.

Согласно Стандарту C89 языка C допускается до 15 уровней вложенности условных операторов, в Стандарте C99 разрешено уже 127 уровней.

Чтобы не запутаться в блоках `if`, в условных операторах используют абзацные отступы (подробнее, см., напр., Голуб А. *Правила программирования на Си и Си++*.)

Прием, известный как ***лестница if-else-if*** (см., напр., Шилдт Г. *Полный справочник по C*.), позволяет упорядочить вложенные блоки `if`.

if (*условие*) *оператор*;

else

```
if (условие) оператор;
else
  if (условие) оператор;
  .
  .
  .
  else оператор;
```

В “лестнице” ***условия*** операторов `if` вычисляются сверху вниз. Если встретилось условие с ненулевым значением (т.е. условие истинно), выполняется оператор этого блока `if`, а оставшаяся часть лестницы пропускается. Если все условия ложны, то выполняется оператор в последнем блоке `else`, или не выполняется ни один оператор, если лестница не заканчивается `else`

Пример. Функция преобразования строки из кодировки ANSI в кодировку OEM-866 (иначе DOS-кодировка)

```
void ansi2oem(char *stroka)
{
int cnt,i=0;
char ch;
while((ch=stroka[i])!='\0')
{
    cnt=ch;
    if ((ch>='a') && (ch<='п')) cnt-=64;
     else if ((ch>='p') && (ch<='я')) cnt-=16;
          else if (ch=='ё') cnt=241;
               else if (ch=='Ё') cnt=240;
                 else if((ch>='A')&&(ch<='Я')) cnt-=64;
    stroka[i]=cnt; i++;
}
}
```

Замечание. Функцию `ansi2oem()` можно использовать для русификации вывода консольных приложений. См. раздел “Строки”.

Оператор ?:

Вместо оператора `if-else` часто используется оператор “`?:`”.

выражение1 ? выражение2 : выражение3

Если **выражение1** истинно (т.е. отлично от нуля), то значение всего оператора равно значению **выражение2,** в противном случае значением всего выражения будет значение, полученное после вычисления **выражение3.**

Отметим, что оператор "?:" имеет три операнда, т.е. является ***тернарным***.

Пример.

```
znak = (x>0)? 1 : -1;
```

Пример. Оператор "?:" в арифметическом выражении

```
double x, y;
cout<<"\n x= "; cin>>x;
y = x * x *((x>0)?1:-1);
```

Пример. Вложенные операторы "?:"

```
// Вейвлет Хаара
double x, f;
cout<<"\n x= "; cin>>x;
f =  ((x<0) || (x>=1)) ? 0 : (x<0.5) ? 1 : -1;
cout<<"\n f = "<<f<<"\n";
```

Инструкция ***switch***

В C/C++ оператор выбора реализован в виде

```
switch (управляющее выражение) {
  case постоянная1:
    последовательность операторов
    break;
  case постоянная2:
    последовательность операторов
    break;
  case постоянная3:
    последовательность операторов
    break;
  default:
    последовательность операторов;
}
```

Значение ***управляющего выражения*** последовательно сравнивается со значениями в списке ***постоянная1, постоянная2,*** .... При обнаружении совпадения выполняется ***последовательность операторов*** соответствующей ветки **case**. Ветка **default** выполняется в том случае, когда значение ***управляющего выражения*** не совпало ни с одной постоянной. Оператор `switch` может не содержать ветки `default`.

Отметим важность использования оператора `break` в каждой ветке оператора выбора. Когда встречается оператор `break`, выполнение оператора `switch` прекращается и управление передается на следующий за `switch` оператор программы. Если оператор `break` отсутствует, то, после выполнения последовательности операторов выбранной ветки, продолжится выполнение операторов следующих веток до тех пор, пока не встретится `break` или не закончится оператор `switch`.

Значение ***управляющего выражения*** должно быть целого или символьного типа.

Пример. Подсчет количества гласных в англоязычном тексте, введенном с клавиатуры (См. Липпман С., Ложойе Ж., Му Б. *Язык программирования C++. Вводный курс*. – М.: ООО "И.Д. Вильямс", 2007).

```
#include <iostream>
int main()
{
char ch;
int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;
while ( cin >> ch )
switch ( ch ) {
  case 'a': ++aCnt;break;
  case 'e':++eCnt;break;
  case 'i':++iCnt;break;
  case 'o':++oCnt;break;
  case 'u':++uCnt;break;
}
cout << "Встретилась a: \t" << aCnt << '\n'
<< "Встретилась e: \t" << eCnt << '\n'
<< "Встретилась i: \t" << iCnt << '\n'
<< "Встретилась o: \t" << oCnt << '\n'
<< "Встретилась u: \t" << uCnt << '\n';
}
```

Пример. Модификация предыдущего примера – вычисляется общее количество гласных.

```
int aeiouCnt = 0;
// ...
switch ( ch )
{
// любой из символов a,e,i,o,u
// увеличит значение aeiouCnt
case 'a':
case 'e':
case 'i':
case 'o':
```

```
case 'u':
++ aeiouCnt;
    break;
}
```

Пример. В C/C++ нет диапазонов, как, например, в языке Pascal. Возможно, здесь проще было бы использовать `if`.

```
int day;
cin>>day;
switch (day)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5: cout<<"\n Workday \n"; break;
    case 6:
    case 7: cout<<"\n day off \n"; break;
}
```

Пример. Стандартный пример оператора выбора – калькулятор.

```
double op1,op2; char ch;
cout<<"\nEnter: operand operation operand\n";
cin>>op1>>ch>>op2;
switch(ch){
    case '+': cout<<"Result="<<op1+op2<<endl;
    break;
    case '*': cout<<"Result="<<op1*op2<<endl;
    break;
    case '-': cout<<"Result="<<op1-op2<<endl;
    break;
    case '/': cout<<"Result="<<op1/op2<<endl;
    break;
    default: cout<<"Nedopustimo"<<endl;        break;
}
```

На количество веток, которые может содержать оператор выбора, иммется ограничение. Стандарт C89, устанавливает ограничение в 257 блоков `case` в операторе `switch`, а Стандарт C99 увеличил это значение до 1023 блоков `case`.

Допускается создавать ***вложенные операторы выбора***. Операторы case внутреннего и внешнего `switch` могут иметь одинаковые константы.

Инструкция цикла `for`

.Общая форма оператора цикла `for`

for (инициализация; условие продолжения; выражение обновления) инструкция цикла

Схема работы инструкции **for:**

i. вычисление выражения в блоке **инициализация**;
ii. вычисление **условия продолжения** и, если условие истинно (т.е. значение отлично от нуля), выполнение **инструкции цикла**;
iii. вычисление **выражения обновления** и снова шаг ii.

Операторы блока **инициализация** выполняются только один раз, чаще всего в этом блоке задается начальное значение переменной, используемой как счетчик цикла.

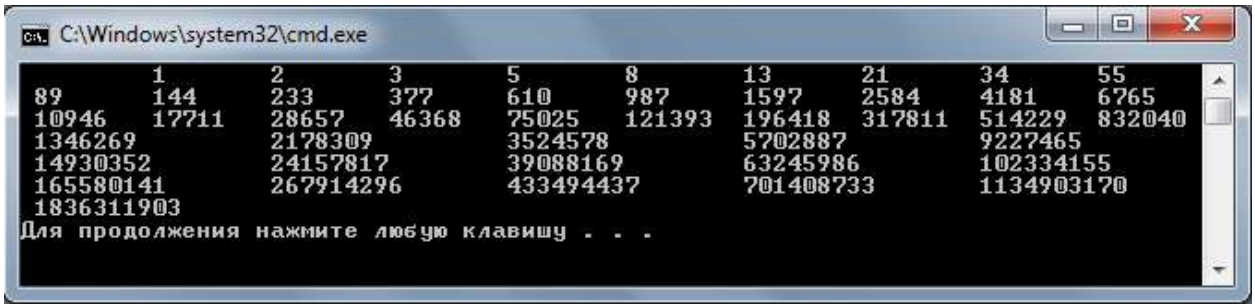
В качестве **условия продолжения** может использоваться любое выражение языка, но, как правило, – это выражение сравнения. Пока это условие истинно, выполняется **инструкция цикла**. Если при первом вычислении **условия продолжения** получаем нулевое значение (ложь) **инструкция цикла** не выполняется ни разу.

После каждого выполнения **инструкции цикла** вычисляется **выражение обновления**. С его помощью можно изменить **условие продолжения** цикла.

Любой блок оператора **for** может отсутствовать.

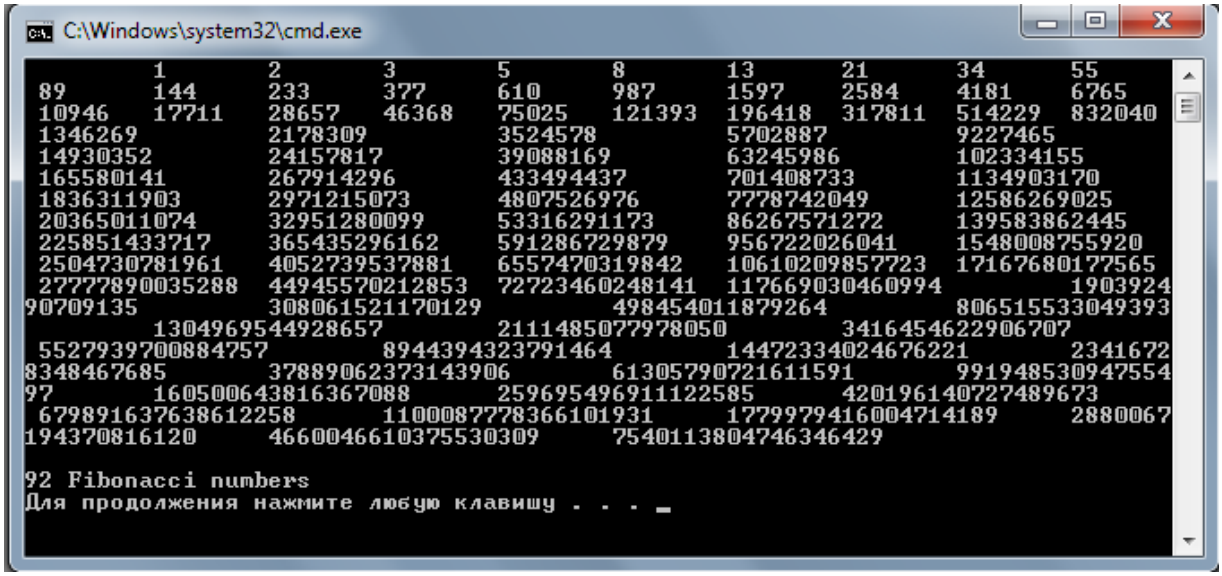
Пример. Вычисление чисел Фибоначчи. Условие окончания цикла – достижение предела хранения целых чисел. В операторе `for` отсутствуют блоки инициализации и обновления.

```
int f,g,h;
f=g=h=1;
for (;(INT_MAX-f)>0;) {
    f=g+h;
    h=g;
    cout<<"\t "<<g; /* печатаем g, последнее f уже
неправильно */
    g=f;
}
```



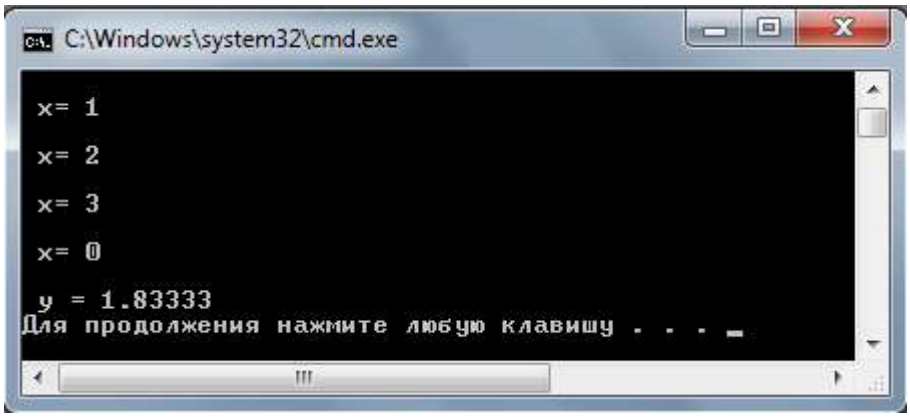
Пример. Те же вычисления, что и в предыдущем примере, но с подсчетом количества полученных чисел Фибоначчи. Кроме того вместо `int` используем `long long int`. В цикле имеется счетчик – переменная `k`, но условие продолжения в данном примере не использует значения счетчика.

```
long long int f,g,h;
int k;
f=g=h=1;
for (k=1;(LLONG_MAX-f)>0;k++) {
     f=g+h;
     h=g;
     cout<<"\t "<<g;
     g=f;
}
cout<<"\n\n"<<k<<" Fibonacci numbers \n";
```



Пример. Выход из цикла – нулевое число при вводе данных. В операторе `for` все управляющие блоки пустые. Выход из цикла производится оператором `break`.

```
int x;
 double y=0;
 for(;;){
      cout<<"\n x= "; cin>>x;
      if (x==0) break;
      y += 1/(double)x;
 }
 cout<<"\n y = "<<y<<'\n';
```



С помощью оператора “,” (запятая) можно включать сразу несколько операторов в любой управляющий блок.

Пример. Все те же числа Фибоначчи. Все операции в одном цикле. В отличие от предыдущих примеров вычисления этих чисел, в теле цикла печатаем `f`, а не `g`. Неправильные значения уже не попадут в тело цикла – после вычисления операторов блока обновления будет проверено условие продолжение и только потом (если условие истинно), будет выполнено тело цикла.

```
// Числа Фибоначчи
int f,g,h;
for (f=g=h=1;(INT_MAX-f)>0;f=g+h,h=g,g=f)
                                        cout<<"\t "<<f;
```

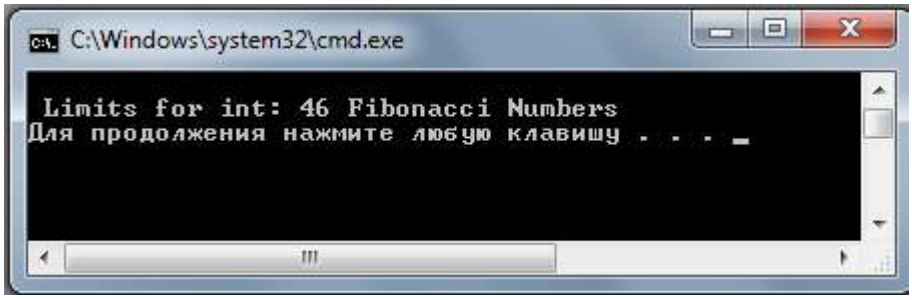
В стандартах C99, C11 языка C и стандартах языка C++ разрешено объявлять переменные в блоке инициализации цикла `for`. Отметим, что в языке C, ориентированном на стандарт C89, этого нет. Объявленные в `for` переменные являются локальными, и их областью видимости является только тело цикла.

Пример. Тот же пример, но объявления переменных помещены в блок инициализации.

```
for (int f=1, g=1, h=1;(INT_MAX-f)>0;f=g+h,h=g,g=f)
                                        cout<<"\t "<<f;
```

Пример. Еще одна иллюстрация оператора цикла – тело цикла пустое.

```
// Числа Фибоначчи
int k=1;
for (int f=1,g=1,h=1;(INT_MAX-f)>0;f=g+h,h=g,g=f, k++);
cout<<"\nLimits for int: "<<k<<" Fibonacci Numbers\n";
```



Замечание. Поместить переменную **k** в блок инициализации цикла не удастся – переменная нужна после выхода из цикла, где уже не видны переменные, объявленные в цикле (`f`, `g`, `h`).

Цикл с предусловием

Цикл с предусловием (т.е. сначала проверка условия и только затем выполнение) реализован инструкцией

`while (условие продолжения) оператор`

Схема выполнения цикла следующая

i. Вычисляется **`условие продолжения`**.
ii. Пока **`условие продолжения`** имеет значение, отличное от 0 (т.е. является истинным), выполняется **`оператор`**.

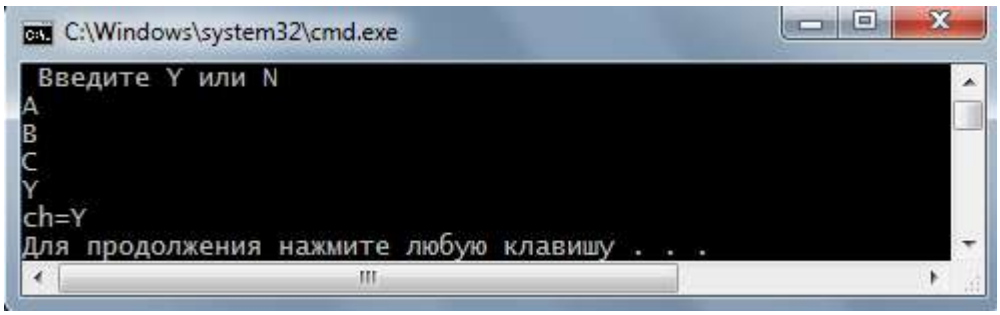
Если при первом вычислении **`условия продолжения`** получаем нулевое значение, **`оператор`** не выполняется ни разу. В цикле с предусловием **`оператор`** может быть также пустым оператором или составным оператором.

Пример. Чтение символов, введенных с клавиатуры, пока не встретится символ “**.**” (точка).

```
char ch = '\0';  // начальное значение
while(ch != '.') ch = getchar();
```

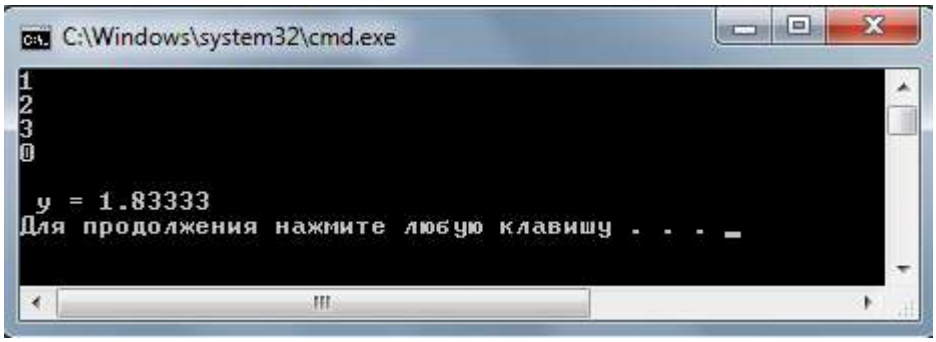
Пример. Цикл выполняется до тех пор, пока не будет введен один из символов `'Y'` или `'N'`. С помощью такого цикла можно сформировать блок ответа на вопрос. Особенность этого цикла – пустой оператор в теле цикла.

```
setlocale(LC_CTYPE, "rus"); // русификация консоли
char ch;
cout<<"\n Введите Y или N \n";
while(((ch = getchar())!='Y') && (ch!='N')); /* пока не
Y или N */
cout<<"ch="<<ch <<'\n';
```



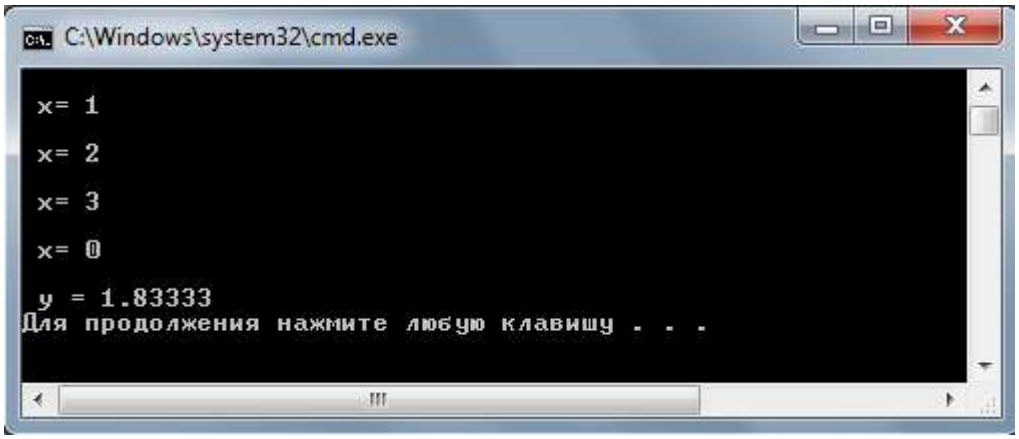
Пример. Модификация примера из раздела, посвященному циклу `for`, – вычисление суммы, пока при вводе не встретится нуль.

```
int x;
double y=0;
while (cin>>x, x!=0) y += 1/(double)x;
cout<<"\n y = "<<y<<'\n';
```



С помощью оператора “,” (запятая) можем добавить и отображение подсказки при вводе

```
int x;
double y=0;
while (cout<<"\n x= ", cin>>x, x!=0) y += 1/(double)x;
cout<<"\n y = "<<y<<'\n';
```



Пример. Чтение всей информации из файла.

```
int i,t;
int x[10];
f=fopen("c:\\tmp\\dannye.txt","r");
i = 0;
```

```
  while (!feof(f))
     {
       fscanf(f,"%d",&t);
       x[i]=t;
       cout<<"\n x="<<x[i++];
      }
   fclose(f);
```

Цикл с постусловием

Цикл с постусловием реализует вариант цикла, в котором сначала выполняется тело цикла и только затем вычисляется условие продолжения цикла. Если значение условия равно нулю (что соответствует false), цикл завершается, если же значение условия отлично от нуля, то снова выполняется тело цикла с последующей проверкой условия. В любом случае, тело цикла выполнится хотя бы один раз. В C/C++ оператор цикла с постусловием имеет вид

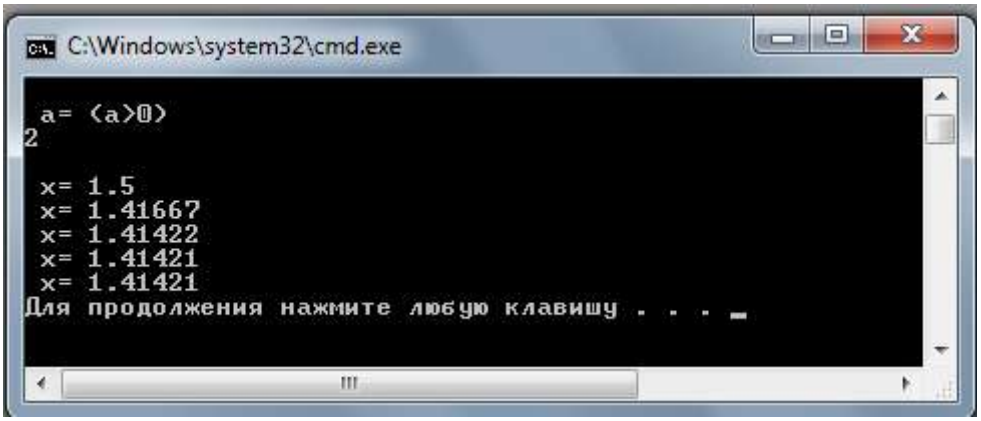
```
do
     оператор
while ( условие продолжения );
```

Условие продолжение и **оператор** подчинены тем же правилам, что и аналогичные блоки уже рассмотренных операторов цикла.

Пример. Хорошо известный из школьного курса математики способ приближенного вычисления квадратного корня из числа.

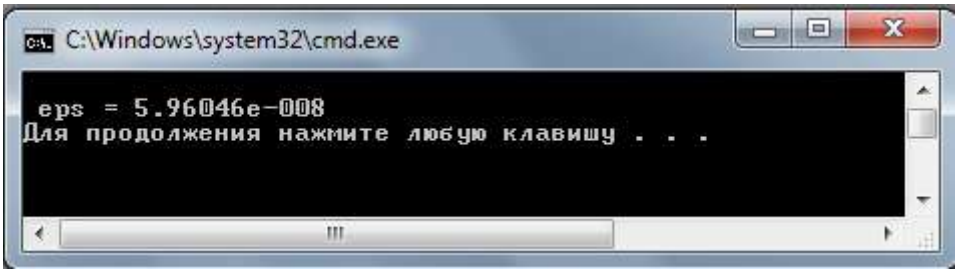
```
// Квадратный корень из числа a

#include <iostream>
#include <cmath>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
double a,x,y,eps,d;
     eps=1.e-6; // точность вычислений
     cout<<"\n a= (a>0) \n"; cin>>a;
     if (a<=0) return 1;
     x=a;
     do{
          y=0.5*(x+a/x);
          d=fabs(x-y);
          x=y;
          cout<<"\n x= "<<x;
     } while (d>eps);
     cout<<'\n';
```

```
    return 0;
}
```



Пример. Вычисление ***машинного эпсилон*** (*machine epsilon*). Название взято из книги Форсайт Дж., Малькольм М., Моулер К. *Машинные методы математических вычислений*. – М.: Мир, 1980, стр. 26. Машинное эпсилон – наименьшее число ε с плавающей точкой, т.ч. $1 \oplus \varepsilon > 1$, где $\oplus$ – плавающее сложение. В книге также приведен алгоритм вычисления этого числа.

```
// машинное эпсилон
float eps1, eps=1;
do{
eps *= 0.5;
eps1 = eps+1;
} while (eps1>1);
cout<<"\n eps = "<<eps<<'\n';
```



Инструкция `break`

С инструкцией `break` уже встречались в разделе, посвященному оператору `switch`.

Инструкция `break` прерывает выполнение операторов цикла `for`, `while`, `do while`, а также оператора выбора `switch`. Программа продолжает работу с оператора, следующего за блоком цикла или инструкции выбора.

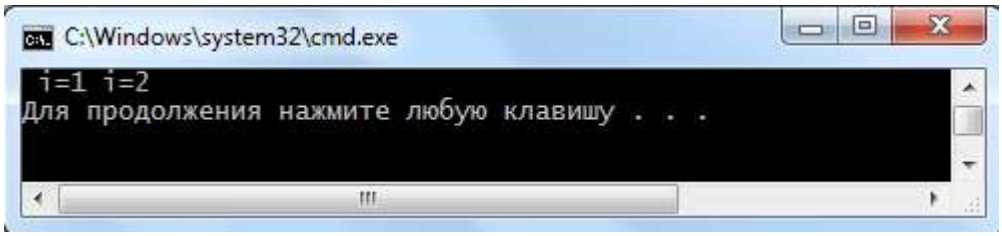
Инструкция `continue`

Инструкция `continue` завершает текущую итерацию цикла и передает управление на вычисление условия, после чего цикл может

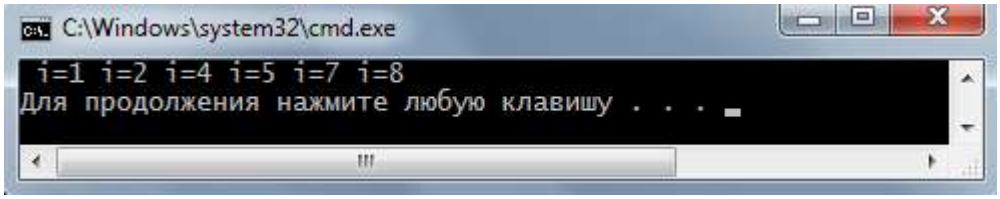
продолжиться. В отличие от инструкции `break`, завершающей выполнение всего цикла, инструкция `continue` завершает выполнение только текущей итерации.

Пример. Операторы `break` и `continue`. Разницу между этими операторами иллюстрирует пример (достаточно убрать комментарии с `break`, а `continue`, наоборот, закомментировать). В приведенном варианте (с `break`) вычисления в цикле прервутся при `i=3`, а в варианте с `continue` цикл выполнится полностью, но будут пропущены значения `i`, равные 3, 6, 9.

```
#include <iostream>
using namespace std;
int main()
{
   int i;
    for (i=1;i<10;i++){
       if((i%3)==0)
              break;
           // continue;
      cout <<" i= "<<i;
    }
    return 0;
}
```



Результаты программы для варианта с `continue`:



Инструкция goto

Инструкция `goto` обеспечивает безусловный переход к другой инструкции внутри той же функции, поэтому современная практика программирования выступает против ее применения.

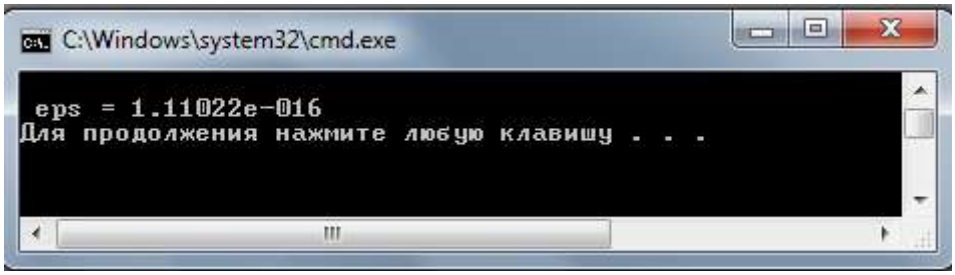
Синтаксис оператора `goto`:

goto метка;

где **метка** – определенный пользователем идентификатор. Метка ставится перед инструкцией, на которую можно перейти с помощью `goto`, и должна заканчиваться двоеточием.

Пример. Вычисление машинного эпсилон. Предложенный код совпадает с приведенным в указанной ранее книге (где код был написан на Фортране, где без `goto` обойтись трудно).

```
// машинное эпсилон
double eps1, eps=1;
label1: eps *= 0.5;
eps1 = eps+1;
if (eps1>1) goto label1;
cout<<"\n eps = "<<eps<<'\n';
```



Оператор `goto` используют редко, более того, стандартной рекомендацией является полное исключение этого оператора из программ на C++ (см., напр., Голуб А. *Правила программирования на Си и Си++*, стр. 128).

Инструкция `return`

Инструкция `return` обеспечивает выход из функции и имеет следующий синтаксис

return *выражение*;

В функциях, имеющих тип `void`, оператор `return` используется без значения. Значение ***выражения*** является возвращаемым значением функции.

Функция может содержать любое количество операторов `return`.

Массивы

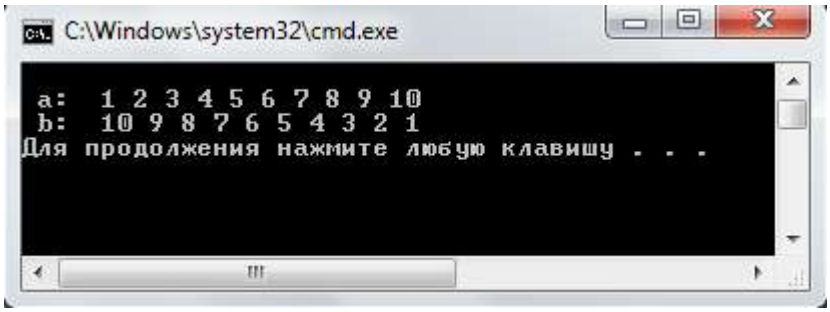
Массив – это набор данных одного типа и объединенных общим именем. Массив создается оператором объявления.

Type arrayName[arraySize];

Объявление массива содержит 3 части. В первой части указан тип элементов массива, затем – имя массива и в третьей части объявления в квадратных скобках указано количество элементов массива.

Пример. Создаются два массива `a` и `b`. Массив `b` состоит из тех же данных, что и `a`, но записанных в обратном порядке.

```
const int N=10;
int a[N];
int b[N];
for (int i=0;i<N;i++) a[i] = i+1;
for (int i=0;i<N;i++) b[i] = a[N-i-1];
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<N;i++) cout<<' '<<b[i];
```

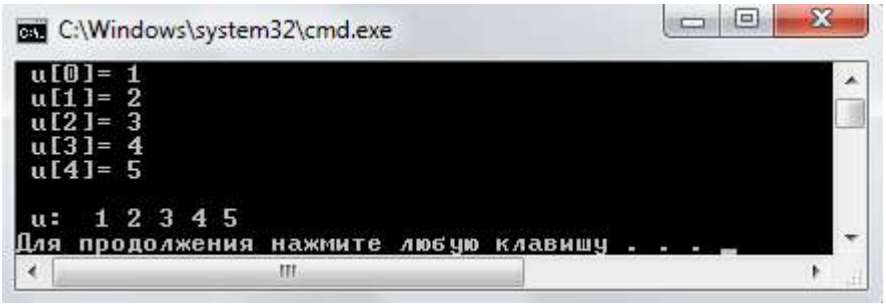


Выражение `arraySize`, задающее количество элементов в объявлении массива, должно быть константой (как в примере.), или выражением, значение которого известно во время компиляции. Если необходимо создать массив, число элементов которого будет установлено только во время выполнения программы, то должны использоваться либо функция `malloc()` или оператор `new`, – это будет показано в разделе “Динамические массивы”.

Доступ к элементам массива осуществляется с помощью индекса: `a[0]`, `a[1]`, `a[2]` и т.д. Индекс помещается в квадратных скобках после имени массива. Индекс первого элемента любого массива равен нулю. В качестве индекса можно применять любое числовое значение, – оно будет преобразовано в целое число.

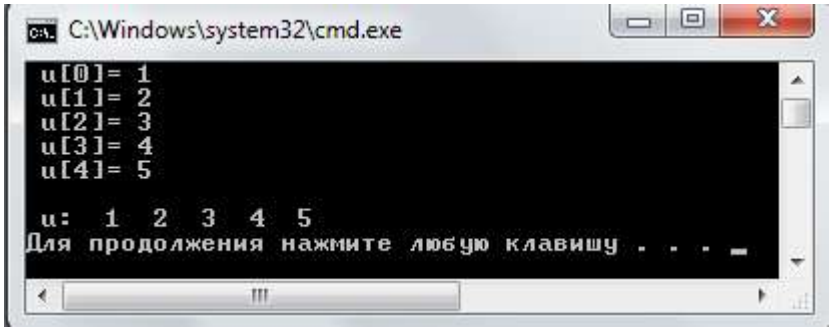
Пример. Заполнение массива числами, введенными с клавиатуры.

```
// Ввод чисел с клавиатуры
const int N=5;
int u[N];
for (int i=0; i<N;i++) {
    cout<<" u["<<i<<"]= "; cin>>u[i];
}
cout<<"\n u: ";
for (int i=0;i<N;i++) cout<<' '<<u[i];
```



Пример. Тот же пример ввода элементов массива с клавиатуры, но с использованием функций только языка С.

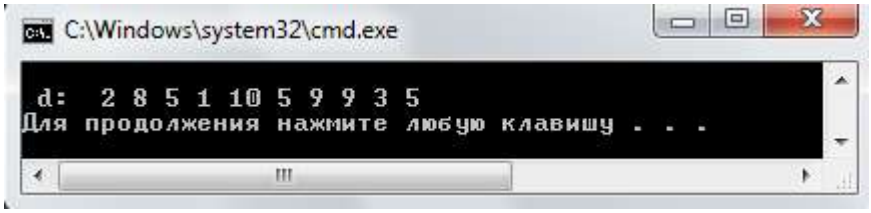
```
// Ввод чисел с клавиатуры
#include <stdio.h>
#define N 5

int main()
{
    int u[N]; int i;
    for (i=0; i<N;i++) {
        printf(" u[%d]= ",i);
        scanf("%d",&u[i]);
    }
    printf("\n u: ");
    for (i=0;i<N;i++) printf(" %d ",u[i]);
    return 0;
}
```



Пример. Заполнение массива случайными числами. Используем функцию `rand()`, генерирующую целые числа от `0` до `RAND_MAX` (константа, определенная в `stdlib.h`). Для изменения диапазона используем операцию получения остатка (“`%`”), в примере выбран диапазон от `1` до `10`.

```
#include <iostream>
#include <stdlib.h> // объявление rand()
using namespace std;

const int N=10;
```

```
int main()
{
     int d[N];
     for (int i=0; i<N;i++) d[i] = 1+rand()%10;
     cout<<"\n d: ";
     for (int i=0;i<N;i++) cout<<' '<<d[i];
     cout<<"\n";
     return 0;
}
```

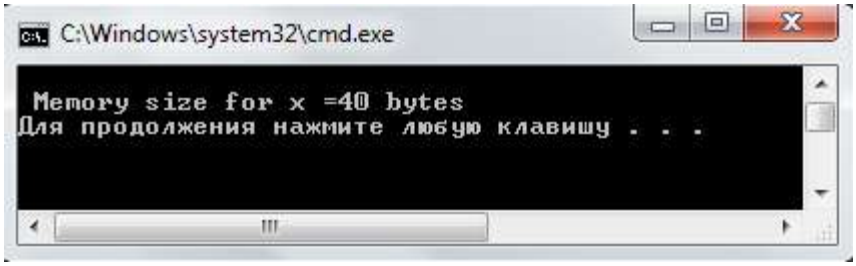


Объем памяти, занятой элементами массива, можно определить с помощью функции `sizeof()` по формуле

количество_байтов = sizeof(тип) × размер_массива

Пример. Вычисление объема памяти, занятой массивом.

```
const int N=10;
float x[N];
int size_x = sizeof(float) * N;
cout<<"\n Memory size for x ="<<size_x<<" bytes\n";
```

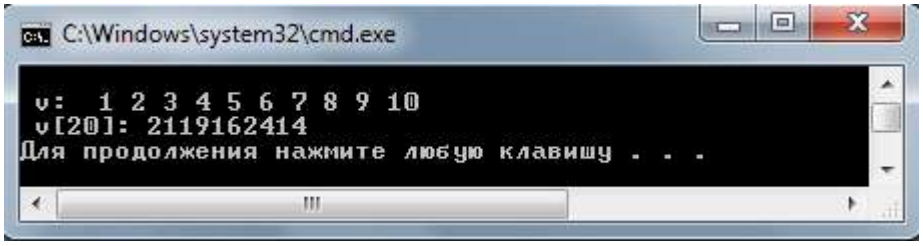


Важная особенность языка C/C++ – отсутствует ***контроль соблюдения границ*** массива. Такая проверка – обязанность программиста.

Пример. Несоблюдение границ массива. В программе сделана попытка выполнить обращение к массиву, указав в качестве индекса значение, превышающее его границу. Программа успешно прошла компиляцию, и на допущенную в программе ошибку не было указано. Элементу присвоено значение соседней с массивом области памяти.

```
// Несоблюдение границ
const int N=10;
int v[N];
cout<<"\n v: ";
for (int i=0;i<N;i++) {
```

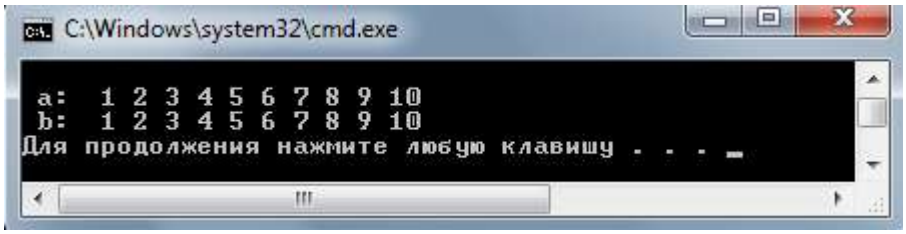
```
    v[i] = i+1;
    cout<<' '<<v[i];
}
// индекс за границей массива:
cout<<"\n v[20]: "<<v[N+10]<<"\n";
```



Нельзя присваивать массиву другой массив, указав лишь их имена, также нельзя сравнивать массивы, складывать и т. д. Все операции разрешено выполнять только с отдельными элементами массива.

Пример. Присвоение значений одного массива другому.

```
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) a[i] = i+1;
// массиву b присваиваются значения массива a
for (int i=0;i<N;i++) b[i] = a[i];
```



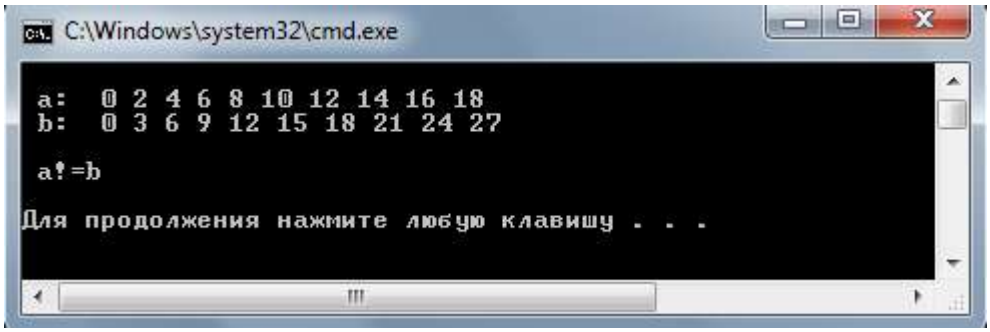
Пример. Сравнение двух массивов на равенство элементов.

```
// Сравнение массивов
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) a[i] = b[i] = i+1;
// проверяем равны ли массивы a и b
bool p=1;
for (int i=0;i<N;i++) p = (b[i] == a[i]);
if (p) cout<<"\n a=b \n";
else  cout<<"\n a!=b \n";
```

Пример. Можно улучшить код и избежать лишних проверок. Если условие продолжения цикла заменить условием **(i<N) && p**, выход из цикла будет выполнен при первом же несовпадении элементов массивов.

```
// Сравнение массивов
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) {
```

```
    a[i] = i*2; b[i] = i*3;
}
// проверяем, равны ли массивы a и b
bool p=1;
for (int i=0;(i<N) && p;i++) p = (b[i] == a[i]);

cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<N;i++) cout<<' '<<b[i];

cout<<"\n";
if (p) cout<<"\n a=b \n";
else  cout<<"\n a!=b \n";
```

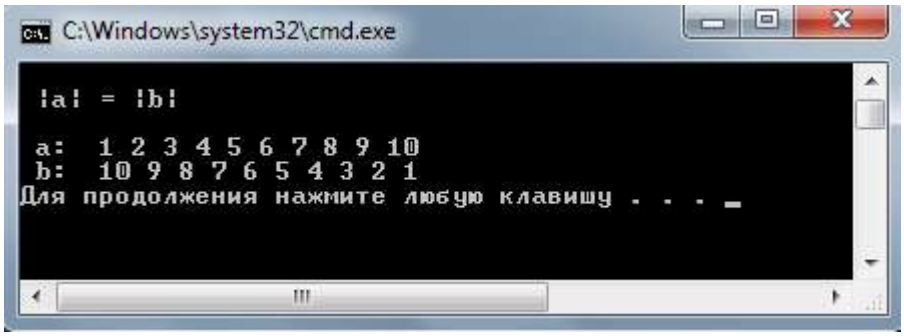


Сложнее проверить содержат ли массивы одни и те же данные, т.е. массивы совпадут после перестановки элементов в одном из них.

Пример. Совпадение данных.

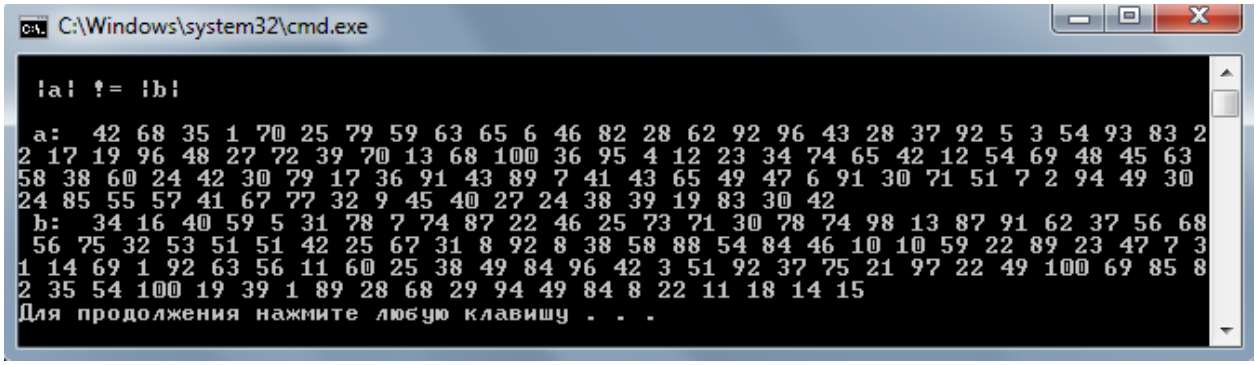
```
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) {
    a[i] = i+1; b[i] = N-i;
}
// проверяем, все ли элементы массива a есть в b
bool q=1; // проверочный флаг
for (int i=0;(i<N) && q;i++) {
    q = 0;
    for (int j=0;j<N;j++)
        if (a[i]==b[j]) { // найдено совпадение
            q=1;
            break; // к следующему индексу i
        }
}

if (q) cout<<"\n |a| = |b| \n";
else  cout<<"\n |a| != |b| \n";
```

```
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<N;i++) cout<<' '<<b[i];
```



Пример. Тот же пример, но в качестве данных берутся наборы случайных чисел из одного диапазона. В коде изменен только блок инициализации массивов.

```
#include <stdlib.h> // объявление rand()
    … … … … … … … … …
// Совпадение случайных данных
const int N=100;
int a[N], b[N];
for (int i=0;i<N;i++) a[i] = 1+rand()%100;
for (int i=0;i<N;i++) b[i] = 1+rand()%100;
    … … … … … … … … …
```

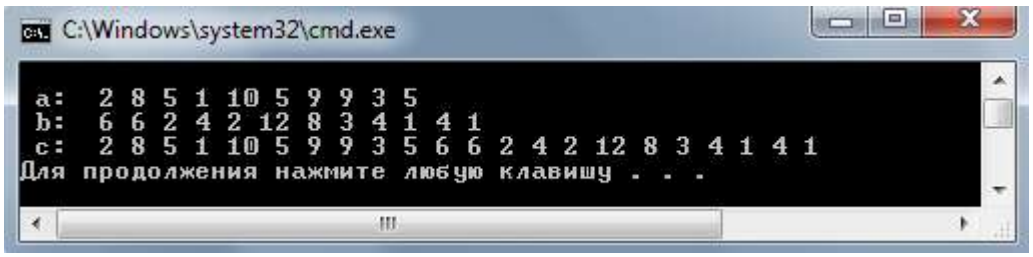


Распространенной операцией с массивами является слияние двух и более массивов в один результирующий массив.

Пример. Слияние двух массивов.

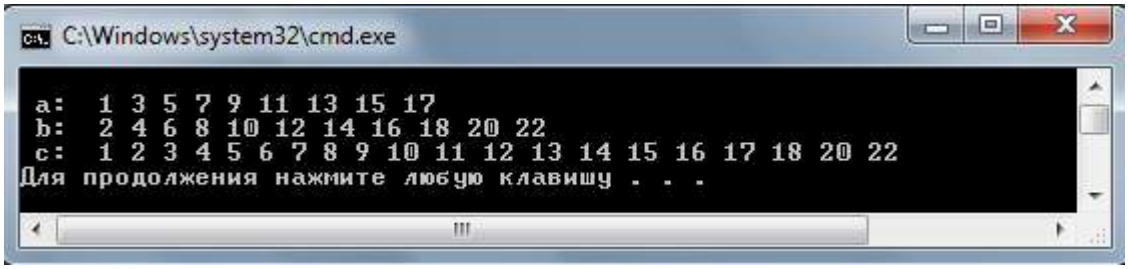
```
// Слияние двух массивов
const int N=10;
const int M=12;
int a[N], b[M], c[N+M];
// инициализация массивов
for (int i=0;i<N;i++)  a[i] = 1+rand()%10;
```

```
for (int i=0;i<M;i++)  b[i] =  1+rand()%12;
// образуем массив присоединением массива b к a
int k=0;
for (int i=0;i<N;i++) c[k++] = a[i];
for (int i=0;i<M;i++) c[k++] = b[i];
// вывод результатов
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<M;i++) cout<<' '<<b[i];
cout<<"\n c: ";
for (int i=0;i<N+M;i++) cout<<' '<<c[i];
cout<<"\n";
```



Пример. Слияние двух упорядоченных массивов a[0]≤a[1]≤…≤a[N] и b[0]≤b[1]≤…≤b[N].

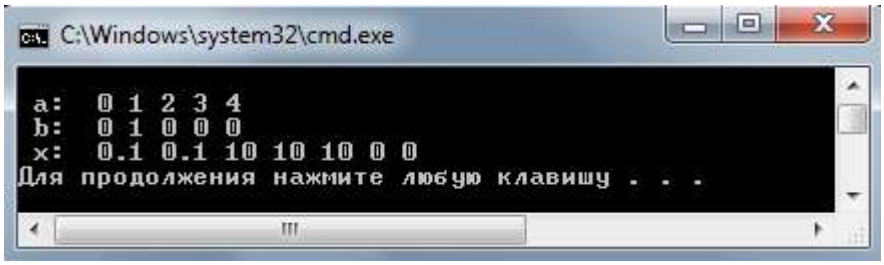
```
// Слияние двух упорядоченных массивов
const int N=9;
const int M=11;
int a[N], b[M], c[N+M];
// инициализация массивов
for (int i=0;i<N;i++)  a[i] = 2*i+1;
for (int i=0;i<M;i++)  b[i] =  2*(i+1);
int i,j,k;
i=j=k=0;
while ((i<N) && (j<M))
    if (a[i]<b[j]) c[k++] = a[i++];
    else c[k++] = b[j++];
// остаток массива a:
while (i<N) c[k++] = a[i++];
// остаток массива b:
while (j<M) c[k++] = b[j++];
```



При объявлении массива можно использовать ***инициализаторы***. Список инициализирующих значений состоит из констант, разделенных запятыми. Весь список заключен в фигурные скобки.

Пример. Инициализация массивов целых чисел и массива чисел с плавающей точкой.

```
// инициализация
int a[5]={0,1,2,3,4};
int b[5]={0,1};
double x[7]={0.1,1.e-1,10,1.e+1,1e1};
```



Количество значений в инициализаторе не должно быть больше размера массива. Если значений в инициализаторе меньше, остальным элементам массива присваивается `0`. Каждая константа должна иметь тип, совместимый с типом массива.

Массивы типа `char` можно инициализировать строковыми константами.

Пример. Инициализация массива типа `char`. Поскольку строки заканчиваются символом `'\0'`, граница массива должна быть на единицу больше длины слова. Поэтому для хранения слова `"february"`, состоящего из 8 символов, потребовался массив из 9 элементов.

```
// инициализация char
char login[8] = {'j','a','n','u','a','r','y','\0'};
char password[9]="february";
cout<<'\n'<<login;
cout<<'\n'<<password;
```

Инициализацию в объявлении массива удобно использовать при реализации численных методов.

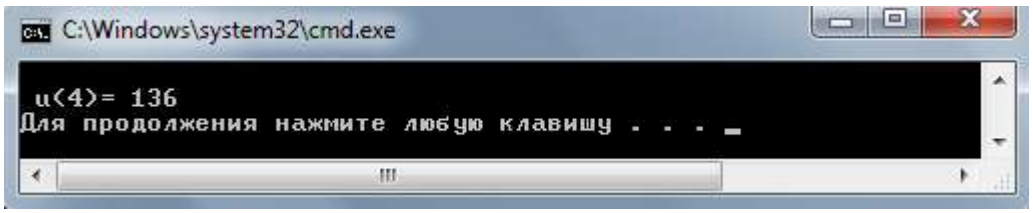
Пример. Вычисление значения многочлена

$u(x) = x^4 - 3x^3 + 6x^2 - 10x + 16$ при $x = 4.$

методом Горнера.

```
#include "stdafx.h"
#include <iostream>
#include <cmath>
using namespace std;
```

```
const int n=4;

int main(int argc, _TCHAR* argv[])
{
     double  a[n+1]={16.0,-10.0,6.0,-3.0,1.0};
     double u,x;
     int k;
     // Вычисление значения многочлена
     for (x = 4,u = a[n],k=n-1;k>=0;k--) u = u*x + a[k];
     cout<<"\n u("<<x<<")= "<<u<<"\n";
          return 0;
}
```



Если при объявлении массива используется инициализатор, можно не указывать размер массива – подсчет будет произведен компилятором. Масивы, объявленные без указания границы изменения индекса, называют ***безразмерными***.

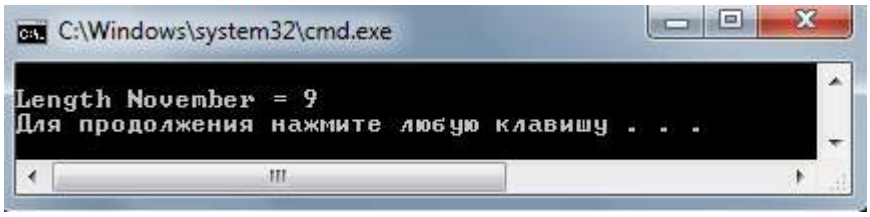
Пример.

```
// безразмерные массивы
int a[]={1,2,3,4,5};
for (int i=0;i<5;i++) cout<<' '<<a[i];
char s[]="November";
cout<<'\n'<<s;
```

Использование безразмерных символьных массивов значительно упрощает их инициализацию, т.к. не нужно подсчитывать количество символов, – проще написать `char s[]="November"`, чем `char s[9]= "November"`. При обработке строк, как правило, знать границу символьного массива не нужно – каждая строка заканчивается символом `'\0'`, что и используют при обработке строк. В следующем примере показано, как границу можно найти.

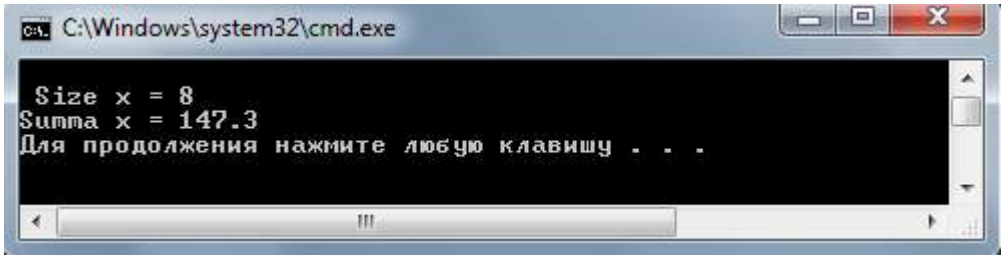
Пример. Вычисление границы безразмерного символьного массива.

```
// безразмерные символьные массивы
char s[]="November";
int l_s=0; // l_s - длина строки s
while (s[l_s++] !='\0');
cout<<"\nLength "<<s<<" = "<<l_s<<"\n";
```



Пример. Вычисление суммы элементов безразмерного числового массива. Количество элементов массива найдено с помощью функции `sizeof()`.

```
// безразмерные числовые массивы
double x[]={1.0,2,3.0,4.,5e1,6.6,0.7,80};
int l_x = sizeof(x)/sizeof(double); /* количество
элементов массива x */
cout<<"\n Size x = "<<l_x;
double s_x=0;
for (int i=0;i<l_x;i++) s_x += x[i];
cout<<"\nSumma x = "<<s_x<<"\n";
```



Строки

Единственный вид строки в языке С – это массив типа `char`, содержащий в качестве последнего элемента символ `'\0'` (***нулевой терминатор, нулевой символ***). Вместо символа `'\0'` можно использовать число `0`.

Пример. Объявление и инициализация строк в С.

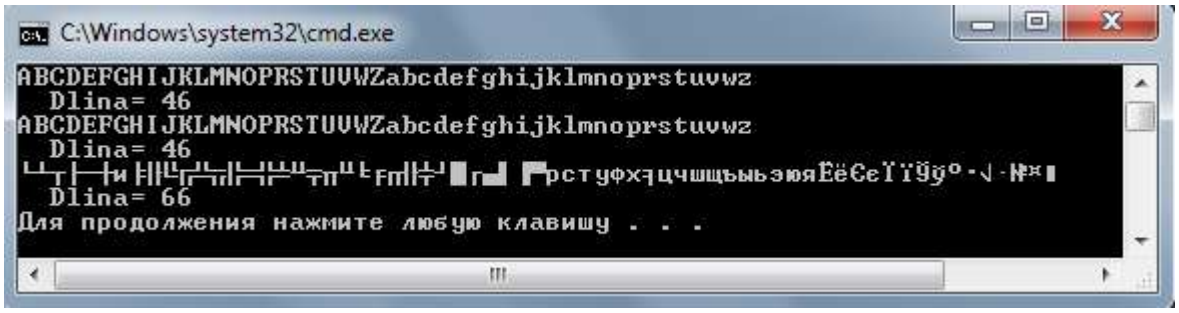
```
char month[4]={'M','a','y','\0'};
char Month[4]={'M','a','y',0};
char password[9]="february";
char fio[4]="FIO";
char Fio[4]={'F','I','O','\0'};//fio==Fio
char s[]="November";
```

Пример. Объявление и инициализация строк, содержащих символы кириллицы – на каждый символ в строке нужно 2 байта.

```
char fio_r[7]="ФИО";
char Fio_r[]={'Ф','И','О','\0'};
char month_r[4]={'М','а','й','\0'};
char Month_r[7]="Май";
```

Пример. Объявление и инициализация длинных строк. Строковые константы, разделенные пробелом, рассматриваются как одна строка. Об этом нужно вспомнить при записи строки, не умещающейся на экране.

```
// Длинные строки
char str_E1[47]=
  "ABCDEFGHIJKLMNOPRSTUVWZabcdefghijklmnoprstuvwz";
char str_E2[47]="ABCDEFGHIJKLMNOPRSTUVWZ"
                        "abcdefghijklmnoprstuvwz";
// - строки str_E1 и str_E2 совпадают
cout<<str_E1<<"\n  Dlina= "<<strlen(str_E1)<<"\n";
cout<<str_E2<<"\n  Dlina= "<<strlen(str_E2)<<"\n";
char str_R[133]="АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
                "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
cout<<str_R<<"\n  Dlina= "<<strlen(str_R)<<"\n";
```



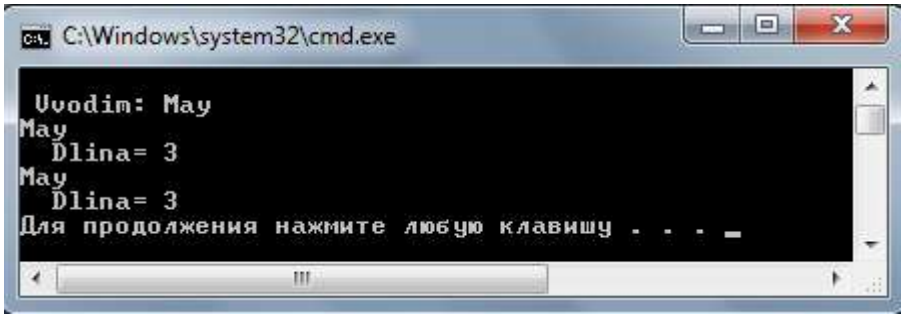
Замечание. Символы кириллицы не отобразились (вместо них значки, – их еще называют “козябликами”) – как исправить положение, показано в одном из следующих примеров. В приложении приведены способы русификации ввода/вывода консольных приложений.

Поскольку строка символов является массивом, нельзя присвоить одну строку другой – необходимо выполнить операцию присваивания для каждого элемента. Точно так же нельзя сравнивать строки. Для выполнения этих и других операций со строками можно использовать встроенные функции, объявленные в файле `string.h`. В частности, присваивание символьных массивов реализуется функцией `strcpy(s1,s2)`, сравнение – функцией `strcmp(s1,s2)`, важнейшая для строк операция склеивания (конкатенация) выполняется функцией `strcat(s1,s2)`, для вычисления длины строки введена функция `strlen(s)`. Поиск в содержимом строки можно выполнить с помощью функций `strchr(s,ch)` и `strstr(s1,s2)`. Использование указанных функций проиллюстрировано дальнейшими примерами.

Пример. Ввод строки с клавиатуры. Длина введенной строки вычисляется с помощью `strlen()`, затем выполняется копирование с помощью функции `strcpy()`.

```
char month[10], temp[10];;
cout<<"\n Vvodim: " ; cin>>month;
```

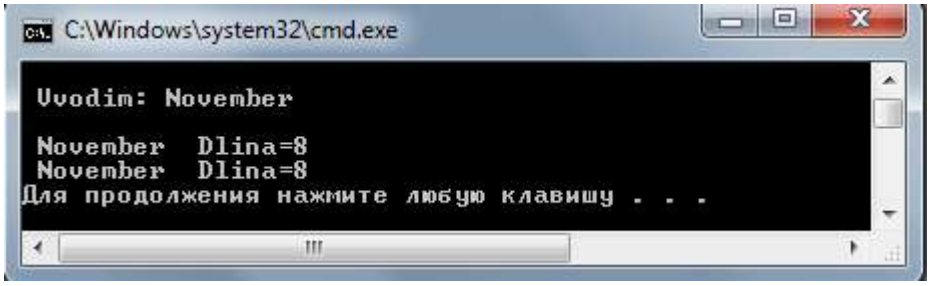
```
cout<<month<<"\n  Dlina= "<<strlen(month)<<"\n";
strcpy(temp,month);// копируем month в temp
cout<<temp<<"\n  Dlina= "<<strlen(temp)<<"\n";
```



Замечание. Функция `strlen()` подсчитывает количество символов в строке без учета символа `'\0'`.

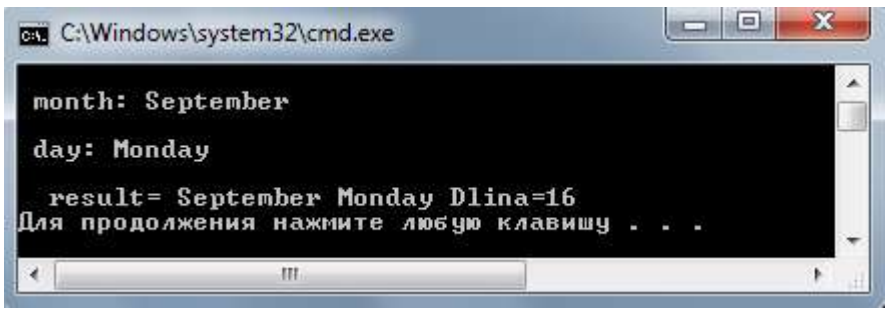
Пример. Ввод строки с клавиатуры средствами языка С.

```
char month[10], temp[10];
printf("\n Vvodim: "); scanf("%s",month);
printf("\n %s  Dlina=%d",month,strlen(month));
strcpy(temp,month);
printf("\n %s  Dlina=%d\n",temp,strlen(temp));
```



Пример. Объединение введенных строк с помощью `strcat()`.

```
// Конкатенация строк
char month[10], day[10], result[21];
cout<<"\n month: " ; cin>>month;
cout<<"\n day: " ; cin>>day;
strcpy(result,month);
strcat(result," ");// добавили пробел
strcat(result,day);
cout<<"\n  result= "<<result<<"
Dlina="<<strlen(result)<<"\n";
```

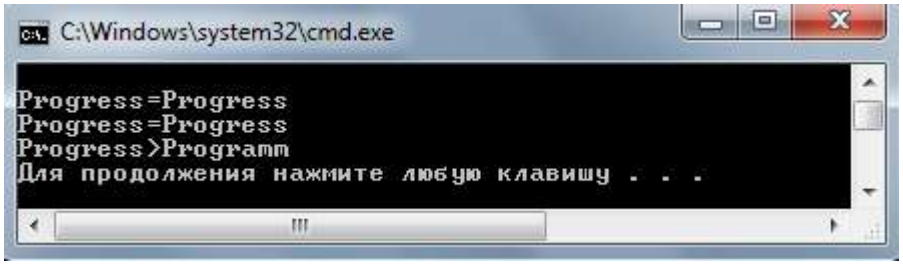


При вводе могут возникнуть ошибки, если количество введенных символов окажется больше, чем выделено памяти для строки. Например, если в программе предыдущего примера ввести "September-" (строка заканчивается символом '-'), то будет сообщено об ошибке, поскольку для хранения month выделено 10 байт, а введенные данные, с учетом нулевого символа, требуют 11 байт. Отметим, что начальные и терминальные пробелы при вводе игнорируются и не вызовут проблем.

Аналогично, недостаток выделенной памяти может привести к ошибке и при выполнении других операций со строками.

Пример. Сравнение строк с помощью функции strcmp(s1,s2). Сравнение производится посимвольно от начала строк, сравниваются коды символов. Функция возвращает 0, если строки s1 и s2 совпадают, отрицательное значение, если s1<s2 и положительное значение, если s1>s2. При сравнении учитывается регистр.

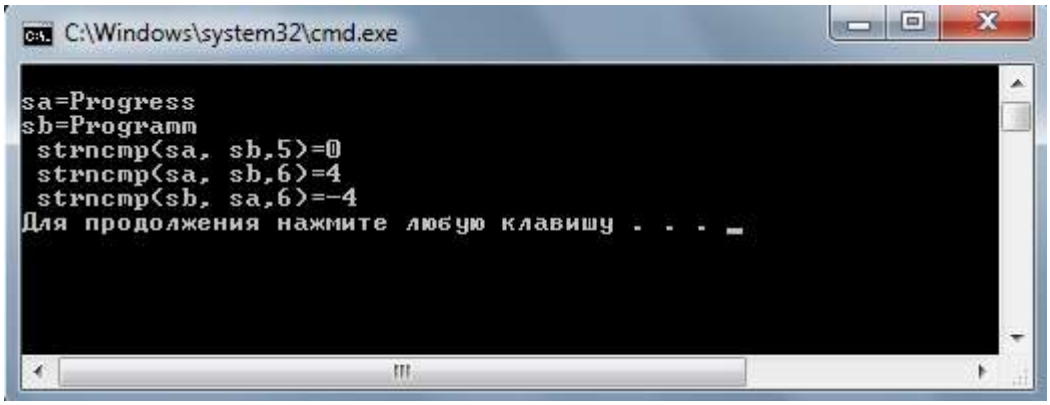
```
// Сравнение строк
char s1[]="Progress";
char s2[]="Progress";
char s3[]="Programm";
// Проверка на равенство строк:
if(strcmp(s1, s2)==0) cout<<"\n"<<s1<<"="<<s2;
// или, что гораздо лучше:
if(!strcmp(s1, s2)) /* отрицание '!', т.к. при
равенстве результат = 0 */
    cout<<"\n"<<s1<<"="<<s2;
// кто раньше в алфавитном порядке:
int p=strcmp(s2, s3);
if(!p) cout<<"\n s2=s3";
else if(p>0) cout<<"\n"<<s2<<">"<<s3;
     else cout<<"\n"<<s2<<"<"<<s3;
```



Замечание. При повторном использовании результата одной и той же функции это значение сохраняют в локальной переменной (в примере – переменная p). Это важное правило позволяет сделать код эффективнее (в книге Голуб А. *Правила программирования на Си и Си++* оно обозначено как “Избегайте дублирования усилий”).

Пример. Сравнение строк с помощью функции `strncmp(s1,s2,n)`. Сравниваются только `n` начальных символов строк.

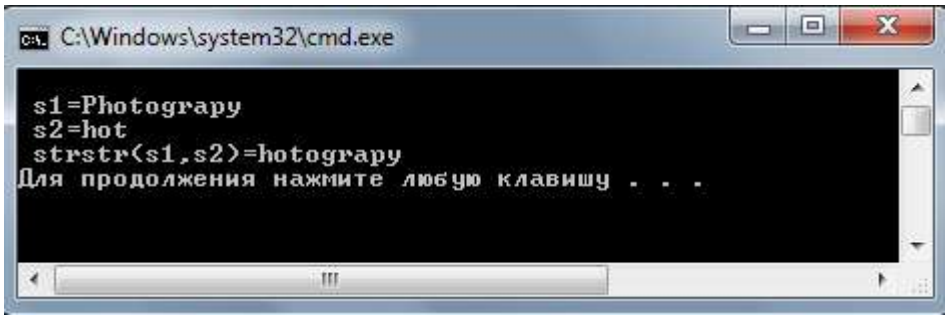
```
// Сравнение начальных отрезков строк
char sa[]="Progress";
char sb[]="Programm";
cout<<"\nsa="<<sa;
cout<<"\nsb="<<sb;
cout<<"\n strncmp(sa, sb,5)="<<strncmp(sa, sb,5);
cout<<"\n strncmp(sa, sb,6)="<<strncmp(sa, sb,6);
cout<<"\n strncmp(sb, sa,6)="<<strncmp(sb, sa,6);
```



Первые 5 символов сравниваемых строк совпадают, поэтому `strncmp(sa, sb,5)` возвращает `0`. Отрезки из 6 символов уже отличаются, значения функций `strncmp(sa, sb,6)` и `strncmp(sb, sa,6)` будет понятно, если отметить, что код символа `'e'` равен `101`, а код символа `'a'` равен `97`.

Пример. Поиск подстроки с помощью функции `strstr(s1,s2)`. Функция возвращает указатель на позицию первого вхождения строки `s1` в `s2`, или `NULL`, если `s1` не содержит `s2`.

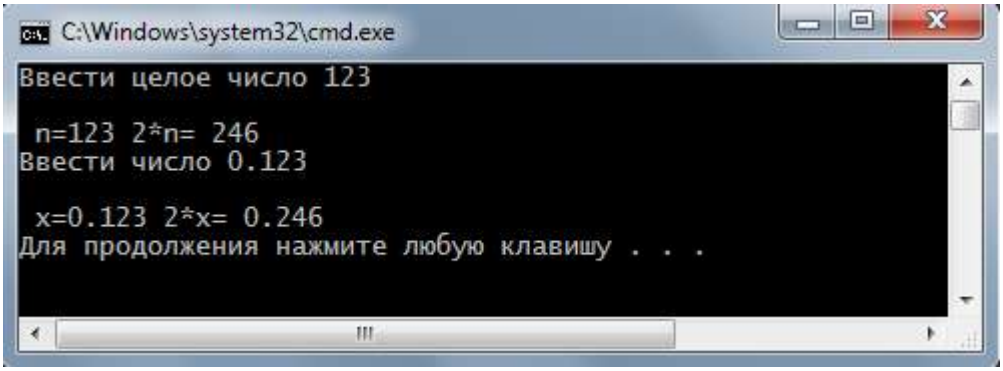
```
// Поиск подстроки
char s1[]="Photograpy";
char s2[]="hot";
char *ps=strstr(s1,s2);
cout<<"\n s1="<<s1;
cout<<"\n s2="<<s2;
if (ps) // ps != NULL, т.е. нашли
    cout<<"\n strstr(s1,s2)="<<ps;
```



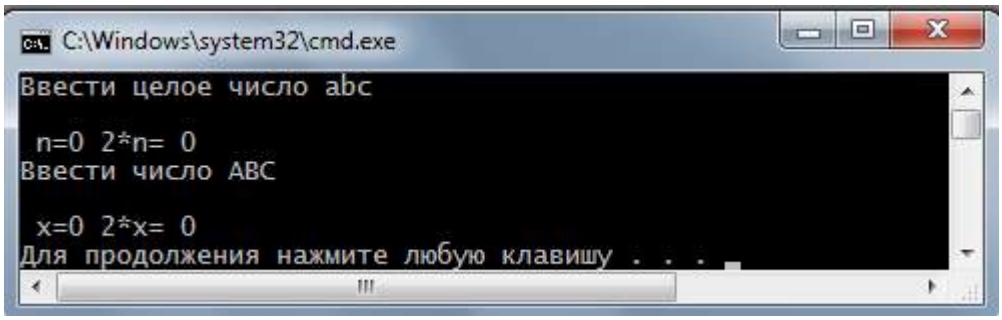
Числовые данные часто вводят в символьном виде, а затем преобразуют в подходящий числовой тип.

Пример. Конвертация символьных массивов в числовые типы. Функция `atoi()` преобразует строку, указанную как параметр, в целое число, а функция `atof()` преобразует строку в число с плавающей точкой. Функции вернут `0`, если преобразование невозможно.

```
#include <iostream>
#include <cstdlib>
#include <clocale>
using namespace std;
int main ()
{
   setlocale(LC_CTYPE, "rus"); // русификация консоли
   char chislo [10];
   cout << " Ввести целое число ";
   cin>>chislo;
// преобразуем символы в целое число:
   int n;
   n=atoi(chislo); // в целое
//
   cout<<"\n n="<<n<<" 2*n= "<<2 * n<<"\n";
   cout << " Ввести число "; cin>>chislo;
// преобразуем символы в число с плав. точкой:
   double x=atof(chislo); // в double
   cout<<"\n x="<<x<<" 2*x= "<<2 * x<<"\n";
   return 0;
}
```

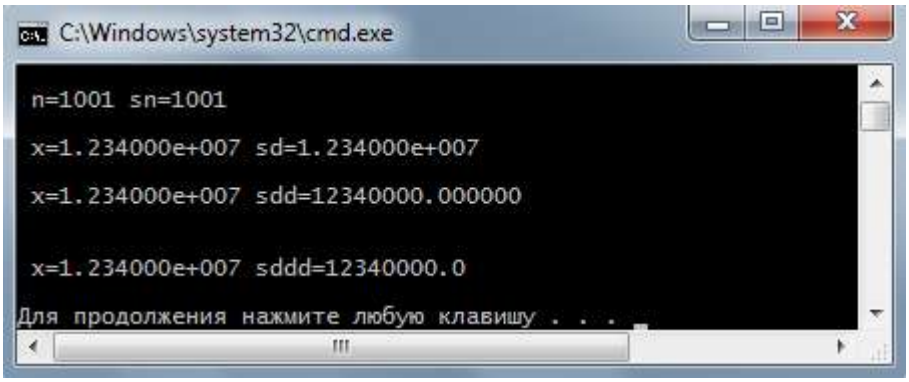


Если преобразование из строки в число невозможно, программа все равно работает, но с числами, равными 0.



Пример. Преобразование числовых данных в строку с помощью функции sprintf(). Эта функция устроена так же, как функция printf(), но вывод производится не на экран, а в строку, укзанную первым параметром.

```
// Конвертация числовых данных в строки
 int n=1001; double x=12340000.00;
 // преобразование целого числа в строку
 char sn[12];
 sprintf(sn,"%d",n);// печать в строку
 printf("\n n=%d sn=%s\n",n,sn);
 // преобразование числа с плав. точкой в строку
 char sd[22];
 sprintf(sd,"%e",x);// печать в строку
 printf("\n x=%e sd=%s\n",x,sd);
 char sdd[22];
 sprintf(sdd,"%f",x);// печать в строку
 printf("\n x=%e sdd=%s\n",x,sdd);
 char sddd[22];
 sprintf(sddd,"%10.1f",x);// печать в строку
 printf("\n x=%e sddd=%s\n\n",x,sddd);
```

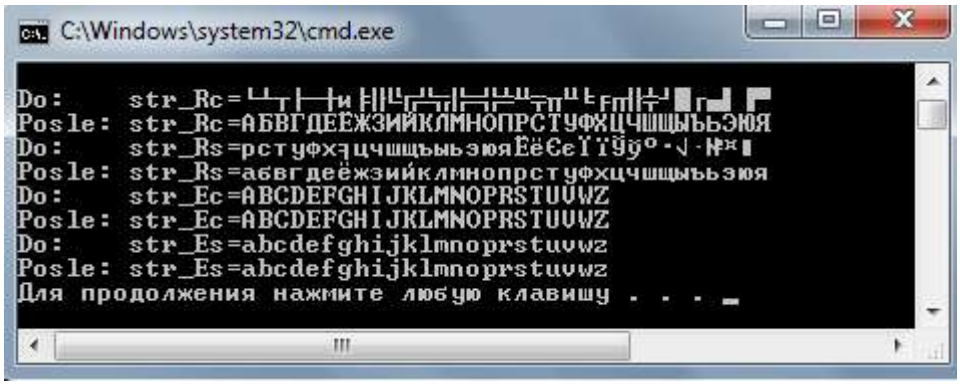


В следующих примерах показано как символьные массивы передаются в функции.

Пример. Символьный массив – параметр функции. Функция ansi2oem() преобразует символы строки, переданной в качестве

параметра, из кодировки cp1251 (ANSI) в кодировку cp866 (OEM-866 или DOS-кодировка).

```
#include <iostream>
using namespace std;
void ansi2oem(char *stroka);
int main()
{
    char str_Rc[]="АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
    cout<<"\nDo:    str_Rc="<<str_Rc;
    ansi2oem(str_Rc);
    cout<<"\nPosle: str_Rc="<<str_Rc;
    char str_Rs[]="абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
    cout<<"\nDo:    str_Rs="<<str_Rs;
    ansi2oem(str_Rs);
    cout<<"\nPosle: str_Rs="<<str_Rs;
    char str_Ec[]="ABCDEFGHIJKLMNOPRSTUVWZ";
    cout<<"\nDo:    str_Ec="<<str_Ec;
    ansi2oem(str_Ec);
    cout<<"\nPosle: str_Ec="<<str_Ec;
    char str_Es[]="abcdefghijklmnoprstuvwz";
    cout<<"\nDo:    str_Es="<<str_Es;
    ansi2oem(str_Es);
    cout<<"\nPosle: str_Es="<<str_Es<<'\n';
    return 0;
}
void ansi2oem(char *stroka)
{
int cnt,i=0;
char ch;
while((ch=stroka[i])!='\0')
 {
    cnt=ch;
    if ((ch>='a') && (ch<='п')) cnt-=64;
     else if ((ch>='p') && (ch<='я')) cnt-=16;
        else if (ch=='ё') cnt=241;
             else if (ch=='Ё') cnt=240;
              else if ((ch>='A') && (ch<='Я')) cnt-=64;
    stroka[i]=cnt; i++;
 }
}
```



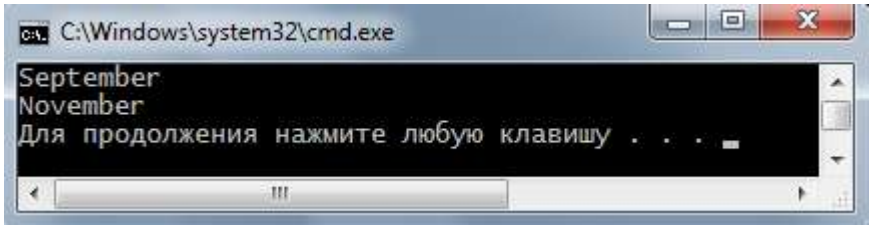
Пример. Функция `copy_string()` содержит 2 параметра в виде символьных массивов. Символы из строки `s_in` копирутся в строку `s_out`.

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

void copy_string (char  [], char  []);

int main ()
{
    char name [30];
    copy_string(name,"Alexandra");
    cout << name<<"\n";
    return 0;
}
void copy_string(char s_out [], char s_in [])
{
  int i=0;
   do {
      s_out[i] = s_in[i];
  } while (s_in[i++] != '\0');
```

Пример. Пожалуй, `cpy()` – самая короткая функция копирования (см. Стауструп Б. *Язык программирования C++*.).

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

void cpy (char *, char *);

int main ()
{
    char s1 [30];
    char s2 [] = "September";
    cpy(s1,s2);
    cout << s1<<"\n";
```

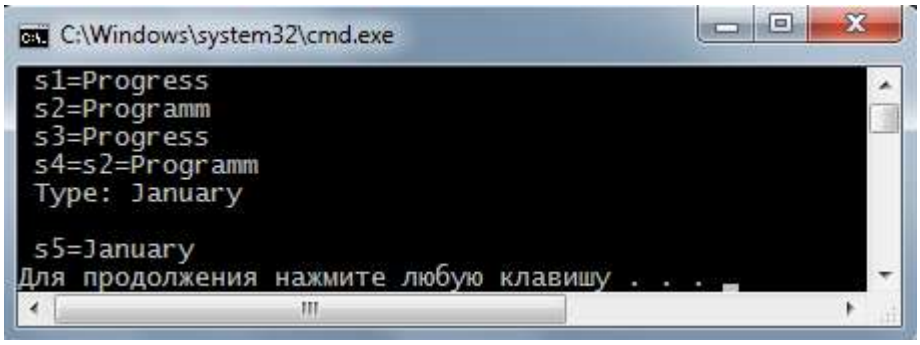
```
    cpy(s2,"November");
    cout << s2<<"\n";
    return 0;
}

void cpy(char *p, char *q)
{ while (*p++ = *q++); }
```



В языке C++ для строковых данных введен также класс `string`. Класс содержит методы, упрощающие обращение со строками – строки можно сравнивать, присоединять, выполнять поиск в содержимом строк. Вполне обоснованной является рекомендация Б. Страуструпа (см. *Язык программирования C++*) – свести к минимуму использование массивов символов, использовать класс `string`.

Пример. Создание строк – объектов класса `string`.

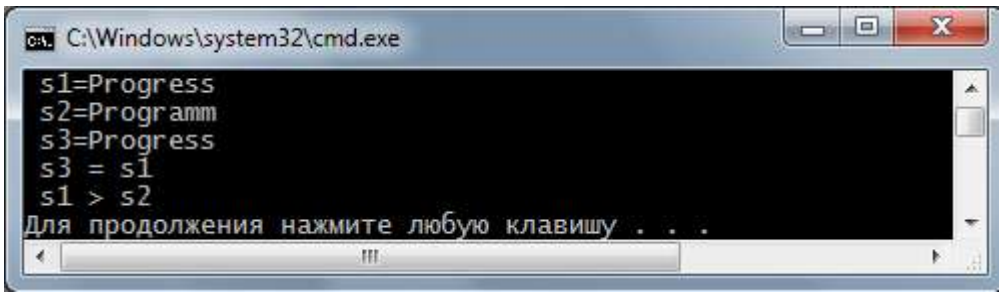
```
string s1("Progress");
string s2 ="Programm";
string s3(s1); // s3 создается такой же, как s1
string s4,s5;
cout<<"\n s1="<<s1;
cout<<"\n s2="<<s2;
cout<<"\n s3="<<s3;
// Присваивание строк:
s4=s2;
cout<<"\n s4=s2="<<s4;
// Ввод строк:
cout<<"\n Type: "; cin>>s5;
cout<<"\n s5="<<s5<<"\n";
```



Пример. Сравнение строк.

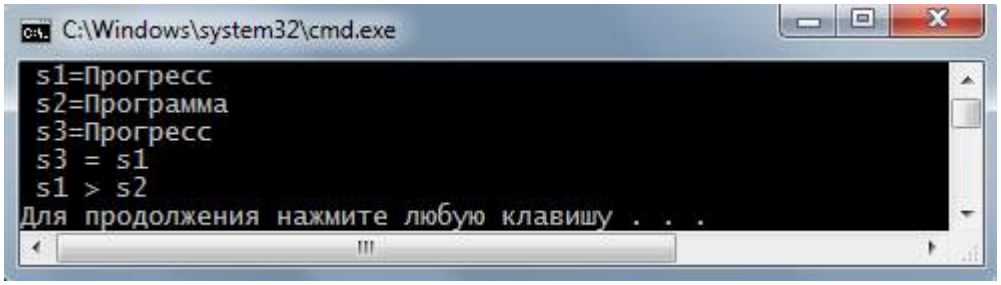
```
// сравнение строк
```

```
string s1("Progress");
string s2 ="Programm";
string s3(s1);
cout<<"\n s1="<<s1;
cout<<"\n s2="<<s2;
cout<<"\n s3="<<s3;
if(s3==s1) cout<<"\n s3 = s1";
else cout<< cout<<"\n s3 != s1";
if (s1>s2) cout<<"\n s1 > s2";
else cout<<"\n s1 <= s2";
```



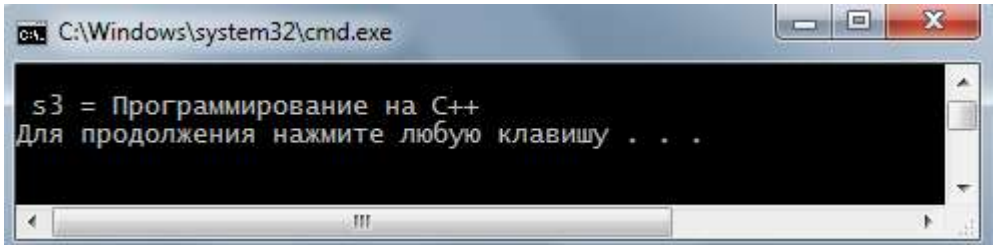
Пример. Сравнение строк с кириллицей.

```
setlocale(LC_CTYPE, "rus");//русификация консоли
string s1("Прогресс");
string s2 ="Программа";
string s3(s1);
cout<<"\n s1="<<s1;
cout<<"\n s2="<<s2;
cout<<"\n s3="<<s3;
if(s3==s1) cout<<"\n s3 = s1";
else cout<< cout<<"\n s3 != s1";
if (s1>s2) cout<<"\n s1 > s2";
else cout<<"\n s1 <= s2";
```



Пример. Конкатенация строк.

```
setlocale(LC_CTYPE, "rus");//русификация консоли
string s1("Программирование");
string s2 ="C++";
string s3=s1 +" на " + s2;
cout<<"\n s3 = "<<s3;
```



Замечание. Следующий код воспринимается компилятором как ошибочный.

```
string str;
str="Programming " + " C++";
cout<<"\n str = "<<str;
```

Хотя проблема, в каком-то смысле, надуманная – можно просто слить строки, передвинув кавычки, – укажем ещё одно решение (раскрывающее объектную природу этого вопроса).

```
string str;
str=string("Programming ") + " C++";
cout<<"\n str = "<<str;
```

Здесь был вызван конструктор класса `string` и создан неименованный объект этого класса. Хотя старый синтаксис приведения типа

```
string str;
str=(string)"Programming " + " C++";
```

также даёт выход из ситуации, но не раскрывает, почему проблема возникла. На самом деле, код

```
s3=s1+s2
```

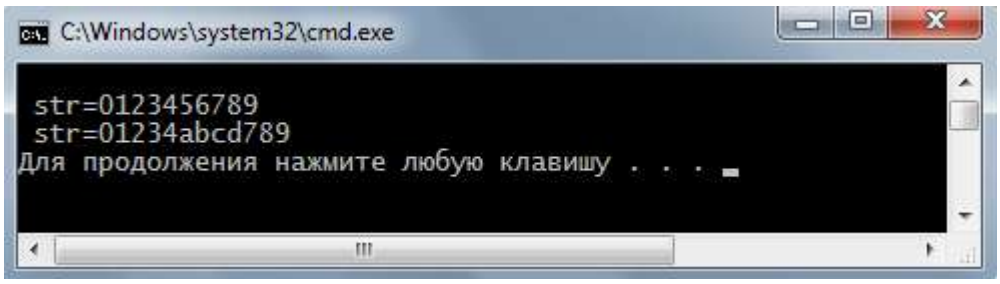
означает

```
s3.operator = (s1.operator +=(s2));
```

т. е. является вызовом перегруженных функций.

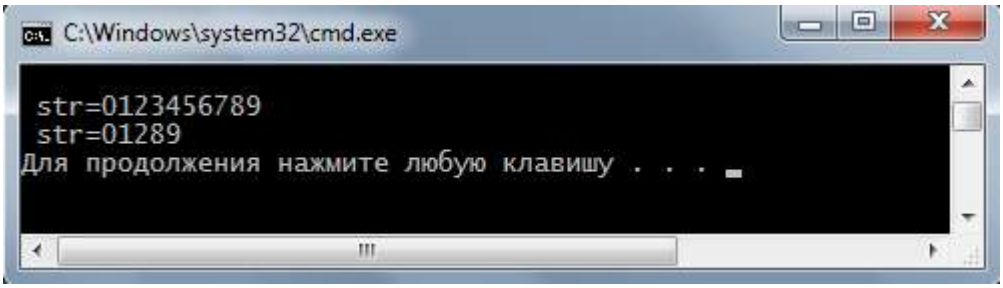
Пример. Замена части строки операцией `replace(n,m,s)`, где `n` – позиция в строке, с которой производится замена, `m` – количество заменяемых символов, а `s` – замещающая строка. В примере замещающий текст равен `"abcd"` – эти `4` символа будут вставлены с `5` позиции вместо `2` символов строки `str`.

```
// Замена части строки
string str("0123456789");
cout<<"\n str="<<str;
str.replace(5,2,"abcd");
cout<<"\n str="<<str;
```



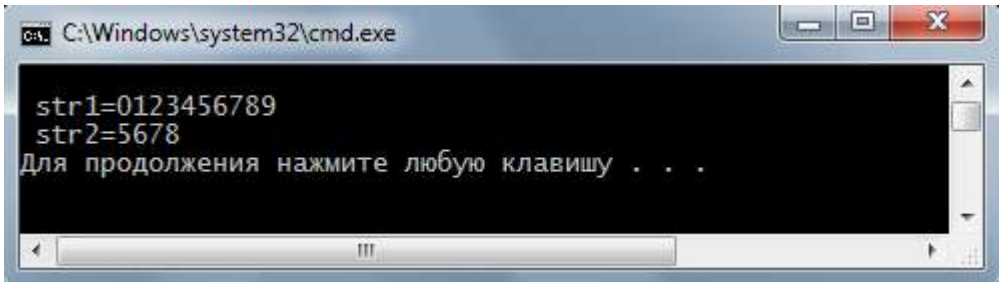
Пример. Удаление части строки операцией `erase(n,m)`, где `n` – позиция в строке, начиная с которой выполняется операция, а – `m` количество удаляемых символов.

```
// Удаление части строки
string str("0123456789");
cout<<"\n str="<<str;
str.erase(3,5); // 5 символов, начиная с 3-го
cout<<"\n str="<<str;
cout<<"\n";
```



Пример. Выделение подстроки операцией `substr(n,m)`, где `n` – позиция в строке, начиная с которой выполняется операция, а – `m` количество выделяемых символов.

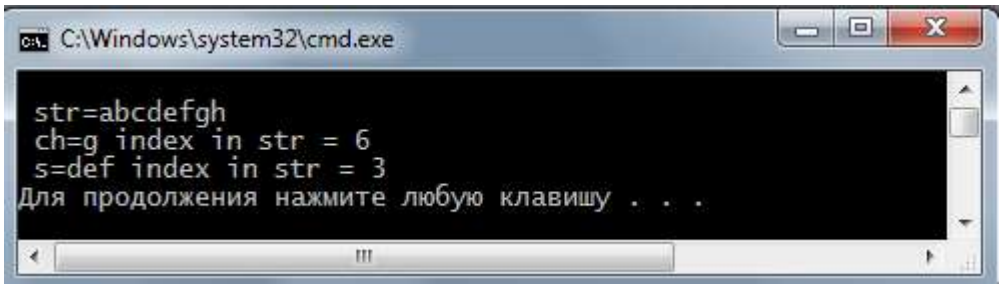
```
// Выделение части строки
string str1("0123456789");
cout<<"\n str1="<<str1;
string str2;
str2=str1.substr(5,4);// 4 символа, начиная с 5-го
cout<<"\n str2="<<str2;
cout<<"\n";
```



Пример. Поиск вхождения символов в строку с помощью операции `find()`.

```
// Поиск в строке
```

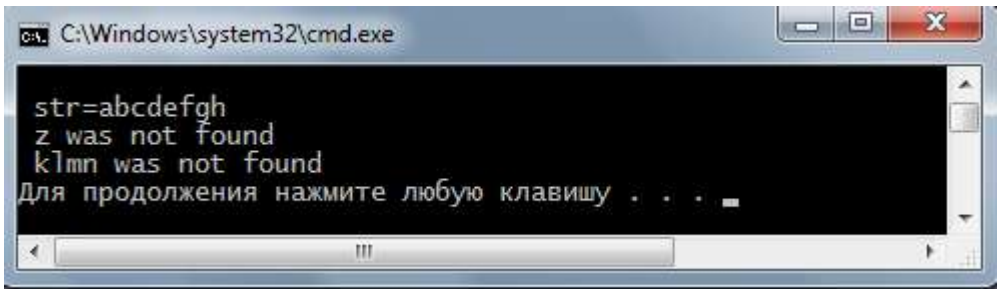
```
string str("abcdefgh");
cout<<"\n str="<<str;
char ch='g';
int ind_ch = str.find(ch); //
cout<<"\n ch="<<ch<<" index in str = "<<ind_ch;
string s="def";
int ind_s = str.find(s);
cout<<"\n s="<<s<<" index in str = "<<ind_s<<"\n";
```



Замечание. Класс `string` поддерживает несколько перегруженных операций поиска `find(s)`. Подробности см., напр., Прата С. *Язык программирования C++ (C++11). Лекции и упражнения*.

Пример. Поиск вхождения символов в строку с помощью операции `find()`. Поиск может оказаться безрезультатным – выяснить это можно с помощью переменной `npos`, как показано в примере.

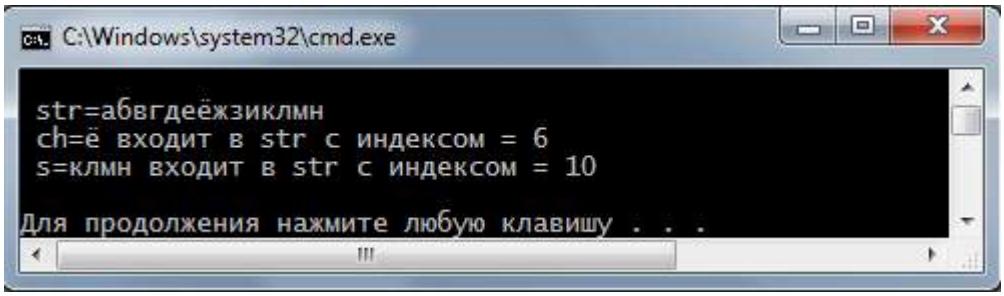
```
// Поиск в строке, npos
string str("abcdefgh");
cout<<"\n str="<<str;
char ch='z';
int ind_ch = str.find(ch);
if (ind_ch != string::npos)
     cout<<"\n ch="<<ch<<" index in str = "<<ind_ch;
else cout<<"\n "<< ch<<" was not found ";
string s="klmn";
int ind_s = str.find(s);
if (ind_s != string::npos)
     cout<<"\n s="<<s<<" index in str = "<<ind_s<<"\n";
else cout<<"\n "<<s<< " was not found ";
```



Замечание. Переменная `npos` является статическим элементом класса `string` и равна максимально возможному количеству символов в объекте класса `string`.

Пример. Операция `find()` успешно ищет текст с кириллицей.

```
// Поиск в строке с кириллицей
setlocale(LC_CTYPE, "rus");//русификация консоли
string str("абвгдеёжзиклмн");
cout<<"\n str="<<str;
char ch='ё';
int ind_ch = str.find(ch);
if (ind_ch != string::npos)
   cout<<"\n ch="<<ch<<" входит в str с индексом="
                      <<ind_ch;
else cout<<"\n "<< ch<<" не найден ";
string s="клмн";
int ind_s = str.find(s);
if (ind_s != string::npos)
    cout<<"\n s="<<s<<" входит в str с индексом = "
                      <<ind_s<<"\n";
else cout<<"\n "<<s<< " не найден ";
```



Замечание. В языке C++ для работы со строками имеется еще один класс – `String`. Функционально этот класс уступает классу `string`. Подробнее об использовании класса `String` см., напр., Штерн В. *Основы C++. Методы программной инженерии*.

Многомерные массивы

В C и C++ поддерживаются многомерные массивы. При объявлении многомерных массивов задается тип элементов массива, имя массива и затем, в отдельных квадратных скобках – количества элементов по каждой размерности:

тип имя_массива **[*Размер1*][*Размер2*]...[*РазмерN*];**

Пример.

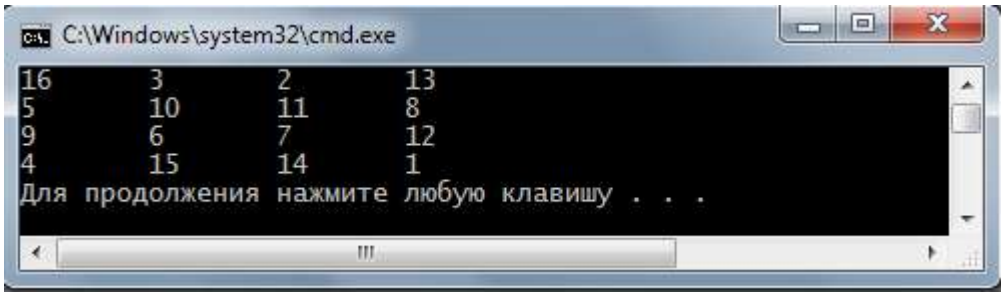
```
// многомерные массивы
const int  K=4;
const int  N=3;
```

```
const int  M=5;
int one[N];
int two[N][M];
int three[K][N][M];
for (int i=0;i<K;i++) one[i]=i*10;
for (int p=0;p<N;p++)
   for (int q=0;q<M;q++) two[p][q]=p*q;
for (int i=0;i<K;i++)
  for (int p=0;p<N;p++)
    for (int q=0;q<M;q++)
        three[i][p][q]=one[i]+ two[p][q];
```

Многомерные массивы можно инициализировать, используя синтаксис, аналогичный одномерному случаю.

Пример. Инициализация двумерного массива (матрицы). Значения можно записать подряд или же разбить на группы, выделив каждую строку матрицы, фигурными скобками (этот способ называют ***subaggregate grouping*** , т.е. ***группирование подагрегатов***).

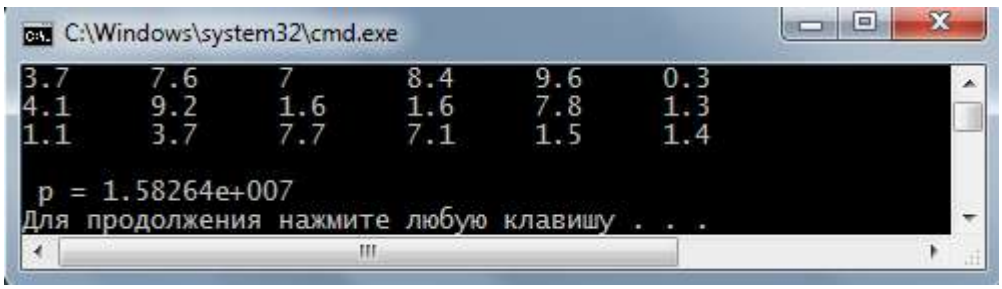
```
// Магический квадрат
const int  N=4;
int mag1[N][N]={16,3,2,13,5,10,11,8,
                                  9,6,7,12,4,15,14,1};
int mag2[N][N] ={
                  {16,3,2,13},
                  {5,10,11,8},
                  {9,6,7,12},
                  {4,15,14,1}
                };
for (int i=0;i<N;i++){
    for (int j=0;j<N;j++) cout<<mag1[i][j]<<"\t";
    cout<<"\n"; // новая строка матрицы
}
```



Пример. Вычисление произведения сумм элементов строк матрицы: $p = \prod_{i=1}^{N} \sum_{j=1}^{M} a_{ij}$. Матрица заполняется случайными числами.

```
// P= \Pi_{i=1}^n \Sigma_{j=1}^m a_{ij}
```

```
#include "stdafx.h"
#include <iostream>
#include <ctime> // для srand()
using namespace std;
const int N=5;
const int M=6;
int main(int argc, _TCHAR* argv[])
{
  double a[N][M]; int i,j;

  // заполняем матрицу случайными числами
  srand(time(NULL)); /* значение времени в генератор
случайных чисел */
  for (i=0;i<N;i++)
    for (j=0;j<M;j++) a[i][j] = (rand()%100) * 0.1;

  double p,s;
  p=1.0;// произведение
  for (i=0;i<N;i++){
        s = 0; // s - сумма элементов строки
        for (j=0; j<M;j++)  s += a[i][j];
        p *= s;
  }

  // Вывод результатов
  for (i=0;i<N;i++){
    for (j=0;j<M;j++) cout<<a[i][j]<<"\t";
    cout<<"\n"; // новая строка матрицы
  }
  cout<<"\n p = "<<p<<"\n";
  return 0;
}
```



Пример. Элементы матрицы вводятся с клавиатуры. Вычисляется след матрицы (сумма диагональных элементов).

```
// След матрицы
const int N=3;
double a[N][N]; int i,j;
```

```
// Блок ввода значений
for (i=0;i<N;i++)
    for (j=0;j<N;j++) {
        cout<<"a["<<i<<"]["<<j<<"]="; cin>>a[i][j];

    }

double tr_a=0.0;
for (i=0;i<N;i++) tr_a+=a[i][i];

cout<<"\n Tr (a) = "<<tr_a<<"\n";
```

Многомерные массиы могут быть безразмерными – можно не указывать размер самого левого измерения, если при объявлении используется инициализатор.

Пример. Безразмерный двумерный массив.

```
// Магический квадрат
const int M=4; /* граница 2-го измерения массива*/
int mag[][M] ={
               {16,3,2,13},
               {5,10,11,8},
               {9,6,7,12},
               {4,15,14,1}
             };
// Вычисляем границу первого измерения:
int n = /* граница 1-го измерения массива*/
 sizeof(mag)/* память под весь массив*/
  /(sizeof(int)*M)/* память, занятая строкой массива*/;
for (int i=0;i<n;i++){
  for (int j=0;j<M;j++) cout<<mag[i][j]<<"\t";
  cout<<"\n"; // новая строка матрицы
}
```

Указатели

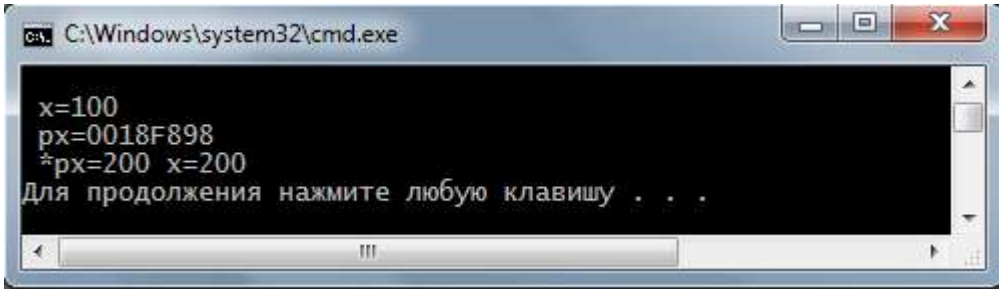
Указатель – это переменная, содержащая адрес другой переменной. Применяя операцию “*” (операция ***разыменования***), получаем значение, записанное по данному адресу. С помощью операции “&”, применённой к переменной, можно узнать адрес, по которому эта переменная хранится в памяти.

При объявлении указателя также используется “*”, кроме того, указывается тип данных, на которые ссылается указатель.

Тип_данных *имя_указателя;

Пример. Объявление указателей и обращение к памяти. Значение переменной `px` – адрес памяти, а `*px` – данные, записанные по адресу `px`.

```
int x=100; // переменная
cout<<"\n x="<<x;
int* px;  // указатель на int
px = &x; // указателю присвоили адрес переменной x
*px =200; /* Изменения значения, записанного по адресу
px */
cout<<"\n px="<<px;
cout<<"\n *px="<<*px<<" x="<<x<<"\n";
```



При создании указателя память выделяется только для хранения адреса. Для выделения памяти под данные используется оператор `new` (или функция `malloc()`). Выделение памяти оператором `new`:

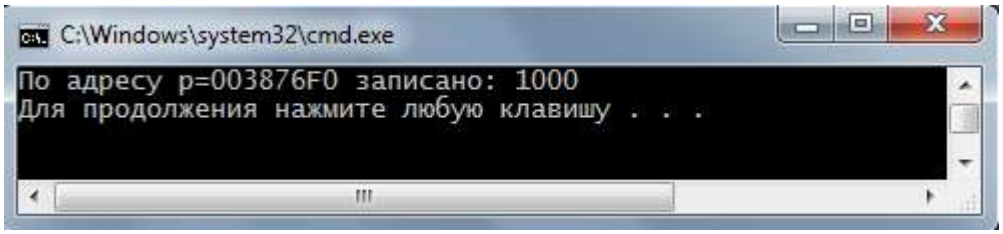
указатель = new Тип

Оператор `new` выделяет память в количестве, необходимом для хранения данных указанного типа. Указатель в левой части оператора присваивания должен быть указателем на тот же тип данных.

Пример. Оператором `new` выделена память для данных типа `int`, с помощью оператора `delete` выполнено освобождение памяти.

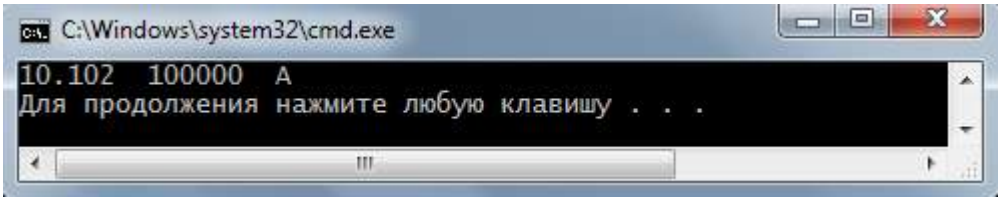
```
#include <iostream>
using namespace std;
int main()
{
    setlocale(LC_CTYPE, "rus");//русификация консоли
    int *p;
    p = new int; /* Выделение памяти для целого. Далее
выполняется проверка, удачно ли произошло выделение
памяти, если нет – выход из программы с кодом
завершения 1: */
    if(!p) {
         cout << "\nНедостаточно памяти\n";
         return 1;
     }
    *p = 1000;
```

```
    cout << "По адресу p="<<p<<" записано: "<<*p<<"\n";
    delete p; // освобождение памяти
    return 0;
}
```



Пример. Использование операторов `new` и `delete`

```
#include <iostream>
using namespace std;
int main()
{
    double *px = new double;
    *px = 10.102;
    int *pn = new int;
    *pn = 100000;
    char *pc = new char;
    *pc = 'A';
    cout << *px << '\t' << *pn << '\t' << *pc;
    cout << '\n';
    delete px;
    delete pn;
    delete pc;
    return 0;
}
```



Возможность выделения памяти с помощью оператора `new` появилась в C++, в языке C для этой цели используется функция `malloc()`:

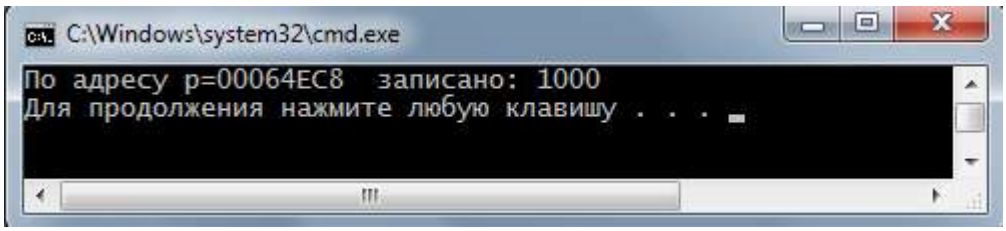
указатель = malloc(количество байт);

Поскольку указатель типизированный, т.е. ссылается на блок памяти, занятой данными определенного типа, требуется выполнить явное приведение типа значения функции. Кроме того, функция `sizeof()` поможет правильно определить необходимое количество байт.

указатель = (Тип *) malloc(sizeof(Тип));

Пример. Выделение памяти функцией `malloc()`.

```
setlocale(LC_CTYPE, "rus");//русификация консоли
int * p;
p = (int *) malloc(sizeof(int)); /* Выделение памяти
для целого. */
if(!p) /* Неудача при выделении памяти */
  {
     printf("\nНедостаточно памяти\n");
     return 1;
  }
*p = 1000;
printf("По адресу p=%p  записано: %d \n",p,*p);
free(p);
```



Замечание. Для вывода значений указателей в функции `printf()` можно использовать спецификатор формата `%p`, который отображает адреса в формате, используемом компилятором.

Как уже было отмечено, для обозначения переменной–указателя используется звёздочка “*”. Пробелы между символом “*”, типом и именем переменной не имеют значения.

```
int  *ptr;
int* ptr;
int * ptr;
```

Приведенные формы объявления указателя равносильны, предпочтения субъективные. При использовании первой формы объявления подчёркивается, что `*ptr` имеет значение `int`. Во втором случае отмечается, что `int*` – это тип “указатель на `int`”.

Следует учитывать особенности использования инструкции “,” (запятая) при объявлении указателей. Так, объявление

```
int * p1, p2;
```

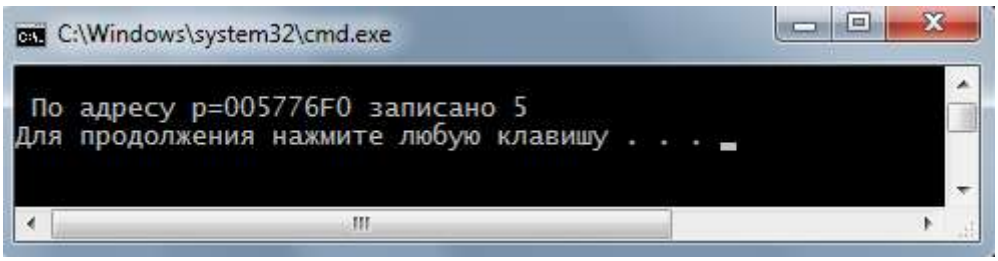
создаёт один указатель `p1` и переменную `p2` типа `int`. Объявление

```
int * p1, * p2;
```

создаёт два указателя `p1` и `p2`.

Пример. Пример инициализации динамической переменной. Значение, используеммое для инициализации, указано в круглых скобках оператора `new`.

```
#include <iostream>
using namespace std;
int main()
{
 setlocale(LC_CTYPE, "rus");//русификация консоли
 int *p;
 p = new int(5); // начальное значение равно 5
 if(!p) {
   cout << "\nНедостаточно памяти\n ";
   return 1;
 }
 cout<<"\n По адресу p="<<p<<" записано "<<*p<<"\n";
delete p; // освобождение памяти
 return 0;
}
```



Для динамически размещаемого одномерного массива используется следующая форма оператора `new`:

p=new type [size]

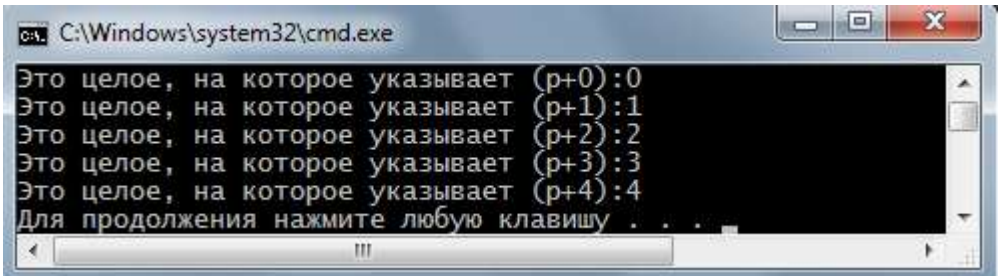
Для удаления динамически размещаемого одномерного массива используется оператор

delete [] p;

Пример. Размещение массива из 5 целых чисел.

```
#include <iostream>
using namespace std;
int main()
{
 setlocale(LC_CTYPE, "rus");//русификация консоли
 int *p;
 int i;
 p = new int [5]; // выделение памяти для 5 целых
 // Проверим, что память выделена
 if(!p) {
```

```
    cout <<"\nНедостаточно памяти\n";
    return 1;
 }
 for(i=0; i<5; i++) p[i] = i;
 for(i=0; i<5; i++) {
  cout<<"Это целое, на которое указывает (p+"<<i<<"):";
  cout << p[i] << "\n";
 }
 delete [] p; // освобождение памяти
 return 0;
}
```



Арифметика указателей

В C/C++ разрешены несколько арифметических операций с участием переменных-указателей. К указателям можно прибавлять и вычитать целые числа, выполнять операции инкремента и декремента, а также вычитать два указателя.

Увеличение и уменьшение указателя на целое число

Увеличение указателя на `1`, означает, что указатель будет ссылаться на следующий блок памяти, занятый переменной того же типа. Увеличение же указателя на целое число `n`, передвигает указатель на `n` блоков в сторону увеличения адресов.

Уменьшение указателя на целое число `n`, передвигает указатель на `n` блоков в сторону уменьшения адресов.

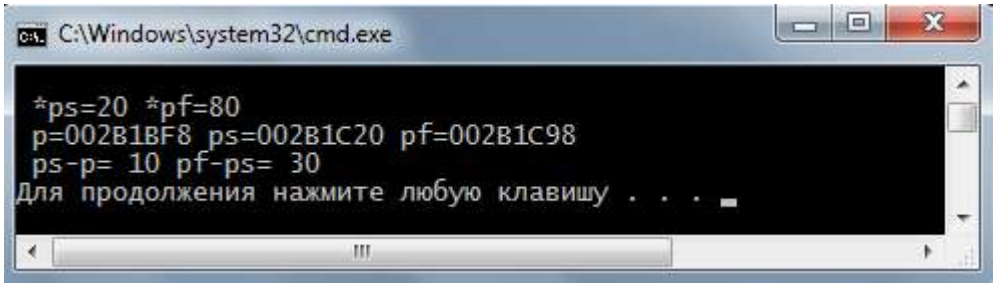
В примере “Размещение массива из 5 целых чисел” выражение `p[i]` используется для обращения к `i`-му элементу массива. Компилятор преобразует это выражение к виду `*(p+i)`. Это означает, что к адресу `p` будет добавлено количество байт, занятых `i` элементами данного типа (в примере – тип `int`).

Вычитание указателей

Можно вычитать два указателя одного и того же типа. Результат этой операции – количество данных, уместившихся между адресами, являющихся значениями этих указателей.

Пример. Выделяется блок для размещения 50 целых чисел. Указателю `ps` присвоено значение `p+10`, т.е. этот указатель ссылается на 10 блок данных, а значения адресов отличаются на число 40 (28 в шестнадцатеричной системе счисления). Указатель `pf` ссылается на 40 блок данных. Из одного указателя вычитается другой.

```
int *p, *ps, *pf;
p = new int [50];
for (int i=0;i<50;i++) *(p+i) = i * 2;
ps=p+10; pf=p+40;
cout<<"\n *ps="<<*ps<<" *pf="<<*pf;
cout<<"\n p="<<p<<" ps="<<ps<<" pf="<<pf;
cout<<"\n ps-p= "<<ps-p<<" pf-ps= "<<pf-ps;
```

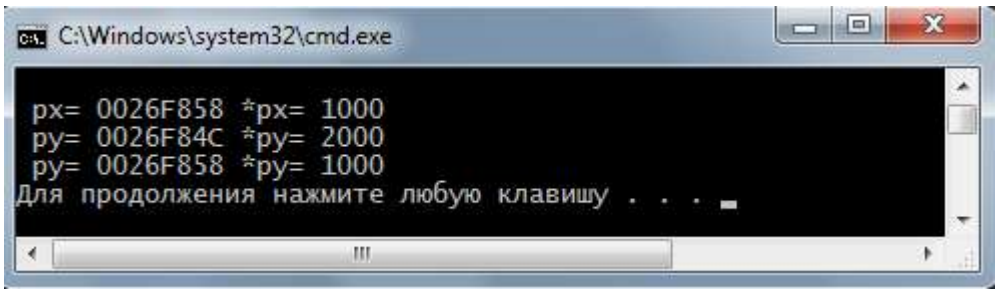


Присваивание указателей

Переменной–указателю можно присвоить значение другого указателя того же типа данных, а также, адрес переменной того же типа.

Пример. Операция присваивания указателей, в результате которой оба указателя `px` и `py` ссылаются на область памяти, в которой размещена переменная `x`.

```
 int x=1000; int y=2000;
 int *px=&x; int *py=&y;
 cout<<"\n px= "<<px<<" *px= "<<*px;
 cout<<"\n py= "<<py<<" *py= "<<*py;
 py=px; /* указателю присвоили значение другого
указателя */
 cout<<"\n py= "<<py<<" *py= "<<*py;
```



Преобразование типа указателя

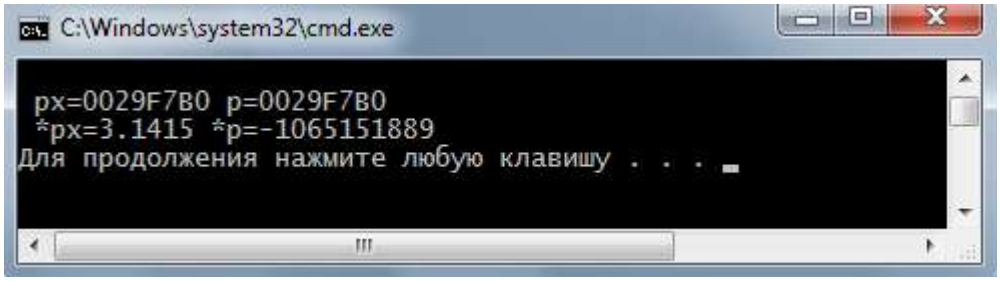
Указатель на один тип данных путем преобразования можно рассматривать как указатель на другой тип данных. Обычно подобное преобразование выполняют для указателей, имеющих тип `void *`, но разрешено выполнять преобразование и для указателей других типов.

Пример. Выделение памяти функцией `malloc()`. Эта функция возвращает значение типа `void *`, в приведенном коде выполнено преобразование к типу `int *`.

```
int * p;
p = (int *) malloc(sizeof(int)); /* Выделение памяти
для целого. */
```

Пример. Как не стоит делать преобразование типа. Указателю `p` на `int` присвоено значение указателя `px` на `double` и выполнено необходимое преобразование типа. В результате, обе переменные `px` и `p` имеют одинаковые значения, но ссылаются на разные данные, – первый указатель ссылается на блок памяти, занятый значением типа `double`, а указатель `p` ссылается только на первые 4 байта этого блока.

```
double x=3.1415; double *px;
px = &x;
int *p;
p=(int *)px;
cout<<"\n px="<<px<<" p="<<p;
cout<<"\n *px="<<*px<<" *p="<<*p;
```



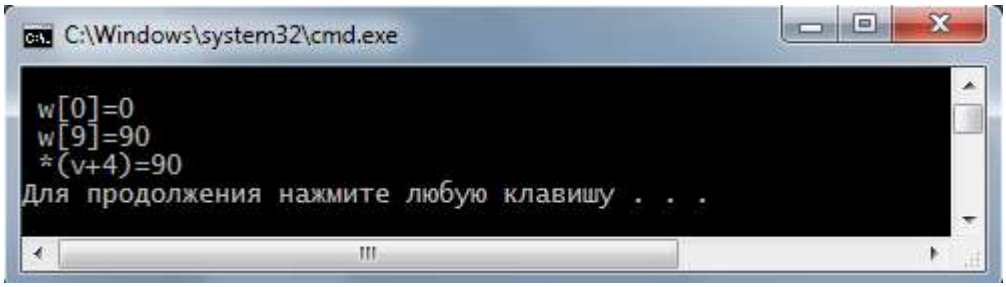
Указатели и массивы

Имя массива является указателем и содержит адрес первого элемента массива. Если, например, `w` имя массива, то `*w` – значение первого элемента массива, а обращение `w[i]` к `i`-му элементу массива есть

сокращенная форма операции с указателями `*(w+i)`. Указатели также можно индексировать, используя запись `v[i]` вместо `*(v+i)`.

Пример.

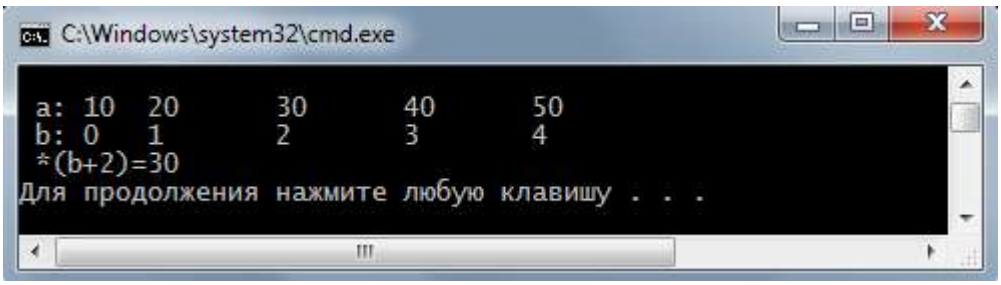
```
const int N=10;
int w[N]=// массив
     {0,10,20,30,40,50,60,70,80,90};
int *v; // указатель
v=w+5;
cout<<"\n w[0]="<<*w;
cout<<"\n w[9]="<<*(w+9);
cout<<"\n *(v+4)="<<v[4];/* указатель можно
индексировать */
```



Хотя указатели и массивы тесно связаны, между ними есть существенное различие – значение указателя можно изменять, а имя массива является константой.

Пример. Указателю `b` присвоено значение указателя `a`. Однако, операция присваивания `a=p` будет воспринята компилятором как ошибка, поскольку, в отличие от `b` указатель `a` является именем массива.

```
const int N=5;
int *b;
b = new int [N];
int a[N]={10,20,30,40,50};
for(int i=0; i<N; i++) b[i] = i;
cout<<"\n a: ";
for(int i=0; i<N; i++) cout<<a[i]<<'\t';
cout<<"\n b: ";
for(int i=0; i<N; i++) cout<<b[i]<<'\t';
b=a; // так можно
cout<<"\n *(b+2)="<<b[2];
/* a=p; // в так нельзя */
```

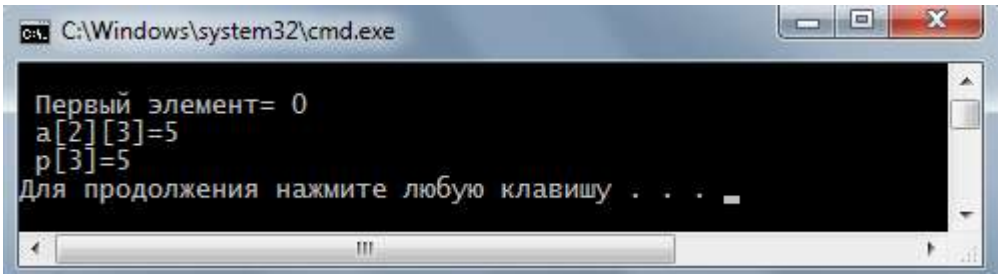


Указатели и многомерные массивы

В многомерных массивах имя массива также указывает на первый элемент массива. Обращение к элементу массива, указанием его индексов, является сокращенной формой выражения с указателями. Пусть `a` – двумерный массив размера `n*m`, где `n`, `m` – заданные константы, тогда выражение `a[i][j]` преобразуется компилятором в операцию с указателями `*(*(a+i)+j)`. Выражение `a[i]` является указателем на первый элемент строки с индексом `i`.

Пример.

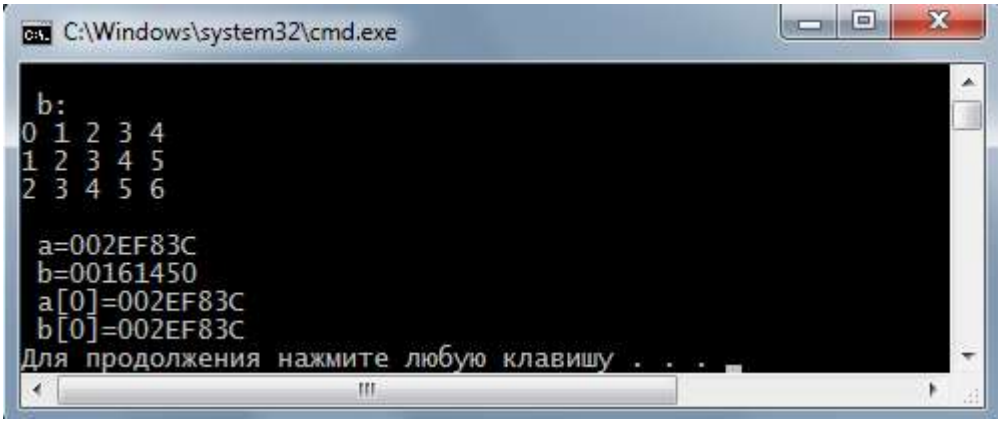
```
const int N=3;
const int M=5;
setlocale(LC_CTYPE, "rus");//русификация консоли
int a[N][M]={{0,1,2,3,4},{1,2,3,4,5},{2,3,4,5,6}};
cout<<"\n Первый элемент= "<<*(*a);
cout<<"\n a[2][3]="<<*(*(a+2)+3);
int *p;
p=a[2];// адрес первого элемента строки 2
cout<<"\n p[3]="<<*(p+3);
```



Пример. Многоуровневая адресация. Обращение к одним и тем же данным с помощью массива и указателя на указатель. Указатели `a` и `b` имеют разные значения, но адреса строк `a[i]` и `b[i]` одинаковые, как следствие, обращения `a[i][j]` и `b[i][j]` дают одинаковый результат.

```
const int N=3;
const int M=5;
int a[N][M]={{0,1,2,3,4},{1,2,3,4,5},{2,3,4,5,6}};
int **b;
b= new int*[N];
for(int i=0; i<N; i++) b[i]=a[i];
```

```
cout<<"\n b: \n";
for(int i=0; i<N; i++){
     for(int j=0; j<M; j++) cout<<b[i][j]<<" ";
     cout<<"\n";
}
cout<<"\n a="<<a;
cout<<"\n b="<<b;
cout<<"\n a[0]="<<a[0];
cout<<"\n b[0]="<<b[0];
```



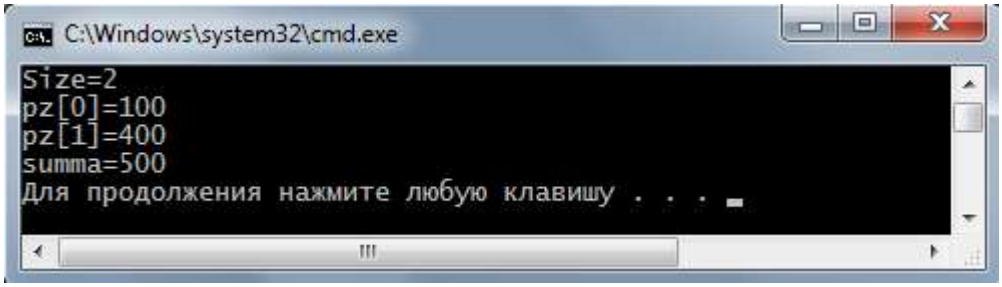
Динамические массивы

Выделение памяти для массива в процессе компиляции называется ***статическим связыванием***. Память под массив выделяется на этапе компиляции. С помощью оператора `new` можем создавать массив во время выполнения программы, размер массива также определяется на этапе выполнения программы. Такой массив называется динамическим, а процесс его создания – ***динамическим связыванием.***

Пример. Размер массива устанавливается во время выполнения.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, _TCHAR* argv[])
{
     int i,size;
     cout<<"Size=";
     cin>>size;
     int * pz= new int[size];
     for(i=0;i<size;i++){
          cout<<"pz["<<i<<"]=";
          cin>>pz[i];
     }
     int sum=0;
```

```
    for(i=0;i<size;i++) sum+=pz[i];
    cout<<"summa="<<sum<<"\n";
    delete [] pz; // освобождаем память
    return 0;
}
```



Пример. Тот же пример, но память выделяется с помощью функции `malloc()`.

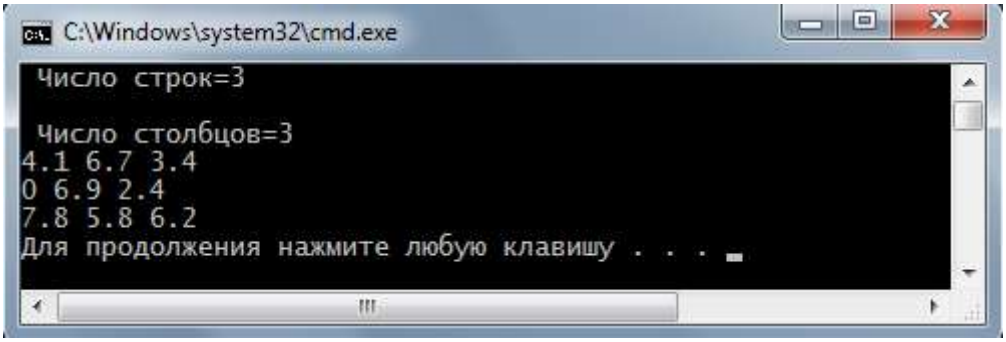
```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
using namespace std;

int main(int argc, _TCHAR* argv[])
{
    int i,size;
    printf("\n Size=");
    scanf("%d",&size);
    int * pz= (int *) malloc(size * sizeof(int));
    for(i=0;i<size;i++){
        printf("pz[%d]=",i);
        scanf("%d",&pz[i]);
    }
    int sum=0;
    for(i=0;i<size;i++) sum+=pz[i];
    printf("summa=%d\n",sum);
    free(pz); // освобождаем память
    return 0;
}
```

Двумерные динамические массивы

Пример. Размер матрицы вводится с клавиатуры во время выполнения программы. Память для размещения данных выделяется с помощью оператора `new`.

```
// Двумерный динамический массив n*m
double **a;
int n, m;
```

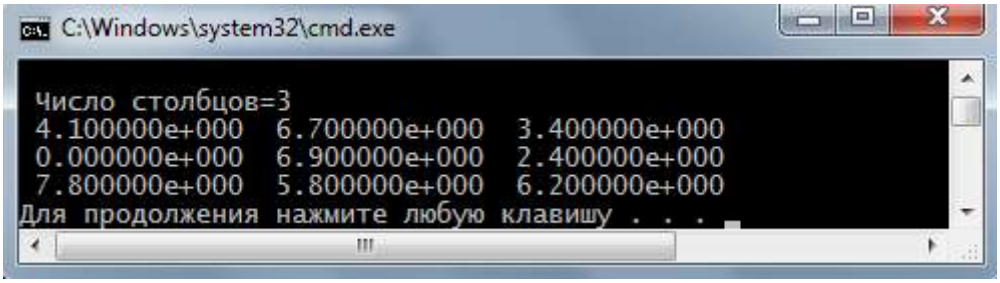
```
setlocale(LC_CTYPE, "rus");//русификация консоли
cout<<"\n Число строк=";
cin>>n;
cout<<"\n Число столбцов=";
cin>>m;
 // Память для размещения данных:
  a=new double* [n];
 for (int i=0;i<n;i++) a[i]=new double [m];
// Заполняем случайными числами:
 for (int i=0;i<n;i++)
   for (int j=0;j<m;j++) a[i][j] = (rand()%100) * 0.1;
// Печатаем:
 for(int i=0; i<n; i++){
    for(int j=0; j<m; j++) cout<<a[i][j]<<" ";
    cout<<"\n";
}
```



Пример. Размер матрицы вводится с клавиатуры во время выполнения программы. Память для размещения данных выделяется с помощью функции `malloc()`.

```
// Двумерный динамический массив n*m
double **a;
int n, m;
int i,j;
setlocale(LC_CTYPE, "rus");
printf("\n Число строк=");
scanf("%d",&n);
printf("\n Число столбцов=");
scanf("%d",&m);
 // Память для размещения данных:
  a=(double **) malloc(n * sizeof(double *));
 for (i=0;i<n;i++) a[i]=(double *) malloc(m *
sizeof(double));
// Заполняем случайными числами
 for (i=0;i<n;i++)
    for (j=0;j<m;j++) a[i][j] = (rand()%100) * 0.1;
// печатаем
```

```
 for(i=0; i<n; i++){
    for(j=0; j<m; j++) printf(" %e ",a[i][j]);
    printf("\n");
}
```



Структуры

Данные различных типов можно объединить под одним именем с помощью структур.

Пример. Создается тип данных для хранения почтового адреса.

```
struct Address {
    char *name;
    int num; // номер дома
    char *street; // улица
    char *city; // город
    int zip; // индекс
};

Address drug;
Address *p=new Address;
Address gruppa[10];
drug.street ="Mushtary"; drug.city="Kazan";
p->city="Moscow";
gruppa[0].num=12;
```

Прежде всего, структура – это тип данных, определяемый программистом. Для объявления структуры используется ключевое слово `struct`, за которым записывается имя структуры. В блоке, выделенном фигурными скобками, объявляются элементы структуры. После закрывающей фигурной скобки обязательно ставится точка с запятой – объявление структуры является оператором. Можно создавать переменные и массивы структурного типа, объявлять функции, возвращающие значения типа структуры и передавать параметры в виде структур. В примере объявлены переменная `drug` и массив `gruppa`, имеющие тип структуры `Address`, а также переменная `p` – указатель на структуру.

В языке С ключевое слово `struct` необходимо использовать и при объявлении переменных (см., напр., Керниган Б.В., Ричи Д.М. *Язык программирования С*.)

```
struct Address drug;
struct Address *p=new Address;
struct Address gruppa[10];
```

В языке С++ эта форма объявления также допустима.

Для доступа к элементам структуры используется “`.`” – оператор доступа к элементу структуры, или, просто оператор точка. Для доступа к элементам структуры с помощью указателя используется оператор стрелки “–>” (состоит из знаков “минус” и “:>”).

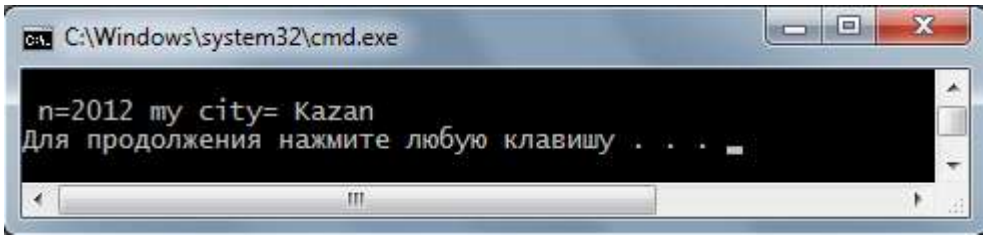
Из ранних версий языка С осталась еще одна форма объявления – с помощью `typedef`

```
typedef struct Address {
    char name[40];
    int num; // номер дома
    char *street; // улица
    char *city; // город
    int zip; // индекс
};
```

Ключевое слово `typedef` можно использовать для создания псевдонимов (алиасов) уже объявленных типов (и не только структур).

Пример. Создаются псевдонимы `address` и `ADDRESS` структуры `Address`. Кроме того, для типа `int` создан псевдоним `I`.

```
typedef struct Address {
    char *street; // улица
    char *city; // город
} address; //псевдоним

typedef struct Address ADDRESS; // ешё один псевдоним
typedef int I; // псевдоним для int
Address drug;
address my;
ADDRESS work;
my.city="Kazan";
I n=2012;
cout<<"\n n="<<n<<" my city= "<<my.city;
```

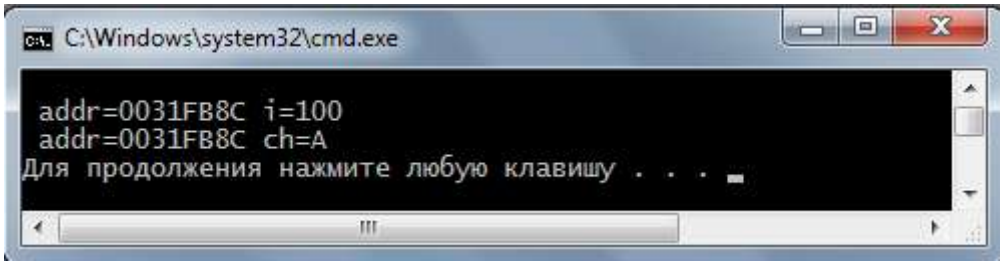


Объединения

Для хранения переменных различных типов в одном месте памяти используются объединения. Объединения позволяют интерпретировать один и тот же набор битов различными способами. Для содания объединений используется ключевое слово `union` и используется тот же синтаксис, что и при объявлении структур.

Пример.

```
union IntOrChar {
  int i;
  char ch;
};

IntOrChar w; // или union IntOrChar w;
w.i=100;
cout<<"\n addr="<<&w<<" i="<<w.i;
w.ch='A';
cout<<"\n addr="<<&w<<" ch="<<w.ch;
```



Компилятор выделяет для хранения данных типа объединения объем памяти, достаточный для самого большого элемента объединения.

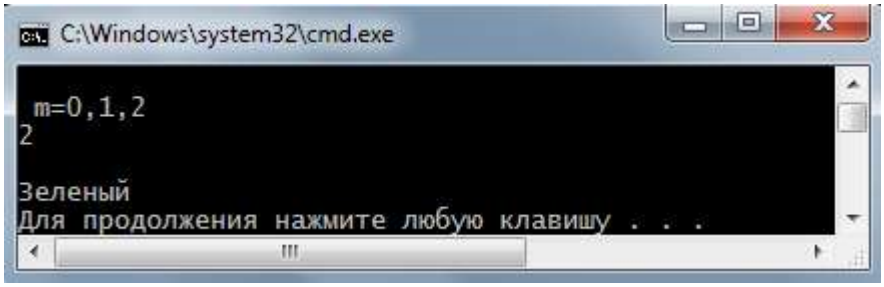
Доступ к элементу объединения выполняется по тем же правилам, что и для структур, т.е. с помощью операторов точки и стрелки.

Перечисления

Языки C и C++ поддерживают ***перечисляемый тип*** (или, просто, ***перечисление***). Этот тип применяется для определения родственных символьных имен, используемых как константы. Для объявления типа используется ключевое слово `enum`, а список перечисления указывается в фигурных скобках через запятую.

Пример. Объявлен перечисляемый тип `tricol` со списком констант `red`, `yellow`, `green`. Переменная `svetophor` может принимать одно из этих значений.

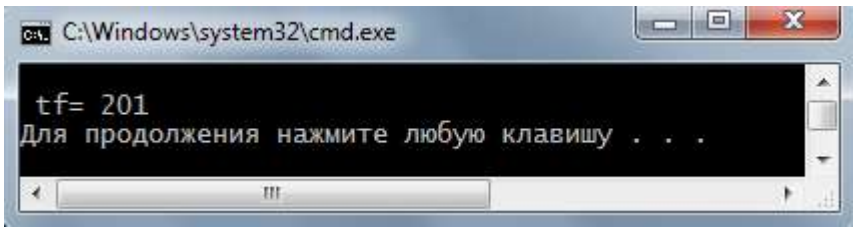
```
setlocale(LC_CTYPE, "rus"); // русификация консоли
enum  tricol {red, yellow, green};
tricol  svetophor; // или enum tricol  svetophor;
svetophor=green;
int m;
cout<<"\n m=0,1,2 \n"; cin>>m;
switch (m){
    case red: cout<<"\nКрасный"; break;
    case yellow: cout<<"\nЖелтый"; break;
    case green: cout<<"\nЗеленый"; break;
    default: cout<<"\nНеопределено";
}
```



Каждый элемент перечислений на самом деле является целым числом, первый элемент списка перечислений имеет значение `0`, следующий равен `1` и т.д. Можно явно указать значения элементов перечисления, как показано в следующем примере.

Пример. Использование инициализаторов. Перечислители, идущие в списке за инициализатором получают следующие по порядку значения.

```
enum  TypeFiles {txt=10, doc=100, pdf=200, exe};
TypeFiles tf=exe;
cout<<"\n tf= "<<tf;
```



Функции

Разделение программы на отдельные логические блоки – функции, позволяет эффективнее управлять кодом.

Использование функции предполагает:

– объявление функции (её прототип), включающее имя функции, тип возвращаемого ею значения и типы параметров функции;

– определение функции, включающее заголовок функции и последовательность операторов, реализующих функцию;

– вызов функции путем указания ее имени и заменой формальных параметров значениями (фактическими параметрами).

Пример. Пример использования функции.

```
#include <iostream>
using namespace std;
int sum(int, int);/* Объявление функции - прототип */
int main()
{   int a,b,c;
    cout << "Vvodim: ";
    cin >> a >> b;
    c=sum(a, b); // Вызов функции
    cout << "\n sum = " << c;
    return 0;
}
int sum(int a, int b)
{ // Определение функции
    return a+b;
}
```

Объявление функции

До обработки вызова функции компилятору должно быть известно её объявление. Прототип функции является оператором и должен заканчиваться точкой с запятой. В прототипе функции можно не приводить имена параметров, – достаточно указать список типов параметров.

```
Возвращаемый_тип Имя_функции(список типов_параметров);
```

Можно обойтись и без прототипа функции – если блок определения функции поместить до блока, в котором производится вызов функции.

Пример. Блок определения функции предшествует её вызову, поэтому в программе нет объявления функции (прототипа).

```
#include <iostream>
using namespace std;
int sum(int a, int b)
{ // Определение функции
    return a+b;
}

int main()
```

```
{    int a,b,c;
     cout << "Vvodim: ";
     cin >> a >> b;
     c=sum(a, b); // Вызов функции
     cout << "\n c = " << c;
     return 0;
}
```

Использование прототипов позволяет компилятору правильно обрабатывать возвращаемые значения функций, следить за соответствием количества аргументов при вызове функции, корректировать применение типов данных для аргументов функции и пытаться, если это возможно, выполнить приведение типов, в случае несоответствия типов при вызове.

Часто прототипы функций помещают в отдельные файлы – ***заголовочные файлы*** (эти файлы имеют расширение `.h`).

Определение функции

В определении функции реализуется алгоритм функции. Определение начинается с заголовка функции. В отличие от прототипа функции, заголовок не должен заканчиваться точкой с запятой. Кроме того, в заголовке обязательно указываются имена параметров.

Функция должна содержать хотя бы один оператор `return` – исключением являются функции, у которых тип возвращаемых значений `void`. Выполнение оператора `return` завершает работу функции и возвращает управление в вызывающую программу. Синтаксис оператора

`return выражение;`

Для функции с типом возвращения `void` оператор `return` необязателен, но его можно использовать завершения работы функции и возврата в вызывающую программу. В этом случае, оператор не содержит возвращаемого значения:

`return;`

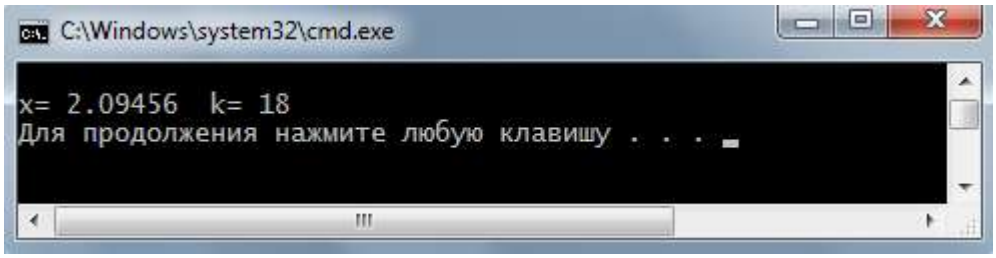
Формальные и фактические параметры

Переменные, перечисленные в заголовке функции, называются ***формальными параметрами***. Эти переменные служат для передачи значений в функцию. При вызове функции формальные параметры заменяются значениями (***фактическими параметрами***).

Все переменные, объявленные в функции, включая параметры функции, действуют только в пределах данной функции. Во время вызова функции компилятор выделяет память для этих переменных, а по завершению выполнения функции, эта память освобождается.

Пример. Метод итераций решения уравнений *y=g(x)*. Приближенное решение вычисляется с точностью, заданной значением переменной `e`.

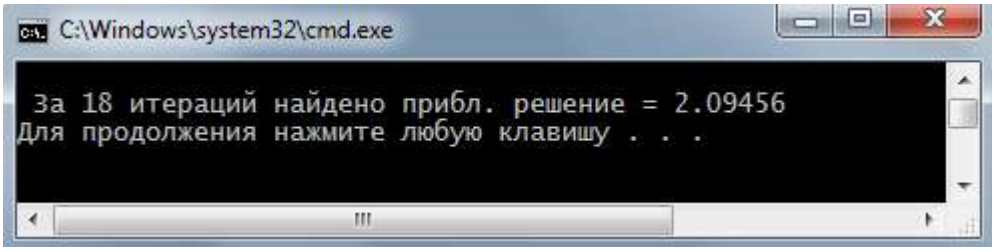
```
// Метод итераций.
//
#include "stdafx.h"
#include <iostream>
#include <cmath>
using namespace std;

const int NMax=1000; // Макс. число шагов метода

double g(double);// Прототип

int _tmain(int argc, _TCHAR* argv[])
{
     double x, x0, y, d, e;
     int k;
     e = 1.e-5; // Точность
     // Начальное приближение:
     x = x0 = 2.5;
     k = 1;
     // Метод итерации:
     do
     {
          y = g(x); // Вызов функции
          d = fabs(y-x);
          k++;
          x = y;
     } while ((d>e) && (k<NMax));
     cout<<"\nx= "<<x<<"  k= "<<k<<"\n";
     return 0;
}

double g(double x) // Определение функции
{
     // Значение правой части x=g(x)
     return - x * x * x * 0.04  + 1.08 * x + 0.2;
}
```



Пример. Реализацию алгоритмов, как правило, оформляют в виде функций, а в функцию `main()` помещают только вызовы этих функций. Следующий код является такой модификацией предыдущего примера.

```
// Метод итераций
//
#include "stdafx.h"
#include <iostream>
#include <cmath>
using namespace std;

const int NMax=1000; // Макс. число шагов метода

// Прототипы функций:
double g(double);
double iterat(double , double , int * );

int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  double y;
  int n;
  y=iterat(2.5,1.e-5,&n);// Вызов функции
  cout<<"\nЗа "<<n<<
       " итераций найдено прибл. решение = "<<y<<"\n";
  return 0;
}

double g(double x) // Определение функции
{
    // Значение правой части x=g(x)
    return - x * x * x * 0.04  + 1.08 * x + 0.2;
}
double iterat(double x,double e, int * k)
 {
     double y;
     double d;
    // Метод итерации:
    *k =1;
    do
    {
        y = g(x);
        d = fabs(y-x);
        (*k)++;
        x = y;
    } while ((d>e) && ((*k)<NMax));
```

```
    return y;
}
```



Передача по значению

Формальные параметры являются отдельными копиями значений фактических аргументов. При завершении функции эти копии уничтожаются.

Пример. Вариант функции обмена в синтаксисе языка С.

```
#include <iostream>
using namespace std;
void swapargs(int * px, int * py);
int main()
{
    int i, j;
    i = 10;    j = 19;
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";
    swapargs(&i, &j);
    cout << "После перестановки: ";
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";
    return 0;
}
void swapargs(int * px, int * py)
{   int t;
    t = *px;     *px = *py;     *py = t;
}
```

Ссылочные переменные

В языке С++ введён новый составной тип данных – ссылочная переменная. Ссылка представляет собой имя, которое является псевдонимом для ранее объявленной переменной. Для объявления ссылочной переменной используется символ ‘**&**’.

Пример.

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

```
int main()
{
    int x;
    int &r = x; // создание независимой ссылки
    x = 10;         // эти две инструкции
    r = 10;       //  идентичны
    r = 100; // x=100;
    // здесь дважды печатается число 100
    cout << x << '\t'<< r <<"\n";
    return 0;
}
```

Основное назначение ссылок – использование в качестве формальных параметров функций. Используя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями.

Пример. Стандартный пример передачи аргументов по ссылке – функция, меняющая значения аргументов

```
#include <iostream>
using namespace std;
void swapargs(int &x, int &y);
int main()
{
    int i, j;
    i = 10;
    j = 19;
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";
    swapargs(i, j);
    cout << "После перестановки: ";
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";
    return 0;
}
void swapargs(int &x, int &y)
{   int t;
    t = x;     x = y;     y = t;
}
```

Пример. Ещё один пример на передачу параметров. Используется и ссылка и указатель.

```
#include <iostream>
using namespace std;
void rneg(int &i); // версия со ссылкой
void pneg(int *i); // версия с указателем
int main()
{   int i = 10;
    int j = 20;
    rneg(i);
    pneg(&j);
    cout << i << ' ' << j << '\n';
    return 0;
}
// использование параметра-ссылки
void rneg(int &i)
{   i = - i; }
// использование параметра-указателя
void pneg(int *i)
{   *i = -*i; }
```

Указатели на функцию

Функции, как и переменные, имеют адреса. Адресом функции является адрес памяти, с которого начинается машинный код функции.

Пример. Еще один вариант реализации метода итераций – в число параметров включена функция (правая часть уравнения $y=g(x)$). Кроме того, приближенное решение, вычисленное функцией `iterat()`, также возвращается с помощью параметра.

```
//  Метод итераций.

#include "stdafx.h"
#include <iostream>
#include <cmath>
using namespace std;

const int NMax=1000; // Макс. число шагов метода

// Прототипы функций:
double g(double);
void iterat(double (*)(double),double, double, double*,
int*);

int _tmain(int argc, _TCHAR* argv[])
{
    double y;
    int n;
```

```
    iterat(g,2.5,1.e-5,&y,&n);// Вызов функции
     cout<<"\ny= "<<y<<"  n= "<<n<<"\n";
    return 0;
}

double g(double x) // Определение функции
{
    // Значение правой части x=g(x)
    return - x * x * x * 0.04  + 1.08 * x + 0.2;
}

void iterat(double (*g)(double),double x,
        double e, double * y, int * k)
 {
    double d;
    // Метод итерации:
    *k =1;
    do
    {
        *y = g(x);
        d = fabs(*y-x);
        (*k)++;
        x = *y;
    } while ((d>e) && ((*k)<NMax));
}
```

Пример. Функция `myoperator()` способна выполнить любую операцию с двумя целыми числами, – эта операция задается третьим параметром (см. также Прата С. *Язык программирования С++*.).

```
int  plus(int,int);
int minus (int,int);
int myoperator (int x, int y, int (*f)(int,int));
int main ()
{
    int m,n;
    m = myoperator(7, 5, plus);
    cout<<"m= "<<m;
    n = myoperator(20, m, minus);
    cout<<" n= "<<n<<endl;
    return 0;
}
int plus(int a, int b){
    return (a+b);
}
int minus (int a, int b){
```

```
     return (a-b);
}
int myoperator (int x, int y, int (*f)(int,int))
{
  int g;
  g = (*f)(x,y);
  return (g);
}
```

Массивы как параметры

При передаче массива в качестве параметра нужно:

– в заголовке функции после имени массива указать пустые квадратные скобки, например,

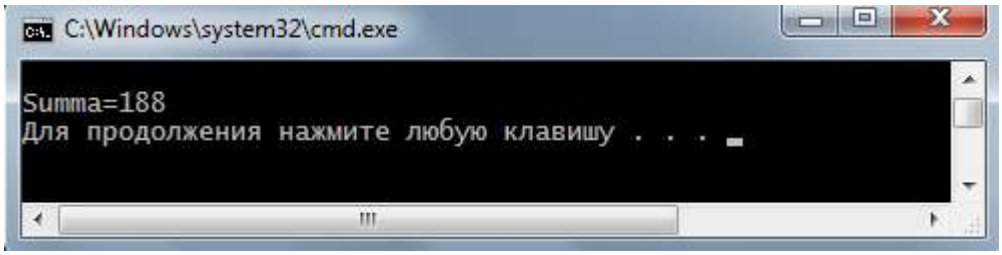
```
int sum_arr(int arr[],int size_arr)
```

– а в вызове функции указать имя массива уже без квадратных скобок:

```
sum=sum_arr(mass,N);
```

Пример. Использование массива как параметра функции.

```
// Массив как параметр функции
//
#include "stdafx.h"
#include <iostream>
using namespace std;

const int N=8;
int sum_arr(int arr[],int);//прототип

int _tmain(int argc, _TCHAR* argv[])
{
   int mass[N]={11,22,17,26,29,22,27,34};
   int sum=sum_arr(mass,N);
   cout<<"\nSumma="<<sum<<"\n";
   return 0;
}

int sum_arr(int arr[],int size_arr)
{
   int s=0;
   for (int i=0;i< size_arr;i++) s+=arr[i];
   return s;
}
```



Передать массив в функцию можно также в форме обычного указателя.

Пример. Ещё одна версия передачи массива как параметра – формальный параметр функции объявлен как указатель.

```
//  Массив как параметр функции
//
#include "stdafx.h"
#include <iostream>
using namespace std;

const int N =8;
int sum_arr(int*,int );//прототип

int _tmain(int argc, _TCHAR* argv[])
{
  int mass[N]={11,22,17,26,29,22,27,34};
  int sum=sum_arr(mass,N);
  cout<<"\nSumma="<<sum<<"\n";
  return 0;
}
int sum_arr(int* arr,int n)
{
  for(int i=0;i<n;i++) {
    s+=arr[i];
    arr[i]=-arr[i];// поменяли знак
   }
  return s;
}
```

Из заголовка функции не видно входным или выходным параметром является массив. Любые изменения элементов массива в теле функции являются изменением значений массива, переданного в качестве параметра. Чтобы не допустить таких изменений, можно добавить в объявление массива-параметра модификатор **const** и тогда компилятор будет выдавать сообщение об ошибке при попытке изменить значения массива.

Пример. Использование массива как параметра функции – с помощью модификатора `const` запрещено изменение значений массива в функции.

```
#include "stdafx.h"
#include <iostream>
```

```
using namespace std;

const int N=8;
int sum_arr(const int [],int);//прототип

int _tmain(int argc, _TCHAR* argv[])
{
     int mass[N]={11,22,17,26,29,22,27,34};
     int sum=sum_arr(mass,N);
     cout<<"\nSumma="<<sum<<"\n";
     return 0;
}

int sum_arr(const int arr[],int n)
{
   int s=0;
   for(int i=0;i<n;i++) {
      s+=arr[i];
      arr[i]=-arr[i];// теперь это ошибка
   }
}
```

Пример. Вычисление границы безразмерного символьного массива оформлено в виде функции `size_char()`.

```
int size_char(char *s); // прототип

int _tmain(int argc, _TCHAR* argv[])
{
     char s[]="November";
     int l_s =size_char(s);
     cout<<"\nLength "<<s<<" = "<<l_s<<"\n";
}
int size_char(char *s)
{ // вычисление длины строки s
     int l_s=0; // l_s - длина строки s
     while (s[l_s++] !='\0');
     return l_s;
}
```

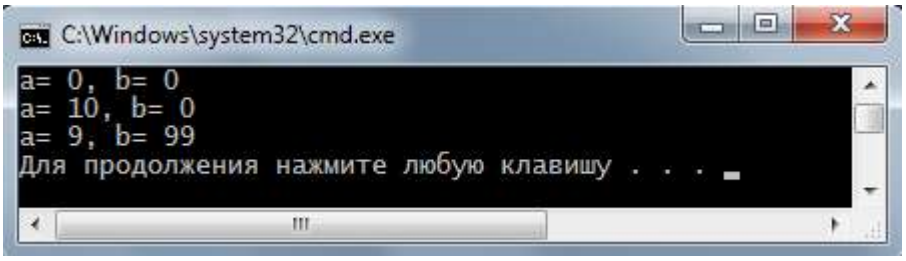
Замечание. В разделе “Массивы” были рассмотрены также безразмерные числовые массивы. Написать функцию вычисления границы таких массивов, как только что было сделано для символьных массивов, не получится – в функцию передается указатель только на начало области памяти, занятой массивом и нет информации о том, где эта область заканчивается. У безразмерных символьных массивов такая информация есть – символ `'\0'`.

Аргументы по умолчанию

В языке С этого средства не было, оно появилось в С++. При объявлении функции можно задать значения по умолчанию для одного или нескольких параметров в списке. При вызове функции можно пропустить аргумент по умолчанию – в функцию будет передано значение, назначенное по умолчанию. В списке параметров аргументы по умолчанию размещаются последними.

Пример. Функция `f()` содержит два параметра, причем оба параметра имеют значения по умолчанию.

```
#include <iostream>
using namespace std;
void f(int a = 0, int b = 0)
{
   cout<<"a= "<< a <<", b= "<<b<<'\n';
}
int main()
{
  f();// переданы значения по умолчанию
  f(10);//второму параметру – значение по умолчанию
  f(9, 99);
  return 0;
}
```



Аргументы по умолчанию необходимо указывать при объявлении функции.

Пример. Функция `f()` содержит два параметра, имеющие значения по умолчанию. Значения по умолчанию заданы в прототипе функции.

```
#include <iostream>
using namespace std;
void f(int = 0, int = 0);
{
   cout<<"a= "<< a <<", b= "<<b<<'\n';
}
int main()
```

```
{
   f();  f(10); f(9, 99);
   return 0;
}
void f(int a, int b)
{
   cout<<"a= "<< a <<", b= "<<b<<'\n';
}
```

Рекурсия

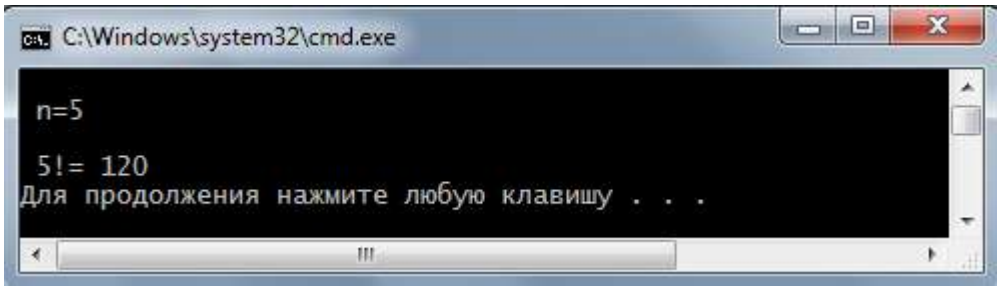
Языки C и C++ поддерживают рекурсию. Рекурсивная функция – это функция, которая содержит вызов этой же функции. Вызов функции может осуществляться и через другую функцию.

Пример. Вычисление факториала *n!=1*2*…*n*. Соотношение

$$n! = (n-1)! * n$$

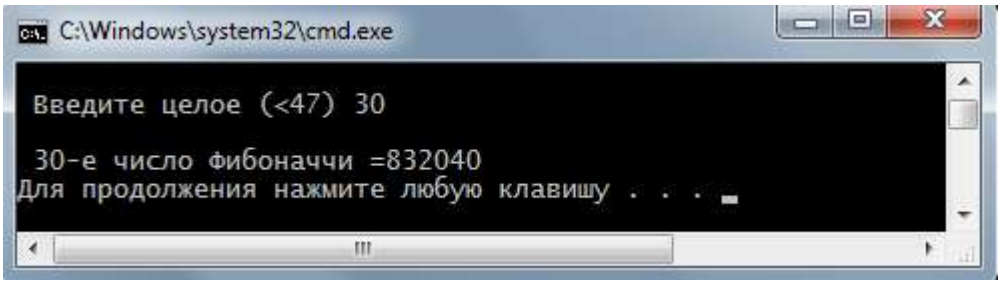
позволяет реализовать рекурсивную функцию вычисления *n!*.

```
#include <iostream>
using namespace std;
int fn(int n);
int main()
{
   int n,m;
   cout<<"\n n=";cin>>n;
   m=fn(n);
   cout<<"\n "<<n<<"!= "<<m<<"\n";
   return 0;
}
int fn(int n)
{
// Вычисление факториала
 if (n==1) return 1;
    else return fn(n-1) * n;
}
```



Пример. Снова числа Фибоначчи.

```
#include <iostream>
using namespace std;
int fn(int n);
int main()
{
// Вычисление чисел Фибоначчи
  setlocale(LC_CTYPE, "rus");//русификация консоли
  int n,m;
  cout<<"\n Введите целое (<47) "; cin>>n;
  m=fibonacci(n);
  cout<<"\n "<<n<<"-е число Фибоначчи ="<<m<<"\n";
  return 0;
}
int fibonacci(int n)
{
    if ((n==0) || (n==1)) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```



Перегрузка функций

Перегрузка функций (*полиморфизм функций*) реализована в C++, в языке C такой возможности не было. Полиморфизм функций позволяет использовать под одним именем несколько функций.

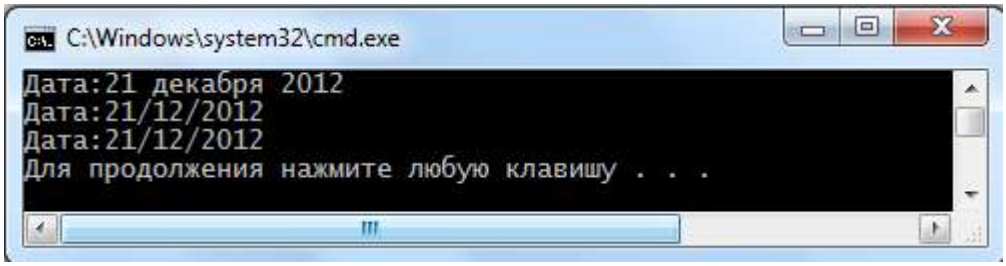
Пример. Перегрузка функций. Две функции с одинаковым именем `f1`. Обратите внимание на аргументы функций – число их различно.

```
#include <iostream>
using namespace std;
void f1(int a);
void f1(int a, int b);
int main()
{   f1(10);
    f1(10, 20);
    return 0;
}
void f1(int a)
{   cout << ", f1(int a) \n";
}
```

```
void f1(int a, int b)
{   cout << ", f1(int a, int b) \n";
}
```

Пример. Перегрузка функций – в программе две функции `date()`, позволяющие передавать дату различными способами.

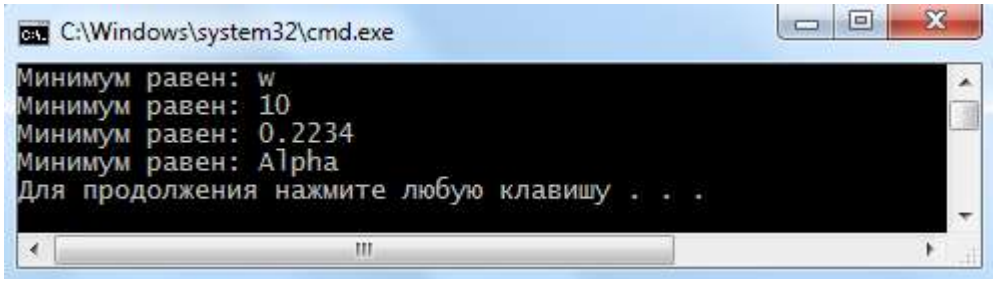
```
#include <iostream>
using namespace std;
void date(char *date); // дата в виде строки
void date(int day, int month, int year);//и в виде
чисел
int main()
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  char *xday="21 декабря 2012";
  date(xday);
  date("21/12/2012");
  date(21,12,2012);
  return 0;
}
// Дата в виде строки
void date(char *date)
{    cout << "Дата:" << date << "\n";
}
// Дата в виде целых чисел
void date(int day, int month, int year)
{   cout << "Дата:" << day << "/";
    cout << month << "/" << year << "\n";
}
```



Пример. Перегрузка функции `minimum()` – в программе сразу несколько функций с этим именем.

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

char minimum(char a, char b);
string minimum(string a, string b);
```

```
int minimum(int a, int b);
double minimum(double a, double b);

int main()
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  cout <<"Минимум равен: " << minimum('x', 'w');
  cout <<"\nМинимум равен: " << minimum(10, 20);
  cout<<"\nМинимум равен: "<<minimum(0.2234, 99.2);
  cout<<"\nМинимум равен: "<<minimum("Alpha", "Beta");
  return 0;
}
// Минимум для char
char minimum(char a, char b)
{
// предварительно символы переводятся в нижний регистр
// функцией tolower()
    return tolower(a) < tolower(b) ? a : b;
}
// Минимум для string
string minimum(string a, string b)
{
    return a < b ? a : b;
}

// Минимум для int
int minimum(int a, int b)
{
    return a < b ? a : b;
}
// Минимум для double
double minimum(double a, double b)
{
    return a < b ? a : b;
}
```



Главную роль в перегрузке функций играет список аргументов, который называют также ***сигнатурой функции***. Если две функции имеют одно и то же количество аргументов, эти аргументы имеют одинаковые типы

и порядок следования, то функции, по определению, имеют одинаковую сигнатуру. В языке C++ можно определить две различные функции с одинаковым именем при условии, если эти функции обладают различными сигнатурами.

Ввод и вывод

Консольный вывод

В языке С консольный вывод выполняется функцией

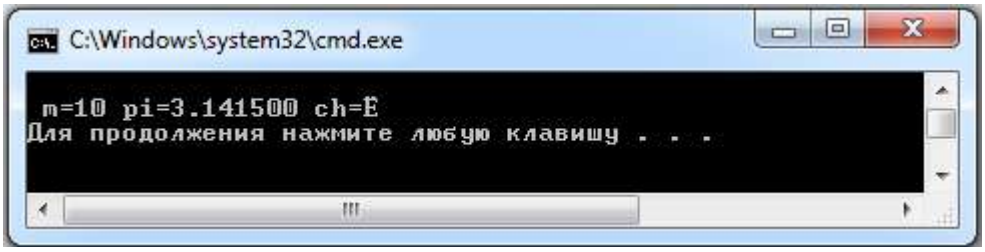
printf(*управляющая строка, список аргументов, ...*)

Управляющая строка содержит информацию, выводимую на экран . Места подстановки значений переменных из ***списка аргументов*** обозначены ***спецификаторами данных***. Признаком спецификатора является символ "%". Количество спецификаторов должно совпадать с количеством аргументов, более того, спецификаторы и аргументы должны попарно соответствовать друг другу.

Как и любая функция языка С, функция `printf()` возвращает значение. Это значение равно числу выведенных символов, если при выводе произошла ошибка – функция возвращает отрицательное число.

Пример.

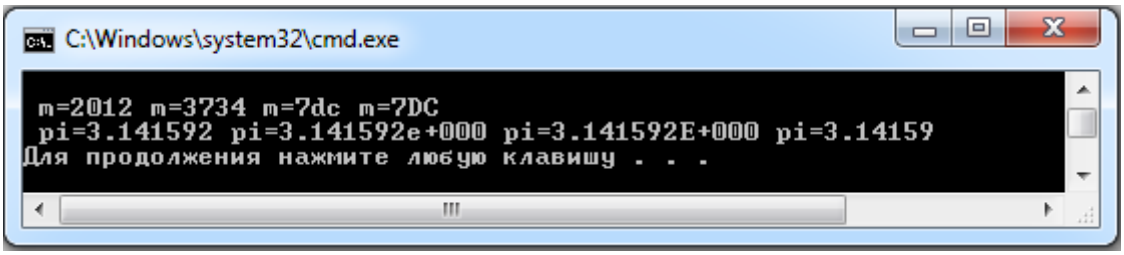
```
setlocale(LC_CTYPE, "rus");//русификация консоли
int m=10; double pi=3.1415; char ch='Ё';
printf("\n m=%d pi=%f ch=%c \n",m,pi,ch);
```



Для каждого встроенного типа данных имеется отдельный спецификатор: `%c` – для типа `char`, `%d` – для целых чисел, `%e`, `%E`, `%f`, `%g` – для чисел с плавающей точкой (два первых формата отображают числа в научной нотации, а последний, в зависимости какой формат окажется короче), `%o` – для вывода чисел без знака в восьмеричной системе счисления, `%x`, `%X` – для вывода чисел без знака в шестнадцатеричной системе счисления, `%p` – для вывода значений указателей (т.е. адресов памяти), `%s` – для строк.

Пример.

```
int m=2012; double pi=3.141592;
printf("\n m=%d m=%o m=%x m=%X",m,m,m,m);
printf("\n pi=%f pi=%e pi=%E pi=%g \n",pi,pi,pi,pi);
```

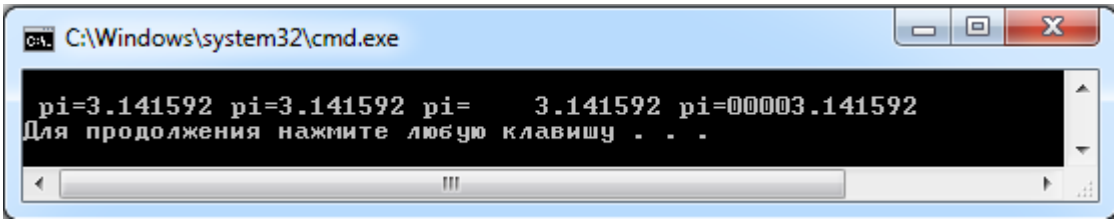


Количество выводимых значений можно уточнить, используя модификаторы формата.

Модификатор минимальной ширины поля указывается между знаком `%` и символом формата, если выводимое число оказывается больше, чем отведенное для него место, то число все равно будет выведено полностью, если же поле окажется шире, то лишние позиции будут дополнены пробелами или нулями, если перед модификатором ширины указан символ “0”.

Пример.

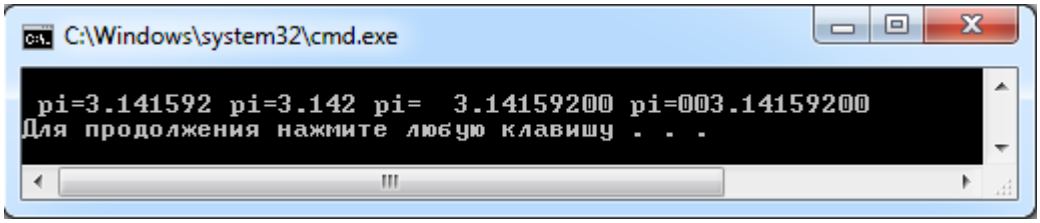
```
double pi=3.141592;
printf("\n pi=%f pi=%2f pi=%12f pi=%012f",pi,pi,pi,pi);
```



Модификатор точности определяет количество значащих цифр. Этот модификатор размещается после модификатора минимальной ширины поля и отделяется точкой.

Пример.

```
double pi=3.141592;
printf("\n pi=%f pi=%.3f pi=%12.8f pi=%012.8f",
           pi,pi,pi,pi);
```



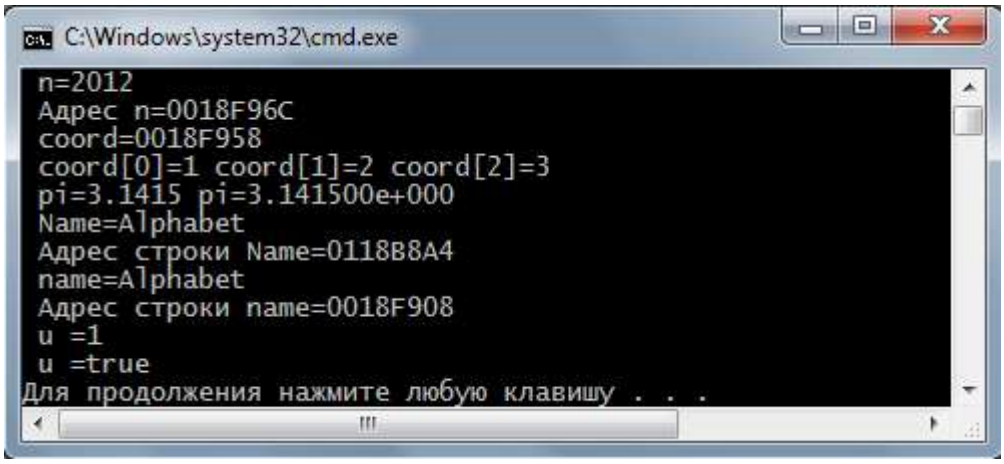
В строке форматирования управлять отображением можно с помощью специальных символов – ***Esc-кодов***: `\n` – переход на новую строку, `\t` – горизонтальная табуляция и др. (см., напр., Дейтел Х. М., Дейтел П. Дж.. *Как программировать на С*.).

Консольный вывод в C++

В языке C++ вывод информации на консоль можно выполнить с помощью объекта `cout` класса `ostream`. Объект `cout` используется вместе с ***оператором вставки*** <<, который умеет работать со всеми базовыми типами языка C++ (подробнее см., напр., Прата С. *Язык программирования C++*). В большинстве уже рассмотренных примеров был использован именно этот способ вывода данных.

Пример.

```
// Вывод данных основных типов
setlocale(LC_CTYPE, "rus"); // русификация консоли
int n=2012;
  cout<<"\n n="<<n;
  cout<<"\n Адрес n="<<&n;
int coord[3]={1,2,3};
  cout<<"\n coord="<<coord<<"\n";
  for(int i=0;i<3;i++)
     cout<<" coord["<<i<<"]=" <<coord[i];
double pi=3.1415;
  cout<<"\n pi="<<pi;
  scientific(cout); // устанавливаем экспоненц.форму
  cout<<" pi="<<pi;
char * Name="Alphabet";
  cout<<"\n Name="<<Name;
  cout<<"\n Адрес строки Name="<<(void *)Name;
string name="Alphabet";
  cout<<"\n name="<<name;
  cout<<"\n Адрес строки name="<<&name;
bool u;
  u=(pi>3) && (pi<4);
  cout<<"\n u ="<<u;
  boolalpha(cout); /* вывод значений в форме true и
false */
  cout<<"\n u ="<<u;
```



С помощью ***манипуляторов***, определенных в стандартной библиотеке языка C++, можно управлять отображением данных. Полный список манипуляторов можно найти в книге Стауструп Б. *Язык программирования C++*.

Пример. Вывод целого числа в различных системах счисления. Установка системы счисления выполняется манипуляторами `dec` (десятичное представление), `hex` (шестнадцатеричное представление) и `oct` (восьмеричное представление). Можно использовать две формы вызова манипулятора, например, установку шестнадцатеричной системы представления чисел можно выполнить в виде

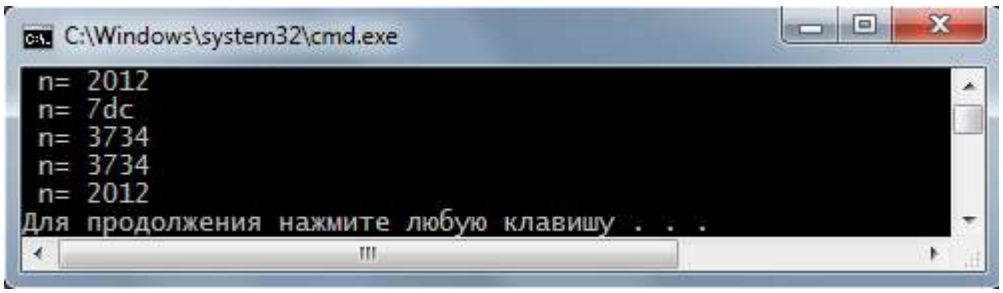
```
hex(cout);
```

и вставкой в `cout`:

```
cout<<"\n n= "<<hex<<n;
```

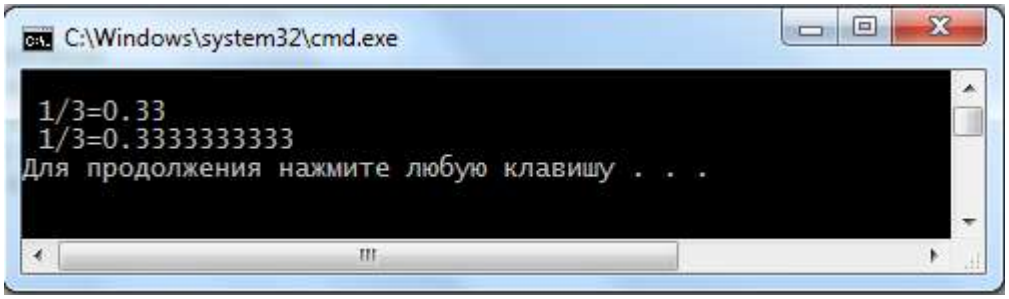
Установка, выполненная манипулятором, сохраняется до следующего явного вызова аналогичного манипулятора.

```
// Целые числа в различных системах счисления

int n=2012;

cout<<"\n n= "<<n; // по умолчание
cout<<"\n n= "<<hex<<n; /* в шестнадцатеричном
представлении */
cout<<"\n n= "<<oct<<n; /* в шестнадцатеричном
представлении */
cout<<"\n n= "<<n; // как в последнем выводе
cout<<"\n n= "<<dec<<n; // в десятичной системе
```



Пример. С помощью манипулятора `precision()` можно установить количество значащих цифр при выводе чисел с плавающей точкой.

```
// Установка точности при выводе

double x=(double)1/3;
cout.precision(2);
cout<<"\n 1/3="<<x;
cout.precision(10);
cout<<"\n 1/3="<<x;
```

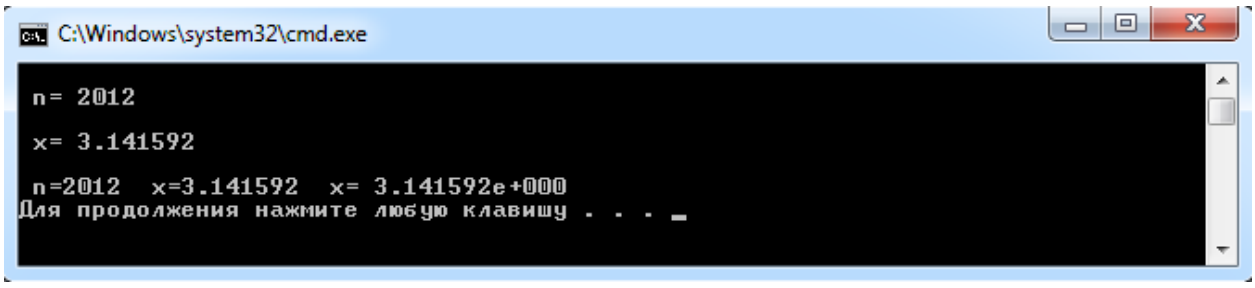


Консольный ввод

Ввод с клавиатуры в языке С выполняется функцией `scanf()`.

Пример. Вводятся целое число и число с плавающей точкой.

```
int n; double x;
printf("\n n= ");
scanf("%d",&n);
printf("\n x= ");
scanf("%lf",&x);
printf("\n n=%d  x=%f  x= %e \n", n, x,x);
```



Синтаксис функции ввода

scanf(*управляющая строка, список аргументов*)

В ***управляющей строке*** содержится описание формата вводимых данных, в ***списке аргументов*** – указатели на переменные, в которых сохраняются введенные данные. Функция `scanf()` возвращает количество введенных данных, в случае ошибки функция возвращает `EOF`.

В ***управляющей строке*** функции `scanf()` используются те же спецификаторы данных, что и в функции `printf()`, отметим, что для ввода значений типа `double` применяется спецификатор `%lf`.

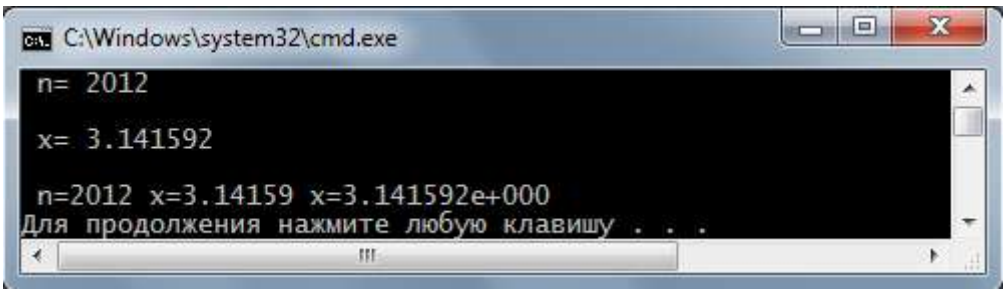
Переменные, которым функция `scanf()` присваивает введеннные значения, передаются в функцию по ссылке, т.е. в списке аргументов указываются адреса этих переменных: `&n`, `&x` и т.д.

Консольный ввод в C++

В языке C++ консольный ввод можно выполнить с помощью объекта `cin` класса `istream`. Этот объект используется вместе с ***оператором извлечения*** `>>`, переопределенным для основных встроенных типов языка (подробнее см., напр., Прата С. *Язык программирования C++*).

Пример. Вводятся целое число и число с плавающей точкой.

```
int n;
cout<<"\n n= ";
cin>>n;
double x;
cout<<"\n x= ";
cin>>x;
cout<<"\n n="<<n<<" x="<<x<<" x="<<scientific<<x;
```



Ввод и вывод в файлы

Завершение работы программы приводит к освобождению памяти – значения всех переменных теряются. Для сохранения информации используются файлы. Файл представляет собой поток байтов. Каждый файл заканчивается маркером конца файла. Когда файл открывается, ему ставится в соответствие поток. Потоки обеспечивают каналы передачи данных между файлами и программами. В начале выполнения программы автоматически

открываются три файла и связанные с ними потоки – это стандартный ввод, стандартный вывод и стандартная ошибка.

Этими потоками можно управлять. Например, если программа имеет имя `myprogr.exe`, то запуск в командной строке

```
>myprog.exe >result.txt
```

перенаправит вывод (по умолчанию, он ориентирован на экран) в файл `result.txt`.

Открытый файл возвращает указатель на структуру `FILE`, определенную в `stdio.h` и содержащую информацию о файле.

Пример. Запись в файл (как принято в С).

```
#include <stdio.h>
FILE *f;
void main()
{
   int i,x;
   f=fopen("c:\\tmp\\result.txt","w");
   for(i=0;i<100;i++)
     {
       x=i*2;
       fprintf(f," %d ",x);
  }
   fclose(f);
}
```

Оператор

```
FILE *f;
```

Объявляет, что переменная `f` является указателем на структуру `FILE`. С каждым файлом необходимо связать отдельную структуру `FILE`.

Открытие файла производится оператором

```
f=fopen("c:\\tmp\\result.txt","w");
```

Функция `fopen()` возвращает указатель на структуру `FILE` открываемого файла, если же файл невозможно открыть, функция возвращает `NULL`. В первом параметре функции указывается имя файла (обращаем внимание на двойные слеши в полном имени файла), а второй параметр указывает режим открытия файла (в данном случае, выбран режим “`w`” – файл для записи).

Оператор

```
fprintf(f," %d ",x);
```

записывает данные в файл, на который указывает первый параметр этой функции – переменная f, в остальном функция полностью аналогична функции printf().

После выполнения операций с передачей данных в файл или из файла, необходимо выполнить закрытие файла с помощью оператора

```
fclose(f);
```

в качества аргумента этой функции используется указатель файла.

Пример. Запись в файл с проверкой дисковой операции (как принято в С). В операторе fprintf() в строке форматирования спецификатор %d записан с пробелом – иначе в результирующем файле получим сплошную запись.

```
#include <stdio.h>
FILE *f;
int main()
{
  int i,x;
  if((f=fopen("result.txt","w"))==NULL){
      printf("File-Error!!!");
      return 1;
 }
  for(i=0;i<10;i++){
       x=i*2;
       fprintf(f," %d ",x);
     }
fclose(f);
return 0;
}
```

Режимы открытия файлов

r – режим чтения данных из файла;

w – режим записи в файл;

a – добавление в конец файла (или создание, если файл не обнаружен);

r+ – открытие файла для обновления (чтение и запись)

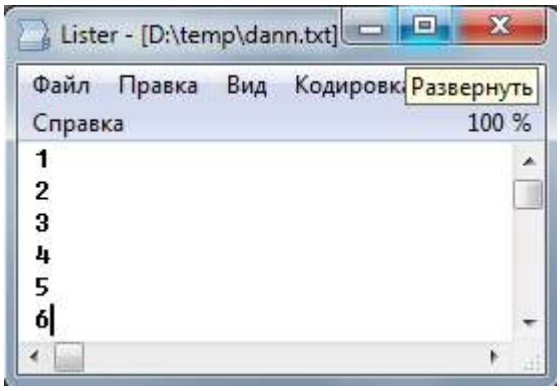
Чтение из файла

Для чтения из файла можно использовать функцию fscanf(). Эта функция является аналогом функции scanf(), но с дополнительным параметром – указателем на файл, из которого будет производиться чтение данных.

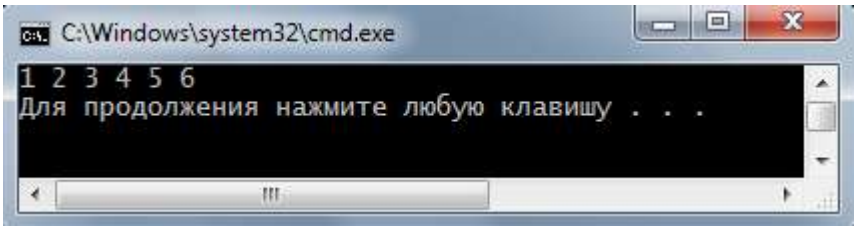
```
FILE *fp;
int n; float x;
```

….
```
fscanf(fp,"%d",&n);
fscanf(fp,"%f",&x);
```
…

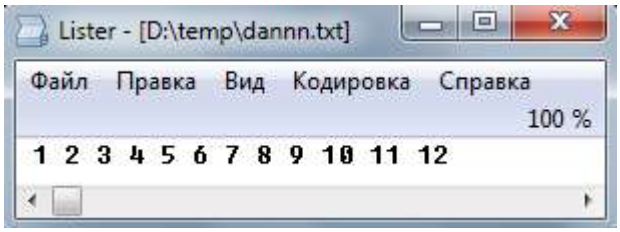
Пример. Чтение данных из файла в массив. Количество данных известно или их в файле больше. Числа записаны в файле по одному на строке (или через пробел).



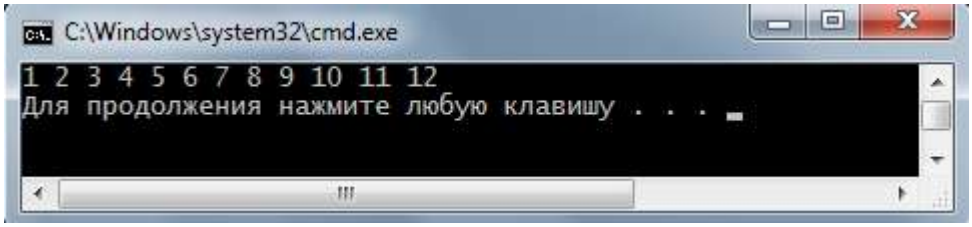

```
// Чтение данных из файла в массив
// количество данных известно или их в файле больше

FILE *fp;
char * fileName="d:\\temp\\dann.txt";
fp=fopen(fileName,"r");
const int n=6;
int mx[n], r;
// чтение из файла
for (int i=0;i<n;i++) {
    fscanf(fp,"%d",&r);
    mx[i]=r;
}
fclose(fp);
// печать:
for (int i=0;i<n;i++) printf("%d ",mx[i]);
```



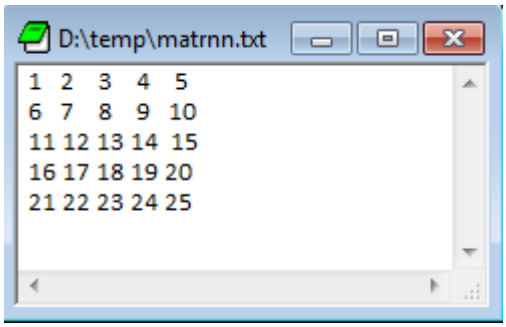
Пример. Чтение данных из файла в массив. Количество данных неизвестно. В программе файл открывается дважды: первый раз, чтобы подсчитать количество элементов, а второй раз – для считывания данных в динамический массив.




```
// Чтение данных из файла в массив
// количество данных неизвестно

FILE *fp;
char * fileName="d:\\temp\\dannn.txt";
fp=fopen(fileName,"r");
// узнаем сколько чисел в файле
int r,n=0;
while (!feof(fp)) // пока не конец файла
{
    fscanf(fp,"%d",&r); n++;
}
fclose(fp); // закрываем файл

int *mx=  new int [n]; // динамический массив из n
элементов

fp=fopen(fileName,"r"); // снова открываем

// чтение из файла
for (int i=0;i<n;i++) {
    fscanf(fp,"%d",&r);
    mx[i]=r;
}
fclose(fp);
// печать:
for (int i=0;i<n;i++) printf("%d ",mx[i]);
```



Пример. Чтение данных из файла в двумерный массив. Количество данных известно или их в файле больше. Числа записаны в файле в виде матрицы.

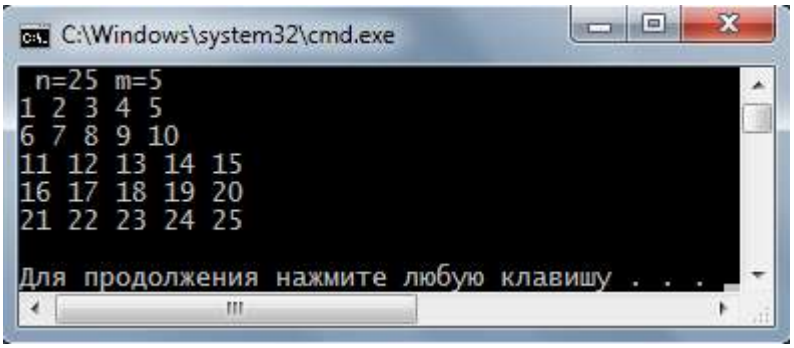



```
// Чтение данных из файла в двумерный массив
// количество данных известно или их в файле больше

FILE *fp;
char * fileName="d:\\temp\\matrnn.txt";
fp=fopen(fileName,"r");
const int n=5;
int ma[n][n], r;
// чтение из файла
for (int i=0;i<n;i++)
  for (int j=0;j<n;j++) {
     fscanf(fp,"%d",&r);
     ma[i][j]=r;
}
fclose(fp);
// печать:
for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++)
      printf("%d ",ma[i][j]);
    printf("\n");
}
```

Пример. Чтение данных из файла в двумерный массив. Количество данных неизвестно. В программе файл открывается дважды: первый раз, чтобы подсчитать количество элементов, а затем – для считывания данных в двумерный динамический массив.

```
// Чтение данных из файла в массив
// количество данных неизвестно

FILE *fp;
char * fileName="d:\\temp\\matrnn.txt";
fp=fopen(fileName,"r");
// узнаем сколько чисел в файле
int n=0;
int r;
while (!feof(fp)) // пока не конец файла
{
```

```
    fscanf(fp,"%d",&r); n++;
}
fclose(fp); // закрываем файл

// вычисляем размер матрицы
int m = (int)sqrt((double)n);
printf("\n n=%d m=%d",n,m);
// динамический массив из m*m элементов:
int **mx=  new int* [m];
for (int i=0;i<m;i++) mx[i]=  new int [m];

fp=fopen(fileName,"r"); // снова открываем файл

// чтение из файла
for (int i=0;i<m;i++)
 for (int j=0;j<m;j++)
   {
    fscanf(fp,"%d",&r);
    mx[i][j]=r;
   }
fclose(fp);
for (int i=0;i<m;i++) {
    for (int j=0;j<m;j++)
      printf("%d ",mx[i][j]);
    printf("\n");
}
```



Пример. Чтение из файла. Функция `isalpha(ch)` возвращает истинное значение, если параметр ch является символом алфавита.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <iostream>
using namespace std;
FILE *f;
int get_word(char *);
int main()
```

```
{  char word[80];/* слово, выделенное из файла */
   int kol=0;  /* количество слов в в файле */
   if((f=fopen("readme.txt","r"))==NULL){printf("File-
Error!!!"); return 1;}
   while (get_word(word))
    {/* цикл продолжается пока get_word() не равно 0*/
     kol++;  cout<<word<<' ';
    }
   fclose(f);
   return 0;
}
int get_word(char *a)
{   /* возвращает 1, если из файла прочитано слово*/
    /* слово передается через параметр a*/
    char ch, i=1;
    while (!isalpha(ch=getc(f))&&ch!=EOF);
    if(ch==EOF) return 0;/* файл кончился*/
    a[0]=ch;
    while(isalpha(ch=getc(f))&&ch!=EOF) a[i++]=ch;
    a[i]='\0';
    return 1;
}
```

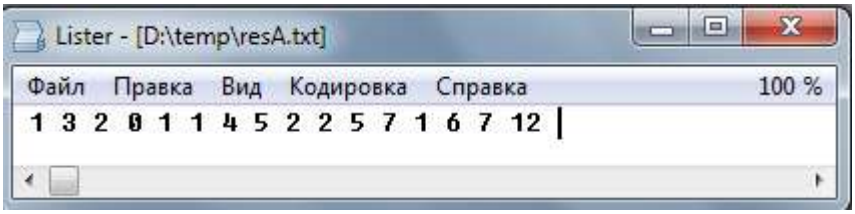
Запись в файл

Для записи в файл можно использовать функцию `fprintf()`. В отличие от функции `printf()`, аналогом которой она является, присутствует еще один параметр – указатель на файл, в который будет производиться запись.

Пример. Запись в файл матрицы целых чисел. В программе числа созданы с помощью генератора случайных чисел.

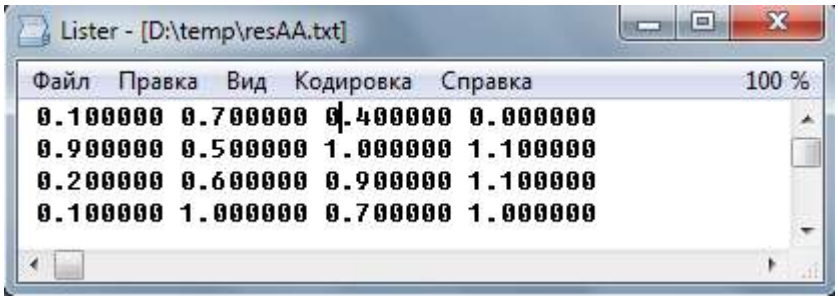
```
// Запись матрицы в файл
//
FILE *fp;
char * fileName="d:\\temp\\resA.txt";
fp=fopen(fileName,"w");
// заполнение массива числами
const int n=4;
int a[n][n];
for (int i=0;i<n;i++)
  for (int j=0;j<n;j++)
    a[i][j]= (rand() % n) +i*j;
// запись в файл
for (int i=0;i<n;i++)
```

```
   for (int j=0;j<n;j++)
    fprintf(fp,"%d ",a[i][j]);
fclose(fp);
```



Пример. Запись в файл матрицы чисел с плавающей точкой. Числа создаются с помощью генератора случайных чисел. Запись в файл производится построчно – каждая строка матрицы с новой строки.

```
/* Запись матрицы в файл.
   Каждая строка матрицы - перевод на новую строку в
файле */
//
FILE *fp;
char * fileName="d:\\temp\\resAA.txt";
fp=fopen(fileName,"w");
// заполнение массива числами
const int n=4;
double a[n][n];
for (int i=0;i<n;i++)
  for (int j=0;j<n;j++)
    a[i][j]= ((rand() % 10) +i*j)*0.1;
// запись в файл
for (int i=0;i<n;i++) {
   for (int j=0;j<n;j++)
    fprintf(fp,"%f ",a[i][j]);
    // или fprintf(fp,"%e ",a[i][j]);
fprintf(fp,"\n");
}
fclose(fp);
```



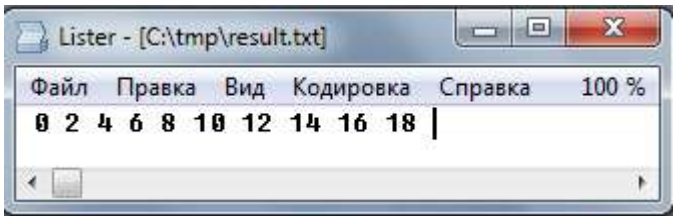
Файловый ввод и вывод в C++

Для работы с файлами в C++, прежде всего, необходимо включить в программу заголовочный файл `fstream.h`. Этот файл содержит описание

классов `istream` (файловый ввод) и `ofstream` (файловый вывод). Все операции с файлами осуществляются через объекты этих классов вызовом соответствующих методов.

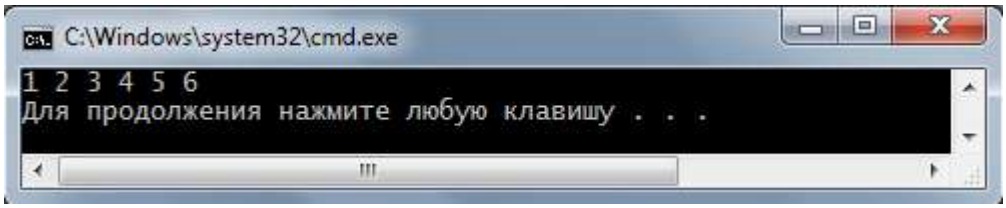
Пример. Этот пример является аналогом примера, приведенного ранее для иллюстрации работы `fprintf()`.

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

int main()
{
 // запись в файл
 ofstream fout("c:\\tmp\\result.txt");
 if(!fout){
     cout<<"\n Нельзя создать файл";
     return 1;
 }
 int x;
 for (int i=0;i<10;i++){
       x=i*2;
       fout<<x<<" ";
    }
 fout.close();
}
```



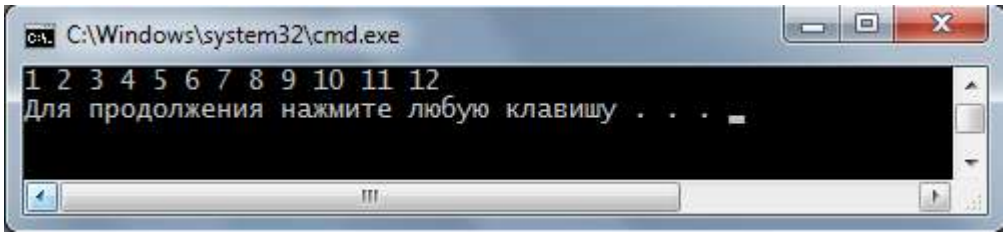
Пример. Чтение данных из файла в массив. Количество данных известно.

```
char * fileName="d:\\temp\\dann.txt";
ifstream fin(fileName); // открытие файла для ввода
const int n=6;
int mx[n];
// чтение из файла
for (int i=0;i<n;i++) fin>>mx[i];
fin.close();
// печать:
for (int i=0;i<n;i++) cout<<mx[i]<<" ";
cout<<"\n";
```



Пример. Чтение данных из файла в массив. Количество данных неизвестно. Ранее, в одном из примеров, эта операция была выполнена с помощью функций `fscanf()` и `feof()`. Файл открывается дважды: сначала для вычисления количество элементов в файле, а затем – для считывания данных в динамический массив.

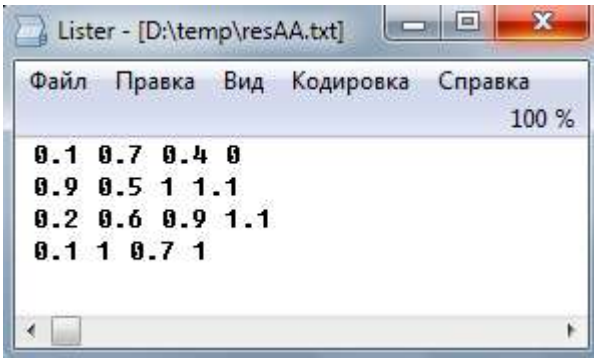
```
char * fileName="d:\\temp\\dannn.txt";
ifstream fin(fileName); // открытие файла
// узнаем сколько чисел в файле
int r,n=0;
while (!fin.eof()) // пока не конец файла
{
    fin>>r; n++;
}
fin.close(); // закрываем файл
//
int *mx=  new int [n]; // динамический массив
ifstream fin2(fileName);  // снова открываем файл
// чтение из файла
for (int i=0;i<n;i++)     fin2>>mx[i];
fin2.close(); // закрываем файл
// печать:
for (int i=0;i<n;i++) cout<<mx[i]<<" ";
```



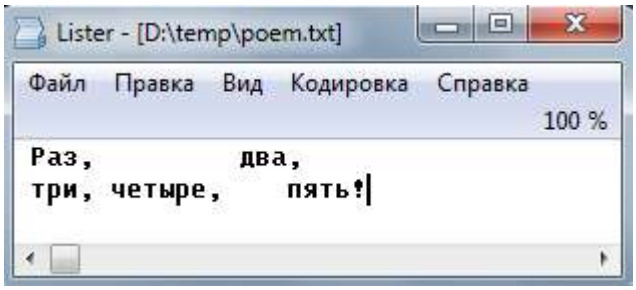
Пример. Запись в файл матрицы чисел с плавающей точкой. Числа создаются с помощью генератора случайных чисел. Запись в файл производится построчно – каждая строка матрицы с новой строки.

```
// Запись матрицы в файл
//
char * fileName="d:\\temp\\resAA.txt";
 ofstream fout(fileName);
// заполнение массива числами
const int n=4;
double a[n][n];
for (int i=0;i<n;i++)
```

```
  for (int j=0;j<n;j++)
    a[i][j]= ((rand() % 10) +i*j)*0.1;
// запись в файл
for (int i=0;i<n;i++) {
   for (int j=0;j<n;j++)
         fout<<a[i][j]<<" ";
   fout<<"\n";
}
fout.close();
```

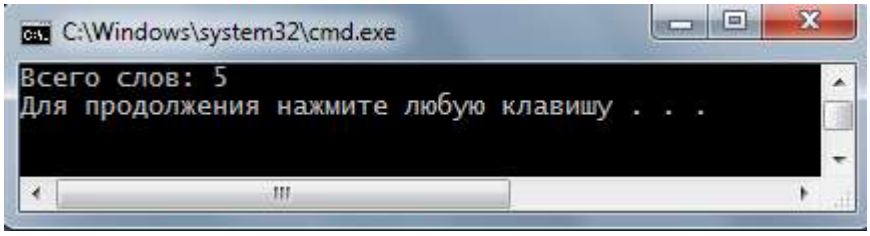


Пример. Подсчет числа слов в текстовом файле.




```
// Подсчет числа слов в текстовом файле
//
   setlocale(LC_CTYPE, "rus"); // русификация консоли
   ifstream in("d:\\temp\\poem.txt");
   if(!in) {
         cout << "\n Ошибка открытия файла";
         return 1; }
   int count = 0;   unsigned char ch;
   in >> ch; // нахождение первого символа не пробела
   in.unsetf(ios::skipws); // не пропускать пробелы
   while(!in.eof()) {
     in >> ch;
     if(isspace(ch)|| in.eof()) {
        count++;
       while(isspace(ch) && !in.eof()) in >> ch;/*
пропускаем пробелы */
      }
    }
```

```
cout << "Всего слов: " << count << '\n';
in.close();
```



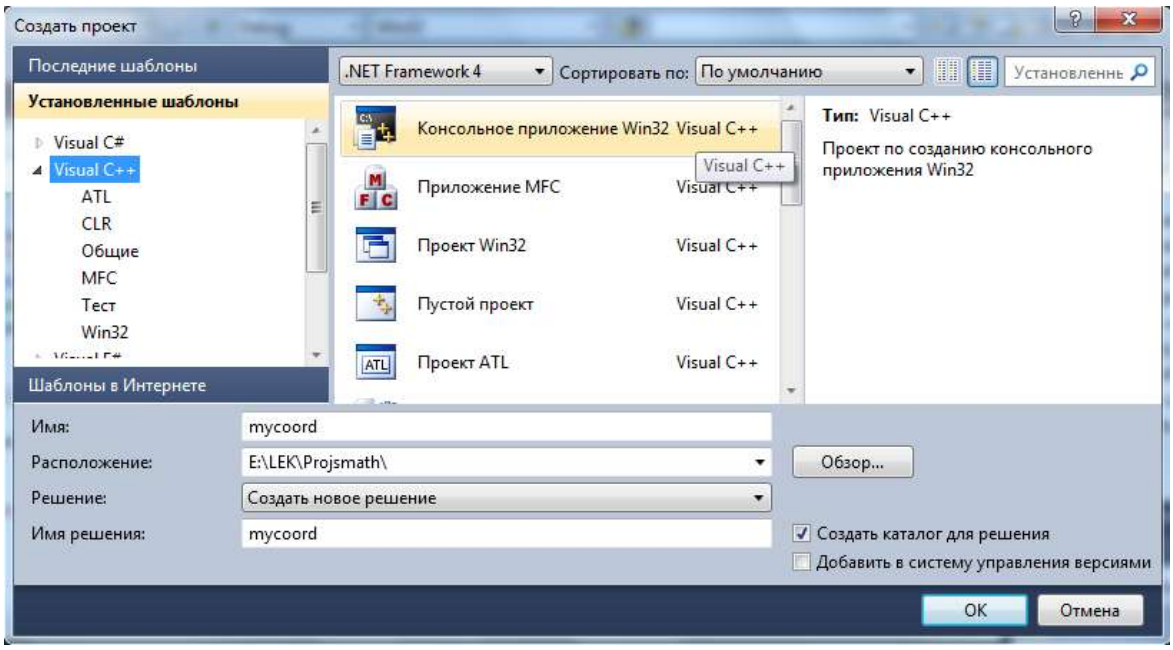
Раздельная компиляция

Раздельная компиляция применяется для разделения программы на несколько файлов с возможностью отдельной компиляции каждого из них и объединением в одно целое (например, `exe`–файл) на заключительном этапе (подробнее, см., напр., Прата С., Язык программирования С++, Стауструп Б. *Язык программирования С++*).

На примере небольшого проекта рассмотрим процесс создания многофайлового проекта. В программе вводятся координаты точки (числа `x` и `y`) и вычисляются: расстояние от начала координат и угол с координатной осью (полярные координаты). Проект будет состоять из трех файлов `mycoord.cpp`, `mytypes.h` и `myfunctions.cpp`. В файле `mycoord.cpp`, содержащем функцию `main()`, выполняется ввод координат, вызов необходимых функций вычисления и отображение результатов. В файл `mytypes.h` будут помещены объявления типов `struct polar` – для полярных координат и `struct rect` – для декартовых. Прототипы функций также разместим в этом файле. Определения функций запишем в отдельный файл с именем `myfunctions.cpp`.

В MS Visual Studio 2010 процесс создания многофайлового проекта состоит из нескольких шагов.

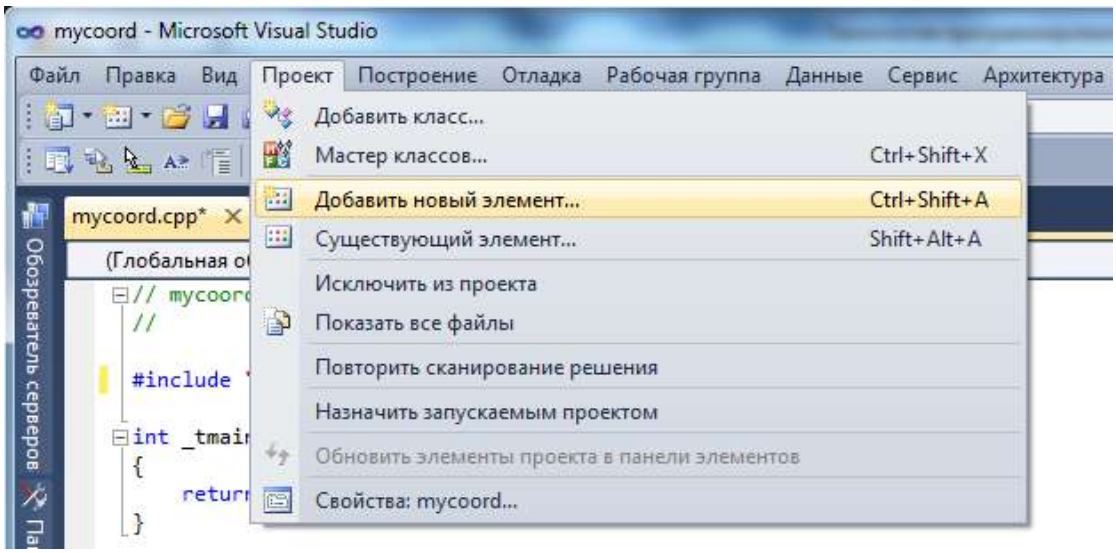
На первом шаге создаем проект с именем `mycoord`.



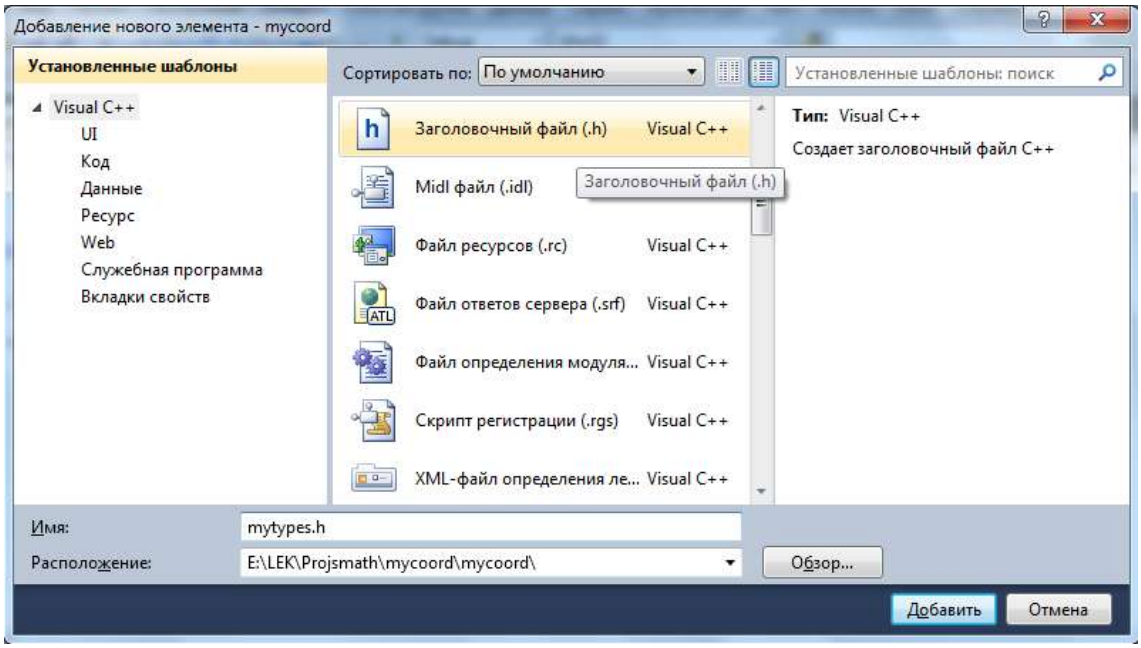
Среда программирования генерирует шаблон консольного приложения

```
// mycoord.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Создаем заголовочный файл `mytypes.h`, в который записываем объявления типов. Для этого в меню `Проект (Project)` выбираем раздел `Добавить новый элемент…. (Add New Item…)`



В окне диалога выбираем элемент Заголовочный файл (.h) (Header File (.h)), вводим имя заголовочного файла mytypes.h и заканчиваем диалог нажатием кнопки Добавить.



Набираем определения типов

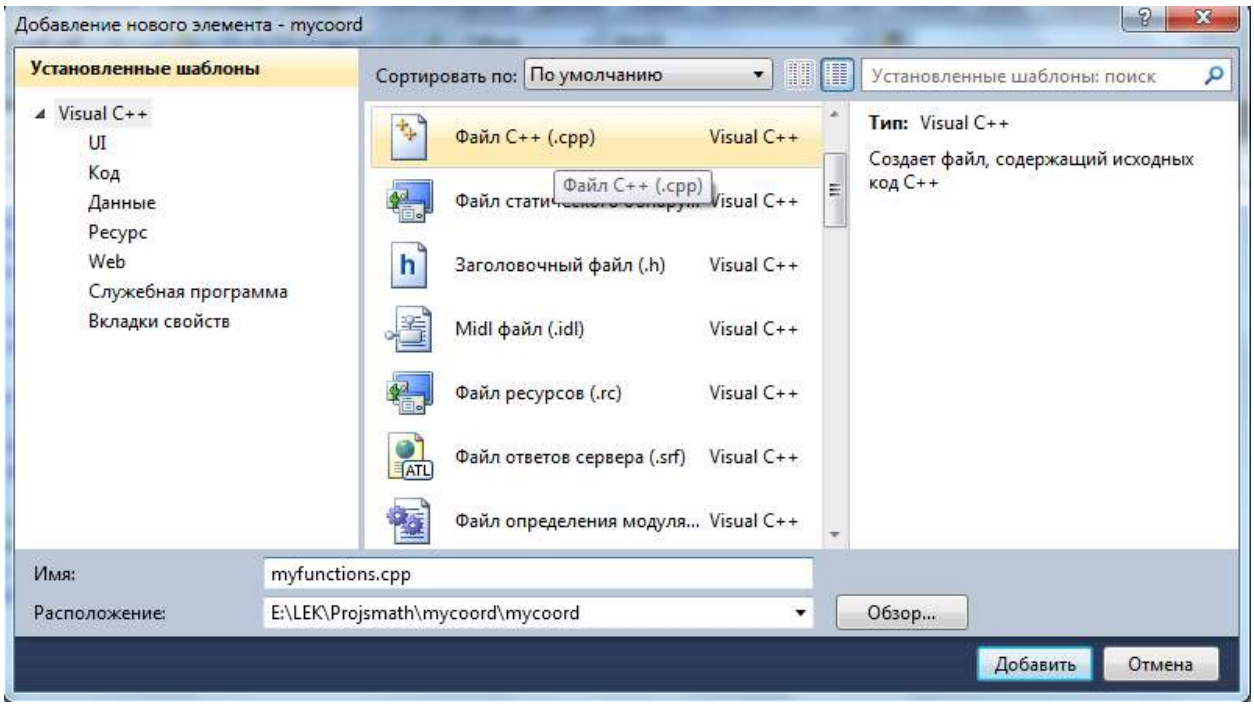
```
// mytypes.h
#ifndef MYTYPES_H_
#define MYTYPES_H_

struct polar
{
    double distance;// расстояние от начала координат
    double angle;  // угол
};

struct rect
{
    double x;     // расстояние по горизонтали
    double y;     // расстояние по вертикали
};

// прототипы функций:
polar rect_to_polar(rect xy);
void show_polar(polar da);
#endif
```

Обращаем внимание на стражи включения

```
#ifndef MYTYPES_H_
#define MYTYPES_H_
      ….

#endif
```

Эта конструкция уже описывалась в разделе “Директивы препроцессора”. Заголовочный файл следует включать в проект только один раз. Если какие-то заголовочные файлы включают (с помощью `#include`) другие заголовочные файлы, то это правило трудно проконтролировать. Стандартная методика предотвращения многократных включений заголовочных файлов, основана на использовании *стражей включения* – директив препроцессора `#ifndef` и `#endif`.

На следующем шаге создаем файл с кодом, реализующим функции, объявленные в заголовочном файле. Для этого снова вызываем диалог добавления нового элемента с помощью меню `Проект (Project)` выбираем раздел `Добавить новый элемент…. (Add New Item…)`, но на этот раз выбираем элемент `Файл C++ (.cpp)`, а затем вводим имя файла `myfunctions.cpp`



и набираем код:

```
#include "stdafx.h"
#include <iostream>
#include <cmath>
#include "mytypes.h"
using namespace std;

polar rect_to_polar(rect xy)
{
  polar coord;
  coord.distance = sqrt(xy.x * xy.x + xy.y * xy.y);
  coord.angle = atan2(xy.y, xy.x);
  return coord;
}
```

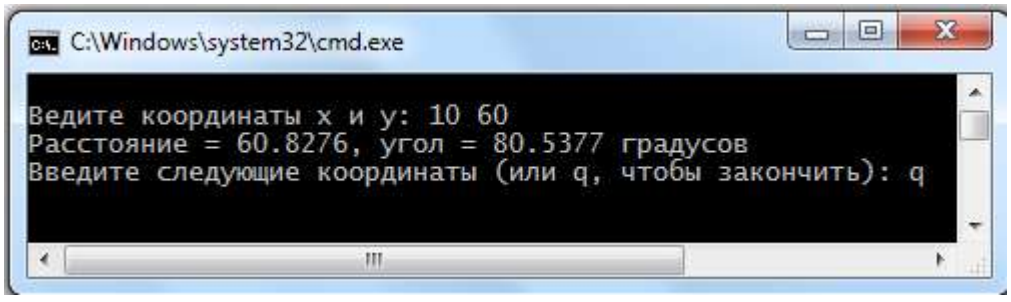
```
void show_polar (polar dapos)
{
    const double Rad_to_deg = 57.29577951;
    cout << "Расстояние = " << dapos.distance;
    cout << ", угол = " << dapos.angle * Rad_to_deg;
    cout << " градусов\n";
}
```

Теперь переключаемся в окно файла `mycoord.cpp` добавляем

```
#include "mytypes.h"
#include <iostream>
using namespace std;
```

а в функции `main()` записываем код обработки данных:

```
#include "stdafx.h"
#include "mytypes.h"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  rect rplace;
  polar pplace;
  cout << "\nВедите координаты x и y: ";
  while (cin >> rplace.x >> rplace.y)
   {
     pplace = rect_to_polar(rplace);
     show_polar(pplace);
     cout << "Введите следующие координаты (или q,
чтобы закончить): ";
   }
  return 0;
}
```



Что включать в заголовочный файл

Во всех объявлениях типы одних и тех функций, классов и т. д. должны быть согласованы. Обычно, для достижения согласованности объявлений в различных единицах трансляции, используются заголовочные файлы (h–файлы), которые включаются в файлы проекта с помощью директивы препроцессора `#include`. В книге Стауструп Б. *Язык программирования C++* приведено “негласное практическое правило”, указывающее какие инструкции следует включать в заголовочные файлы. Процитируем это правило. Негласное практическое правило гласит, что заголовочный файл может содержать: именованные пространства имен (`namespace N{/* */}`), определения типов (`struct Point {int x, y;};`), объявления шаблонов (`template<class T> class Z;`), объявления функций (`extern int strlen(const char*);`), определения встроенных функций (`inline char get(char* p) {return *p++;}`), объявления данных (`extern int a;`), определения констант (`const float pi=3.1415;`), перечисления (`enum Light {red, yellow, green};`), объявления имен (`class Matrix;`), директивы включения (`#include <algorithm>`), макроопределения (`#define VERSION 12`), директивы условной компиляции (`#ifdef __cplusplus`), комментарии (`/* */`).

Это практическое правило не является требованием языка. С другой стороны, заголовочный файл никогда не должен содержать: определения обычных функций (`char get(char* p){return *p++;}`), определения данных (`int a;`), определения агрегатов (`short tbl[]={1,2,3};`), неименованные пространства имен (`namespace {/* */}`), экспортируемые определения шаблонов (`export template<class T> f(T t){/* .*/}`).

Области действия идентификаторов

Область действия идентификатора (диапазон доступа, область видимости) определяет, в каких частях программы этот идентификатор является доступным. Так, переменную, определенную в функции, можно использовать только в этой функции. Если же переменная определена до определений функций (в том числе и до функции `main()`), то её можно использовать во всех функциях данного файла.

Различают следующие области действия имен: блоки (т.е. группа операторов, ограниченная фигурными скобками), функции, классы, файлы, пространства имен, программа.

Переменная, объявленная в блоке, известна только в пределах этого блока. Такую переменную обычно называют *локальной переменной*.

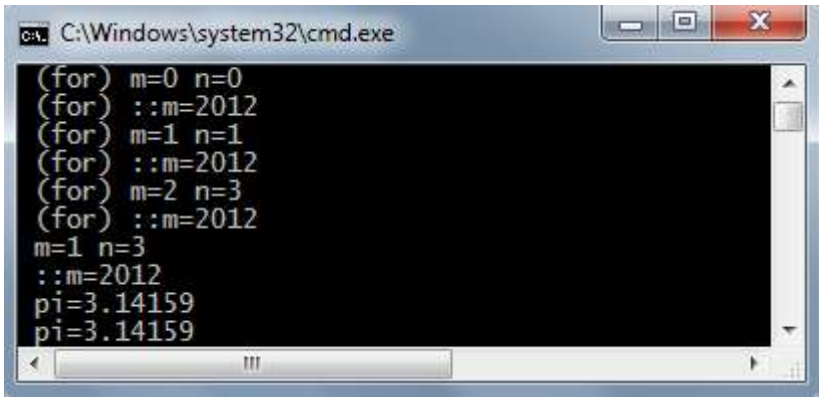
Переменную, объявленную в начале программы, прежде определений

функций (но необязательно до объявлений функций – прототипов) называют *глобальной переменной*, поскольку она известна во всем файле.

Имя должно быть уникальным в пределах своей области действия, т.е. не должно быть *конфликта имен*. Однако в различных областях действия могут быть объявлены переменные с одинаковыми именами. Если же объявить переменную с именем, совпадающим с именем глобальной переменной, – что допустимо, – то локальная переменная "перекроет" глобальную. *Операция глобальной области действия* (или оператор глобального разрешения) "`::`" позволяет обращаться к глобальной переменной из вложенной области действия.

Пример. В программе три различных переменных с именем `m` – глобальная переменная и две локальных переменных, одна из которых доступна только в цикле `for`. С помощью операции "`::`" выполнено обращение к глобальной переменной `m`. Для глобальной константы `pi` применять оператор глобального разрешения не обязательно – это имя уникально в пределах всей программы. В цикле `for` нельзя получить доступ к другой переменной `m`, объявленной в `main()`, но с помощью операции "`::`" можно работать с глобальной переменной `m`.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
// глобальные переменные
const double pi=3.141592;
int m=2012;

int _tmain(int argc, _TCHAR* argv[])
{
 int n=0, m=1; // видимость только в main()
 for (int m=0;m<3;m++) // m видна только в for
  {
    n+=m;
    cout<<"\n (for) m="<<m<<" n="<<n;
    cout<<"\n (for) ::m="<<::m; /* вызов глобальной
переменной */
  }
  cout<<"\n m="<<m<<" n="<<n;
  cout<<"\n ::m="<<::m; // вызов глобальной переменной
  cout<<"\n pi="<<pi; // вызов глобальной переменной
  cout<<"\n pi="<<::pi; // и еще один вызов
  return 0;
}
```



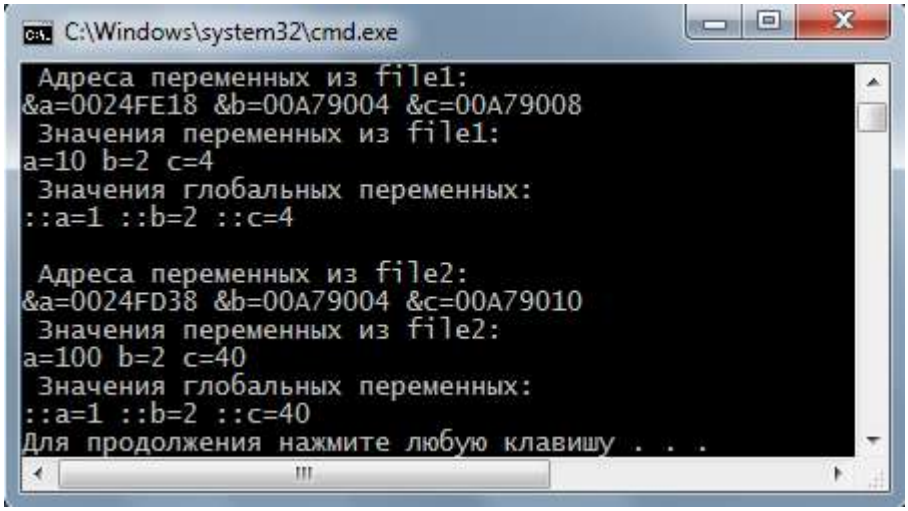
Связывание

Связывание определяет способ использования имени в различных файлах проекта. Имя с *внешним связыванием* можно использовать разными файлами, а имя с *внутренним связыванием* только функциями одного файла.

Пример. Внешнее и внутреннее связывание. В многофайловом проекте внешнюю переменную можно объявить только в одном файле. Во всех остальных файлах, где применяется эта переменная, она должна быть объявлена с ключевым словом **`extern`**.

```
// file1.cpp:
//
#include "stdafx.h"
#include <iostream>
using namespace std;
//Глобальные переменные:
int a=1;            //определение внешней переменной
int b=2;          //определение внешней переменной
static int c=4;//т.к. static, то это внутренняя
переменная
void func_file2(); // прототип функции из другого файла

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_CTYPE, "rus"); // русификация консоли
    int a=10;// локальная переменная
    cout<<"\n Адреса переменных из file1:\n";
    cout<<"&a="<<&a<<" &b="<<&b<<" &c="<<&c;
    cout<<"\n Значения переменных из file1:\n";
    cout<<"a="<<a<<" b="<<b<<" c="<<c;
    cout<<"\n Значения глобальных переменных:\n";
    cout<<"::a="<<::a<<" ::b="<<::b<<" ::c="<<::c;
    cout<<"\n";
    func_file2();
    cout<<"\n";
    return 0;
```

```
}

// file2.cpp:
//
#include "stdafx.h"
#include <iostream>
using namespace std;
extern int a;  // переменная из другого файла
extern int b;  // переменная из другого файла
static int c=40;//т.к. static, то это внутренняя
переменная
void func_file2() // определение функции
{
     int a=100; // локальная переменная
     cout<<"\n Адреса переменных из file2:\n";
     cout<<"&a="<<&a<<" &b="<<&b<<" &c="<<&c;
     cout<<"\n Значения переменных из file2:\n";
     cout<<"a="<<a<<" b="<<b<<" c="<<c;
     cout<<"\n Значения глобальных переменных:\n";
     cout<<"::a="<<::a<<" ::b="<<::b<<" ::c="<<::c;
}
```



Пространства имен

Технология *пространства имён* (*Namespace*) используется в большинстве современных языков программирования. Пространство имён группирует некоторое множество идентификаторов и используется как средство исключения конфликта имён.

Пространства имён можно рассматривать как механизм отражения логического группирования (см., напр., Стауструп Б. *Язык программирования C++*).

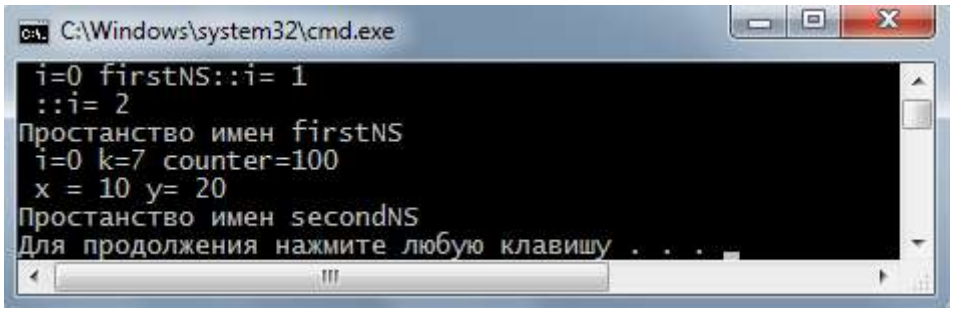
Пространства имён определяются либо на глобальном уровне, либо внутри других пространств имён. Нельзя создавать пространства имён в

блоке. Имя, объявленное в пространстве имён, обладает внешним связыванием.

Пример. Демонстрация пространства имён. В программе имеется глобальная переменная с именем `i`, локальная переменная с тем же именем, а также переменная `i` в пространстве имён `firstNS`. Символьные массивы с именем `str` определены как в пространстве имён `firstNS`, так и в `secondNS`. Все эти имена используются в программе. В самом начале программы с помощью директивы `using namespace firstNS;` импортированы все имена из пространства имен `firstNS`. Поэтому в `main()` обращение `str` относится к переменной из пространства имён `firstNS`. Однако запись `i` означает обращение к локальной переменной – локальные переменные "сильнее" остальных переменных с тем же именем. *Квалифицированные имена* (т.е. с явным указанием пространства имён) позволяют добраться до переменных из различных пространств имён. Для вызова глобальной переменной используется запись `::i`, а обращение `firstNS::i` обозначает доступ к переменной из пространства имён `firstNS`.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
// определение первого пространства имен
namespace firstNS {
    int i, k, counter;
    char str[] = "Простанство имен firstNS\n";
}
// определение второго пространства имен
namespace secondNS {
   int x, y;
   char str[] = "Простанство имен secondNS\n";
}

int i=2;// глобальная переменная

using namespace firstNS; /* подключаем все имена из
firstNS */

int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  int i=0;
  firstNS::i=1;
  cout<<" i="<<i<<" firstNS::i= "<<firstNS::i<<'\n';
  cout<<" ::i= "<<::i<<'\n';
```

```
  cout << str;
  k=7;
  counter=100;
  cout<<" i="<<i<<" k="<<k<<" counter="<<counter<<'\n';
  // используем пространство имен secondNS
  secondNS::x = 10;
  secondNS::y = 20;
  cout << " x = " << secondNS::x;
  cout << " y= " << secondNS::y << endl;
  // импорт имени str из  пространства имен secondNS:
  using  secondNS::str;
  cout << str;
  return 0;
}
```



Подключение имен из пространства имён осуществляется с помощью двух механизмов – `using`–объявления и `using`–директивы. При обращении к переменной можно явно указать пространство имён, используя операцию "`::`".

Для импорта всех имен из пространства имён в программу используется `using`–директивы

`using namespace  namespace-name`

Все программы этого руководства содержат строки

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

с помощью которых подключаются функции стандартной библиотеки. Если убрать эту `using`–директиве, то вместо привычных `cin` и `cout` пришлось бы явно указывать принадлежность имён к пространству `std` и записывать `std::cin` и `std::cout`.

Пример. Явное задание используемого пространства имен.

```
#include <iostream>
int main()
{
   double val;
   std::cout << "Vvodim chislo: ";
```

```
    std::cin >> val;
    val*=100;
    std::cout << "Poluchili: ";
    std::cout << val;
    return 0;
}
```

Пример. Введение в глобальное пространство только нескольких имен.

```
#include <iostream>
// обеспечение доступа к потокам cin и cout
using std::cout;
using std::cin;
int main()
{
   double val;
    cout << "Vvodim chislo: ";   cin >> val;
    val *=100;
    cout << "Poluchili: ";
    cout << val;
    return 0;
}
```

Объявление `using` вводит локальный синоним для отдельного имени, например,

```
using Sunday::str;
```

Далее в программе для доступа к переменной `Sunday::str` можно использовать запись `str`.

Такие синонимы следует делать как можно более локальными во избежание конфликтов имён (см., напр., Стауструп Б. *Язык программирования C++*). Например, в следующем примере, объявление

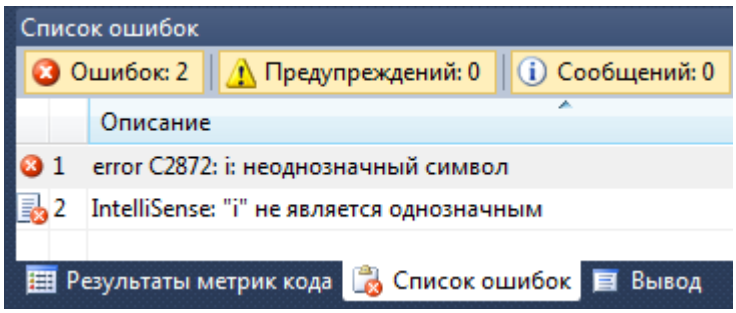
```
using Saturday::str;
```

приведет к ошибке (повторное объявление переменной). Устранить ошибку можно только путем включения этого объявления, а также операторов, использующих `str`, в блок с помощью фигурных скобок.

Если одно и то же имя определено как в пространстве имён, так и в области объявлений программы, то при попытке применить `using`–объявление оба имени вступят в конфликт, и будет выдано сообщение об ошибке. Если же использовать `using`–директиву, то локальная версия переменной перекроет версию из пространства имён.

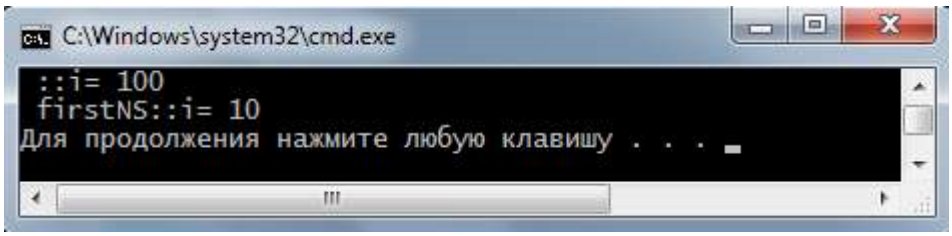
Пример. Переменная `i` объявлена в пространстве имён `firstNS` и в области глобальных переменных – это привело к конфликту имён.

```
#include <iostream>
using namespace std;
// определение пространства имен
namespace firstNS {
    int i=10;
}
int i=100;
using namespace firstNS;
int main()
{
     cout<<" i= "<<i<<'\n';
    return 0;
}
```



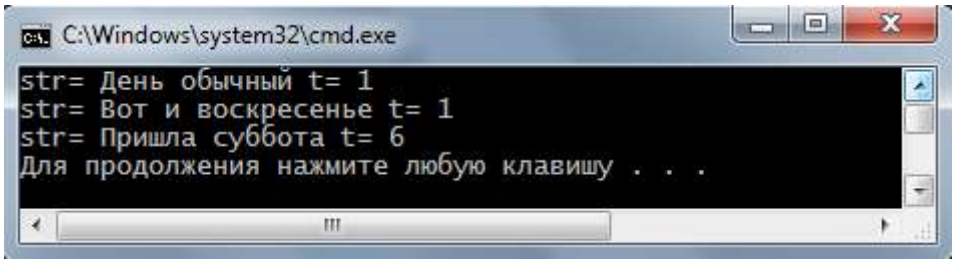
Конечно же, использование квалифицированных имён решит проблему.

```
#include <iostream>
using namespace std;
// определение пространства имен
namespace firstNS {
    int i=10;
}
int i=100;
using namespace firstNS;

int main()
{
    cout<<" ::i= "<<::i<<'\n';
    cout<<" firstNS::i= "<<firstNS::i<<'\n';
    return 0;
}
```



Пример. Три пространства имён содержат переменные с одинаковыми именами. С помощью `using`-директивы импортируются имена одного из пространств имён – `Week`. Поэтому первый оператор вывода обращается к переменным из этого пространства. Далее, с помощью `using`-объявления в локальную область объявлений добавляется имя `Sunday::str` и, поэтому следующий оператор `cout` рассматривает `str` как `Sunday::str`, но переменную `t` как `Week::t`.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
namespace Sunday {
    char str[] = "Вот и воскресенье";
    int t=7;
}
namespace Saturday {
    char str[] = "Пришла суббота";
    int t=6;
}
namespace Week {
    char str[] = "День обычный";
    int t=1;
}
using namespace Week;

int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_CTYPE, "rus"); // русификация консоли
  cout << "str= "<<str<<" t= "<<t;
  using Sunday::str;
  cout << "str= "<<str<<" t= "<<t;
// using Saturday::str;
   cout<<"str="<<Saturday::str<<" t="<<Saturday::t;
    return 0;
}
```

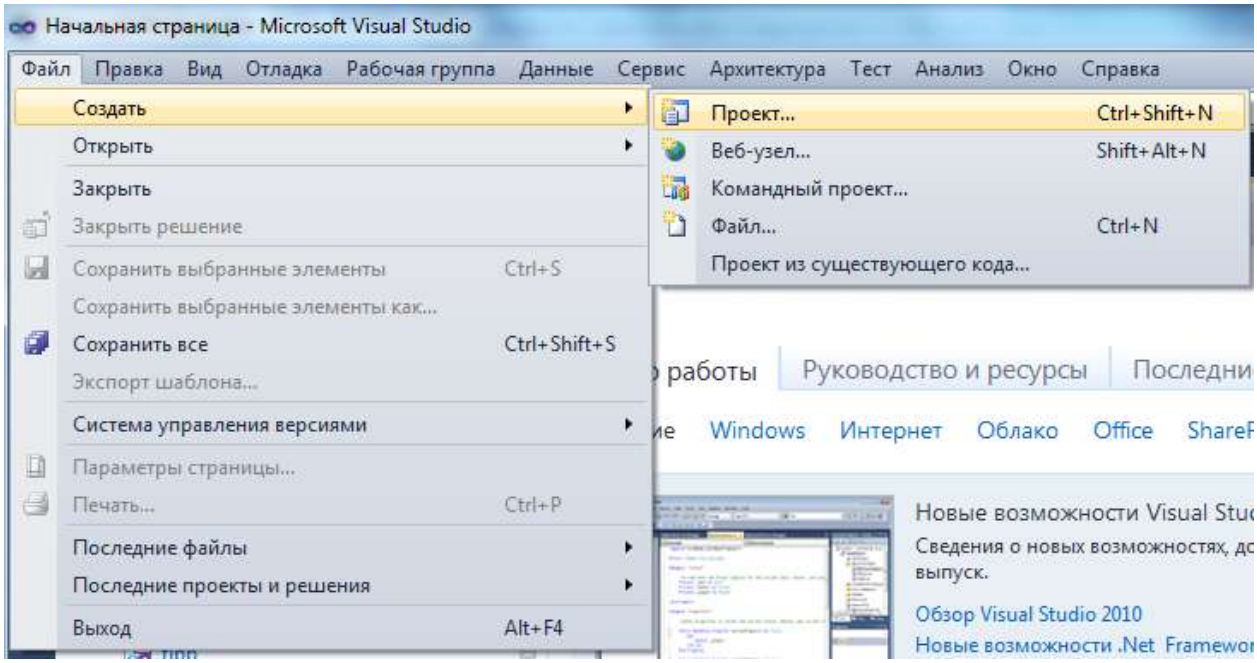


Приложение. Создание консольных приложений в MS Visual Studio 2010

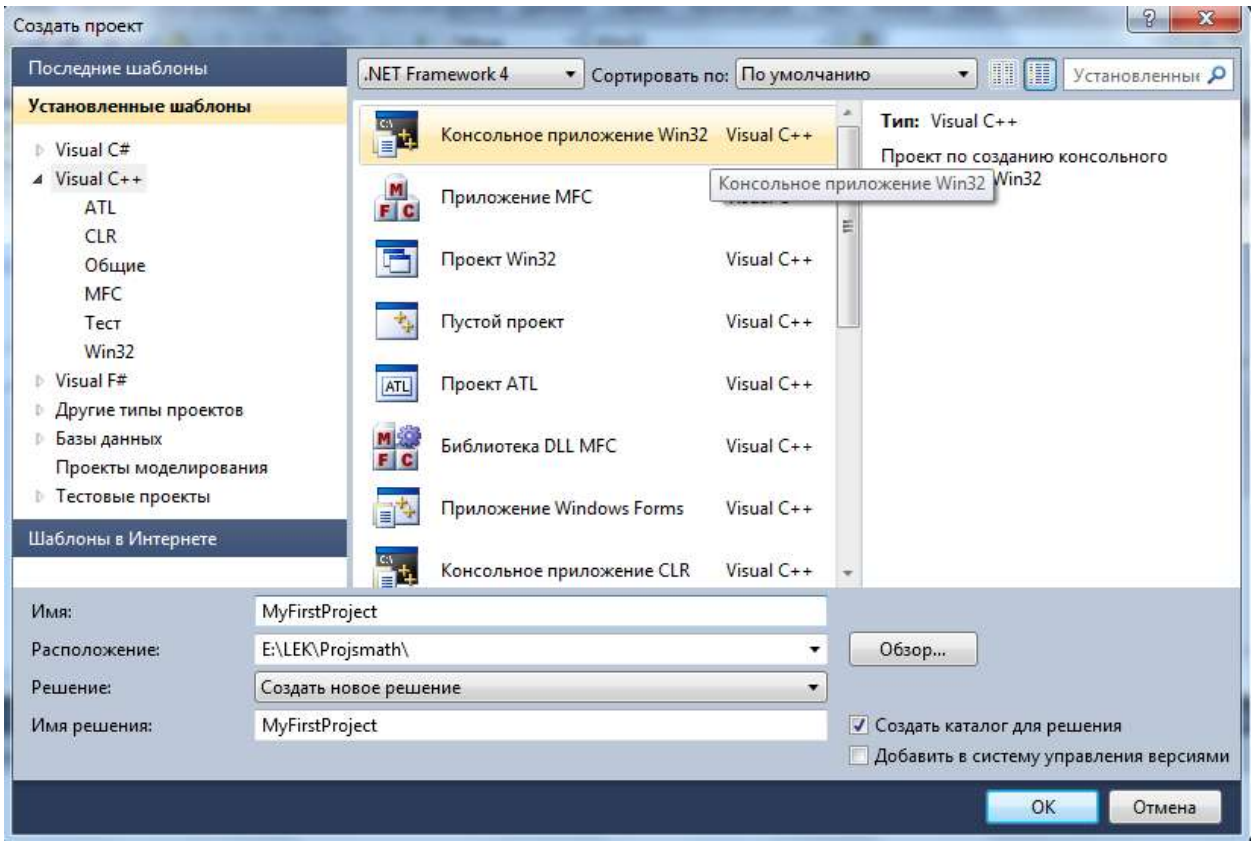
Консольные приложения – это программы, которые выполняются в окне "**Командная строка**". Эти приложения работают только в текстовом режиме. Консольные приложения компилируются в `.exe`–файл, который можно запускать на выполнение из командной строки как автономное приложение.

Для создания консольного приложения в среде программирования Microsoft Visual Studio 2010 можно использовать мастер приложений и шаблон консольного приложения.

Мастер приложений вызывается с помощью меню **Файл**, в котором выбирается пункт **Создать**, а затем пункт **Проект...**

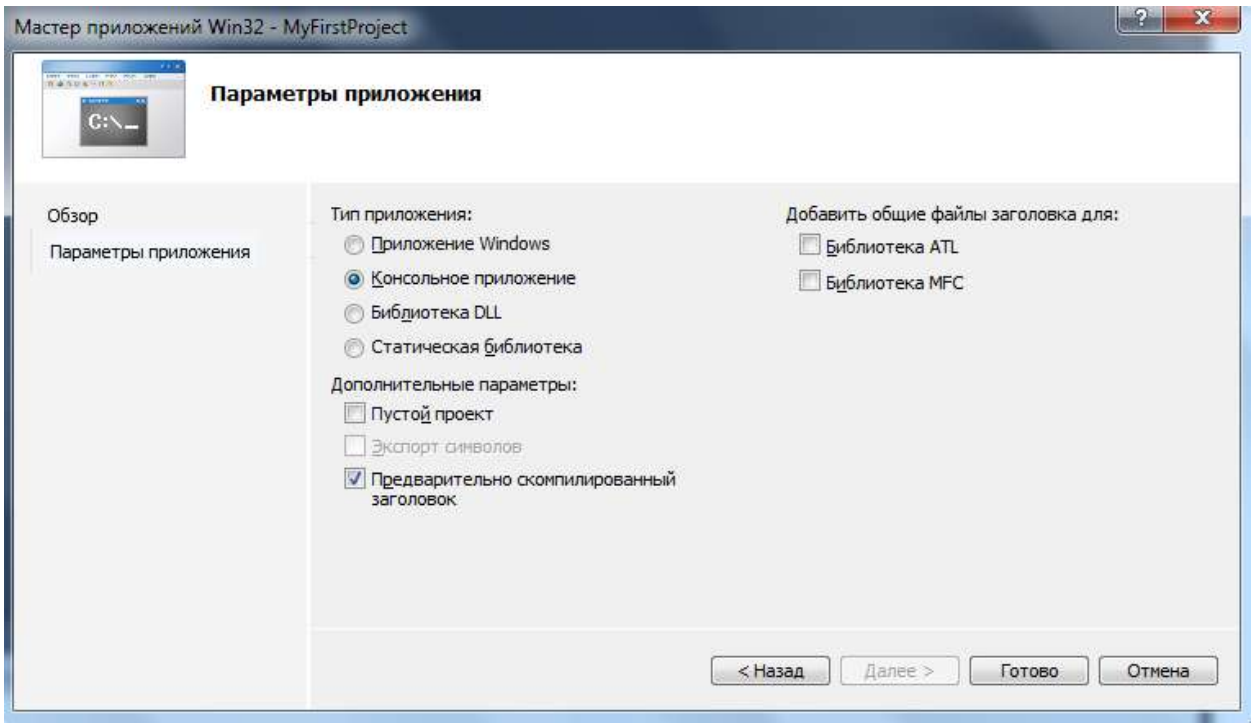


Мастер приложений предложит выбрать шаблон приложения в диалоговом окне **Создать проект**.

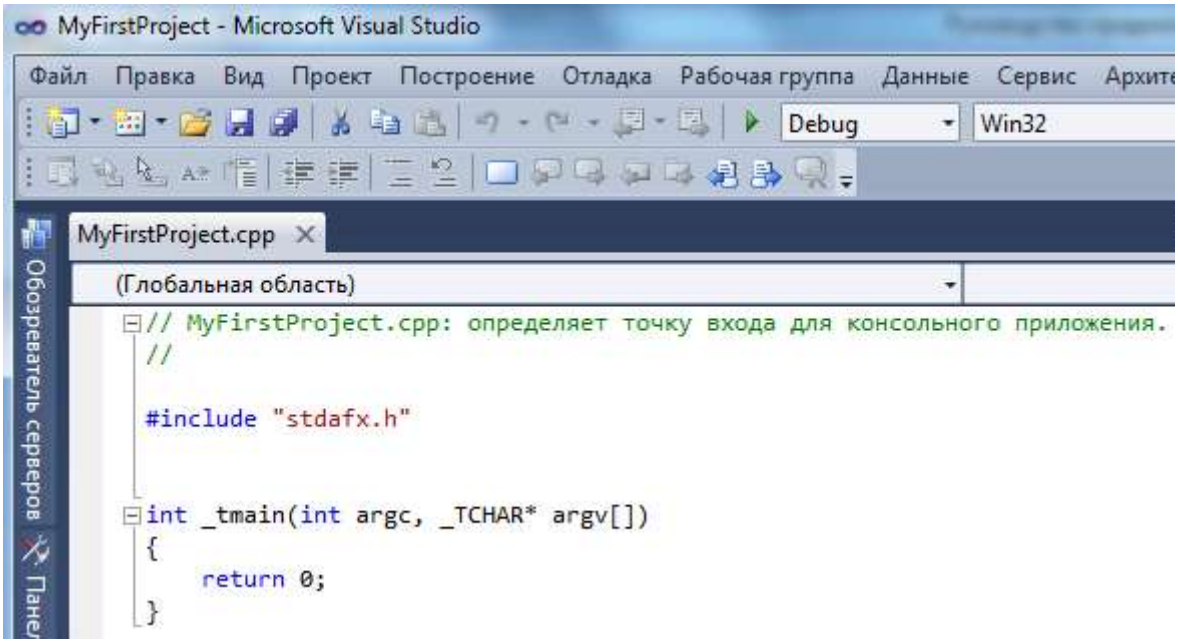


Рассмотрим это окно подробнее. Помимо шаблона **Консольное приложение Win32**, мастер предлагает большой список других шаблонов. Также в этом окне имеются поля для ввода имени проекта и указания папки, в которой будут записываться файлы проекта. С помощью кнопки **Обзор**... можно вызвать диалог выбора папки. Обращаем внимание на переключатель **Создать каталог для решения** (он размещен под кнопкой **Обзор**...). Если этот переключатель помечен, будет создана папка с именем проекта. В Visual Studio можно также создавать ***решения***, содержащие несколько проектов (подробнее см. раздел "Решения как контейнеры" на сайте MSDN – http://msdn.microsoft.com/ru-ru/library/df8st53z.aspx).

После нажатия на **OK** мастер приложений предложит подтвердить установленные параметры – нажатием на кнопку **Готово** – или продолжить настройку параметров (кнопка **Далее>>)**



Нажатие на кнопку **Готово** завершает работу мастера приложений и будет открыто окно с кодом



Кроме того, в папке проекта (в примере, папка имеет имя `MyFirstProject`) будут созданы файлы проекта – назначение этих файлов описано в файле `ReadMe.txt` (`MyFirstProject\MyFirstProject\ReadMe.txt`).

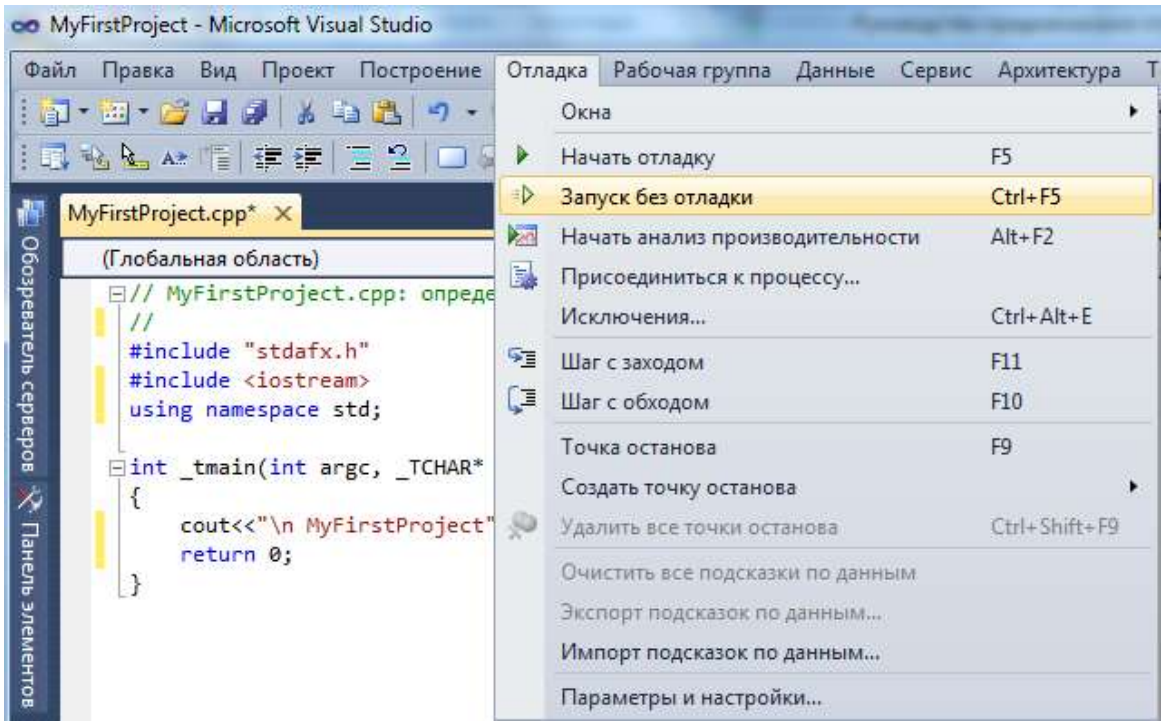
В большинстве консольных приложений потребуется ввод данных с клавиатуры и вывод на экран. Добавление строк

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

позволит использовать в программе объекты cin для ввода данных и cout – для вывода информации на экран.

```
// MyFirstProject.cpp: определяет точку входа для
консольного приложения.
//
#include "stdafx.h"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"\n MyFirstProject";
    return 0;
}
```

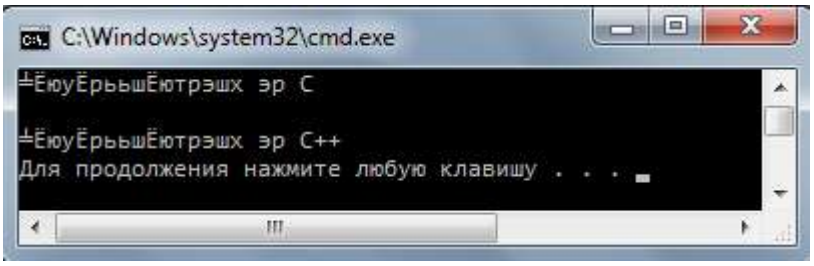
Компиляция и запуск программы выполняется с помощью меню **Отладка – Запуск без отладки** или командой **Ctrl+F5**.



Приложение. Русификация консольного ввода-вывода

При выполнении консольных приложений тексты, содержащие символы кириллицы, отображаются неправильно – в виде знаков псевдографики

```
printf("\nПрограммирование на C\n");
cout<<"\nПрограммирование на C++\n";
```



Это связано с различием кодировок, используемых в среде подготовки программ и консольном окне. В редакторе среды программирования (например, MS Visual Studio) используется кодировка cp1251 (“Кириллическая Windows”), а в окне консоли – кодировка cp866 (иначе DOS-кодировка).

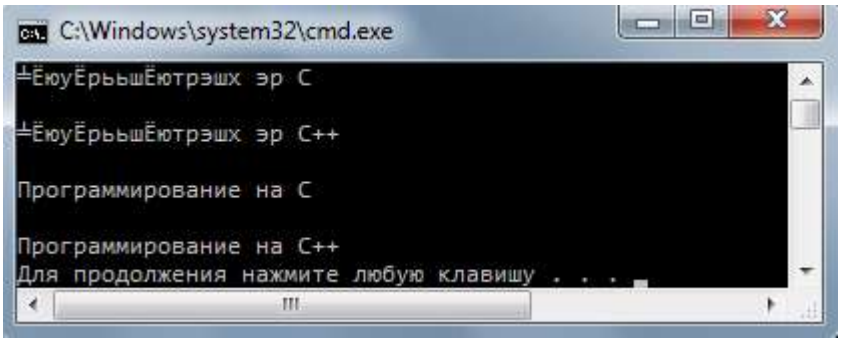
Есть несколько способов решить вопрос.

Первый, самый простой – перейти на латиницу и записывать все поясняющие тексты при выводе латинскими буквами.

Следующий вариант решения – перекодировка сообщений перед их выводом.

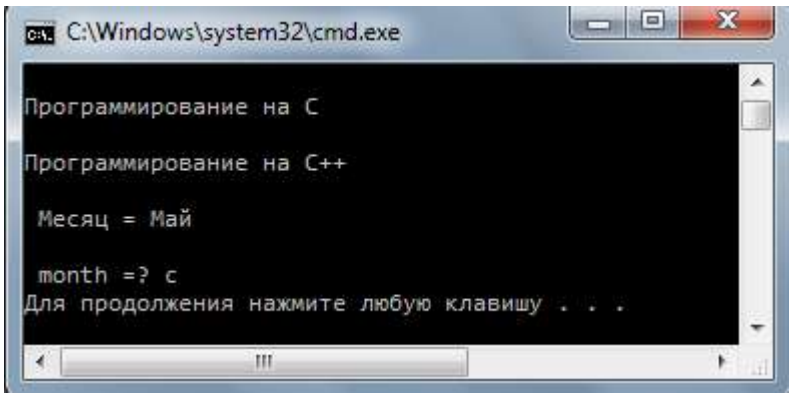
```
// Изменение кодировки cp1251 на cp866
#include <iostream>
using namespace std;
char * AnsiToOem(char *stroka);

int _tmain(int argc, _TCHAR* argv[])
{
    printf("\nПрограммирование на C\n");
    cout<<"\nПрограммирование на C++\n";
    char s1[]="\nПрограммирование на C\n";
    char s2[]="\nПрограммирование на C++\n";
    printf(AnsiToOem(s1));
    cout<<AnsiToOem(s2);
    return 0;
}
char * AnsiToOem(char *stroka)
{
    int cnt,i=0;
    char ch;
```

```
    while((ch=stroka[i])!='\0')
    {
      cnt=ch;
      if ((ch>='а') && (ch<='п')) cnt-=64;
        else if ((ch>='р') && (ch<='я')) cnt-=16;
          else if (ch=='ё') cnt=241;
           else if (ch=='Ё') cnt=240;
             else if ((ch>='А') && (ch<='Я')) cnt-=64;
      stroka[i]=cnt; i++;
    }
    return stroka;
}
```



Можно также установить кодовую страницу с помощью функции `setlocale()`, объявленную в файле `locale.h`. Этот способ, в большинстве случаев, наиболее оптимальный.

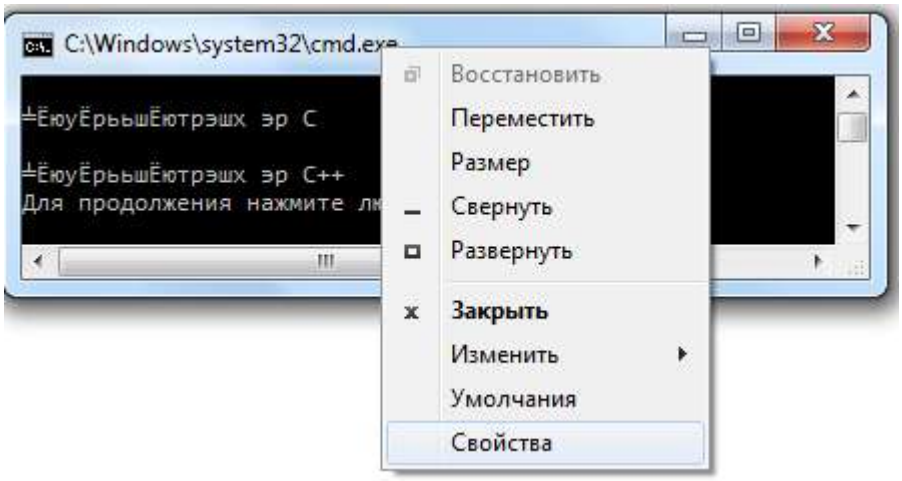
```
#include <iostream>
#include <clocale>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_CTYPE, "rus");
    printf("\nПрограммирование на С\n");
    cout<<"\nПрограммирование на С++\n";
    char month[10];
    cout<<"\n Месяц = ";cin>>month;
    cout<<"\n month ="<<month<<"\n";
    return 0;
}
```



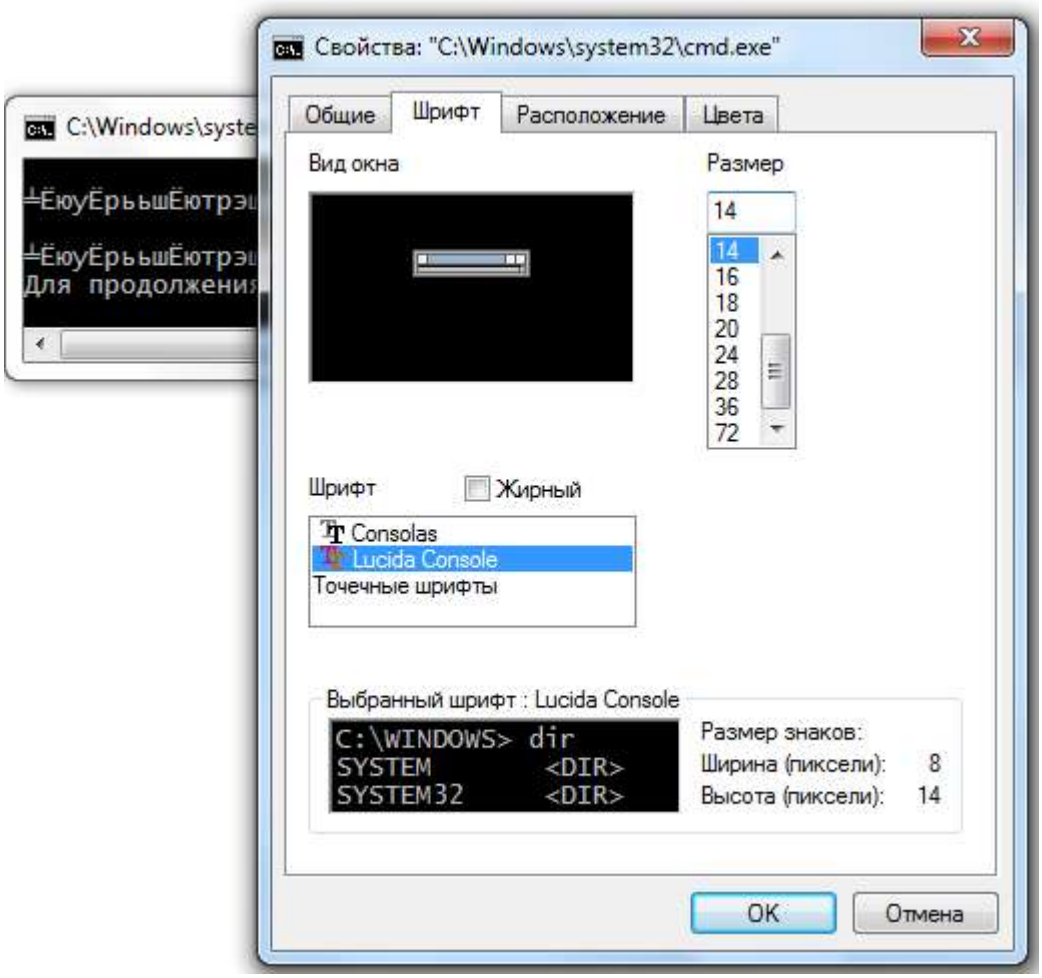
Однако, вывод символов, записанных в строку с клавиатуры (в примере ввели “Май” как значение строки `month`), по-прежнему, не дружит с кириллицей.

Еще один способ русификации консольного вывода заключается в настройке сеанса командной строки (см., напр., CppStudio. Программирование на C++. – http://cppstudio.com/).

Прежде всего, выберем для окна консоли шрифт, поддерживающий кириллицу. Для этого откроем системное меню консольного окна (щелкнув правой кнопкой мыши по заголовку окна) и выберем пункт “Свойства”.

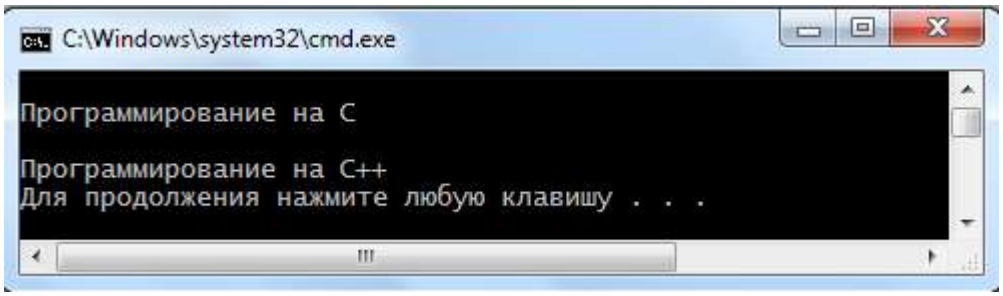


Перейдем на вкладку “Шрифт” и поменяем шрифт “Consolas”, установленный по умолчанию на “Lucida Console”.



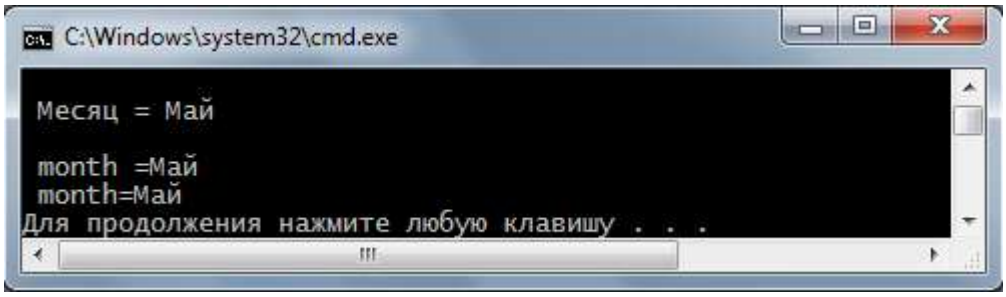
Эти настройки достаточно выполнить один раз, а в каждой программе нужно устанавливать кодовые страницы с помощью функций `SetConsoleCP()` и `SetConsoleOutputCP()`, объявленных в файле `windows.h`.

```
#include <iostream>
#include <windows.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
 SetConsoleCP(1251);// установка cp1251 для ввода
 SetConsoleOutputCP(1251);//установка cp1251 для вывода
 printf("\nПрограммирование на C\n");
 cout<<"\nПрограммирование на C++\n";
 return 0;
}
```



Символы кириллицы, введенные с клавиатуры, также правильно отобразятся функцией `printf()` и оператором `cout`.

```
#include <iostream>
#include <windows.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
 SetConsoleCP(1251);// cp1251 - для потока ввода
 SetConsoleOutputCP(1251);//cp1251 - для потока вывода
 char month[10];
 cout<<"\n Месяц = ";cin>>month;
 cout<<"\n month ="<<month;
 printf("\n month=%s\n",month);
return 0;
}
```



Литература

1. Керниган Б.В., Ричи Д.М. *Язык программирования С*. – М.: Издательский дом «Вильямс», 2009. – 304 с. http://www.intuit.ru/department/pl/cpl/
2. Керниган Б.В., Пайк Р. *Практика программирования*. – М.: Издательский дом «Вильямс», 2004. – 288 с.
3. Prata S. *C Primer Plus***. –** SAMS, 2004. – 984 p.
4. Прата С. *Язык программирования С++ (С++11)*. Лекции и упражнения, 6-е издание . — М.: «Вильямс», 2012. — 1248 с.
5. Шилдт Г. *Полный справочник по С*. – http://cpp.com.ru/shildt_spr_po_c/index.html
6. Дейтел Х. М., Дейтел П. Дж.. *Как программировать на С***.** – М.: ЗАО «Издательство БИНОМ», 2000. – 1008 с.
7. Deitel P., Harvey Deitel H.. *C: How to Program.* – Prentice Hall, 2009. – 1008 p.
8. *Голуб А.* И. Веревка достаточной длины, чтобы... выстрелить себе в ногу. Правила программирования на Си и Си++. – http://e-maxx.ru/bookz/files/golub_cord.pdf.
9. Штерн В. *Основы С++. Методы программной инженерии*. – М.: Издательство «Лори», 2003. – 860 с.
10. Хэзфилд Р., Кирби Л., Корбит Д. и др. Искусство программирования на С: Фундаментальные алгоритмы, структуры данных и примеры приложений. – К.: Изд-во “ДиаСофт”, 2001. – 736 с.
11. Саттер Г., Александреску А. *Стандарты программирования на С++. 101 правило и рекомендация.* – М.: Издательский дом "Вильямс", 2005.— 224 с.
12. Фридман А., Кландер Л., Михаэлис М., Шильдт Х. *С/С++. Архив программ*. – М.: ЗАО «Издательство БИНОМ», 2001. – 640 с.
13. Динман М.И. *С++. Освой на примерах*. – СПб.: БХВ-Петербург, 2006. – 384 с.
14. Кетков Ю.Л. *Введение в языки программирования С и С++*. – Интернет Университет Информационных технологий. http://www.intuit.ru/department/pl/ccpp/
15. Ворожцов А.В., Винокуров Н.А. *Лекции “Алгоритмы: построение, анализ и реализация на языке программирования Си”*. - М.: МФТИ, 2007. - 452 с.
16. Стауструп Б. *Язык программирования С++*. – М.: Издательство «Бином», 2011. – 1136 с.
17. Samuel P. Harbison III, Guy L. Steele Jr. C A Reference Manual. Fifth Edition. – Prentice-Hall, Inc., 2002. – 533 p.
18. Savitch W. *Problem Solving C++*. – Addison-Wesley, 2008. – 1048 p.
19. Scheinerman E. *C++ for Mathematicians. An Introduction for Students and Professionals*. - Chapman & Hall/CRC, 2006. – 521p.

20. Хабибуллин И.Ш. *Программирование на языке высокого уровня. C/C++*. – СПб.: БХВ-Петербург, 2006. – 512 с.
21. Stallman R.M., Weinberg Z.. *The C Preprocessor*. - Free Software Foundation, Inc., 2011. - 83 p. - http://gcc.gnu.org/onlinedocs/cpp.pdf
22. Кнут Д. *Искусство программирования, том 2. Получисленные алгоритмы* — 3-е изд. — М.: «Вильямс», 2007. – 832 с.
23. Стивенс Э. *Самоучитель по C++ от Wiley*. – М.: БИНОМ. Лаборатория знаний, 2005. – 872 с.
24. *C — The ISO Standard — Rationale, Revision 5.10*. – http://www.open-std.org/JTC1/SC22/WG14/www/docs/C99RationaleV5.10.pdf
25. *Programming languages — C.* ISO/IEC 9899:201x. Committee Draft — April 12, 2011. – http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
26. *Working Draft, Standard for Programming Language C++*. ISO/ISC DTR 19769 (February 28, 2011). - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/ 2011/ n3242.pdf
27. Липпман С., Ложойе Ж., Му Б. *Язык программирования C++. Вводный курс*. – М.: ООО “И.Д. Вильямс”, 2007. – 896 с.

Электронные ресурсы

1. Программирование на C и C++. – http://cpp.com.ru/
2. CppStudio. Программирование на C++. – http://cppstudio.com/
3. C++ reference. C reference. - http://en.cppreference.com/w/Main_Page
4. Справка по C++. Справка по C. - http://ru.cppreference.com/w/
5. cplus-plus.ru.Изучение языков программирования. – http://www.cplus-plus.ru/
6. CYBERN.RU. Для программистов и продвинутых пользователей ПК. – http://cybern.ru/
7. C Programming and C++ Programming. - http://www.cprogramming.com/
8. FunctionX Tutorials. - http://functionx.com/
9. Bjarne Stroustrup's homepage. – http://www2.research.att.com/~bs/homepage.html
10. Visual C++. – http://msdn.microsoft.com/ru-ru/library/60k1461a
11. Programmers Area. CyberGuru.ru. – http://www.cyberguru.ru/programming/cpp

Предметный указатель

Евгений Константинович Липачёв

Технология программирования. Базовые конструкции C/C++

Учебно-справочное пособие

Казанский университет