ФГБОУВПО «Воронежский государственный технический университет»
кафедра компьютерных интеллектуальных технологий проектирования

265-2011

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к практическим работам по дисциплине для студентов направления 230100.64 «Информатика и вычислительная техника» профиля «Системы автоматизированного проектирования в машиностроении» очной формы обучения
Часть 2

Воронеж 2011

Составители: канд. техн. наук А.Н. Юров,
канд. техн. наук М.В. Паринов,
д-р. техн. наук М.И. Чижов,
ст. преп. В.А. Рыжков

УДК 004.9

Методические указания к практическим работам по дисциплине для студентов направления 230100.64 «Информатика и вычислительная техника» профиля «Системы автоматизированного проектирования в машиностроении» очной формы обучения. Ч.2 / ФГБОУ ВПО «Воронежский государственный технический университет»; сост. А.Н. Юров, М.В. Паринов, М.И. Чижов, В.А. Рыжков. Воронеж, 2011. 42 с.

Методические указания содержат перечень базовых разделов с примерами решения задач на языке С++ по дисциплине «Программирование на языках высокого уровня».

Предназначены для студентов 1 курса.

Методические указания подготовлены в электронном виде в текстовом редакторе Microsoft Word 2007.

Ил. 4. Библиогр.: 10 назв.

Рецензент д-р техн. наук, проф. В.Н. Старов

Ответственный за выпуск зав. кафедрой д-р техн. наук, проф. М.И. Чижов

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

6. ОРГАНИЗАЦИЯ ЦИКЛИЧЕСКИХ КОНСТРУКЦИЙ

6.1. Общие сведения о повторяющихся процессах и способах их реализации

При разработке программного обеспечения в ряде случаев необходимо использовать такие действия, которые выполняются с заданной периодичностью, например ввод однотипных данных, вычисление подобных величин по известным формулам и т.д. В жизни многократно-повторяющиеся события также не являются редкостью - смена дня и ночи, ход секундной (минутной, часовой) стрелки, периодичность сезонов и многие другие явления и процессы.

Вычислительный процесс с многократным повторением однотипных вычислений для различных значений обрабатываемых величин (переменных) называется циклическим, повторяющиеся участки вычисления - циклами, изменяющиеся в цикле величины - переменные цикла.

Для организации циклов необходимо в алгоритмах предусмотреть:

-организацию цикла: задание начальных значений переменным цикла перед его выполнением;

-тело цикла, действия, повторяемые в цикле для различных значений переменных цикла;

- изменение значений переменных цикла перед каждым его повторением;

-управление циклом посредством проверки условия продолжения или окончания цикла.

6.2. Конструкция повторения for

Конструкция цикла for в C++ отличается большей гибкостью, поскольку с его помощью можно организовать как фиксированные, так и условные итерации. Однако в большинстве случае конструкция применяется как

фиксированная. Форма записи конструкции цикла for выглядит следующим образом:

```
for (< инициализация переменных управления циклом>;
<проверка продолжения цикла>;
< модификация переменных управления циклом>)
<выражение>;
```

В следующем примере показана реализация цикла for при выводе заданной числовой последовательности целых чисел.

```
#include <iostream>
#include <cstdlib>
int main()
{
   //вывод на экран значений от 1 до 10
    for (int i=1; i<=10; i++)
    std::cout<<i<<std::endl;
    system("PAUSE");
    return 0;
}
```

Оператор for имеет три компоненты, каждый из которых необязателен. Первый компонент инициализирует переменные управления циклом. Второй - это условие, которое проверяет, будет ли цикл выполнять следующую итерацию, последний компонент - приложение, которое изменяет переменные управления циклом; как правило, в указанном блоке операции инкремента или декремента (увеличение или уменьшение числа на единицу). В качестве примера приведена программа, которая рассчитывает сумму и среднее значение ряда целых чисел, используя цикл for.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main ()
{
int count=0;
double sum=0.0;
int first, last, temp;
cout << "Enter a first Int:";
```

```
cin >> first;
cout << "Enter a last Int:";
cin >> last;
if (first > last)
  { temp = first;
     first = last;
     last = temp;
  }
for (int i=first; i<=last; i++)
  {
  count ++;
  sum +=static_cast<double>(i);
  }
cout << "Add Int value:"
<< first << " to " << last << "="
<< sum << endl;
cout << "Average value" << sum/count<<endl;
system("PAUSE");
return 0;
}
```

Приведение типа double к типу int выполнено с помощью инструкции static_cast.

Допускается исключение одного, двух или трех выражений в описании цикла for. Если оставить все три компонента цикла пустыми, результатом будет организация работы бесконечного цикла. C++ позволяют выходить из них следующими способами:

-оператор break вызывает переход к выполнению кода, следующего за текущим циклом, во многом подобен тому, как он мог бы быть использован, чтобы продолжить выполнение вне оператора switch. Использование оператора break позволяет выйти из цикла и продолжить работу, оставшейся части программы;

-оператор return осуществляет возврат из текущей функции (включая main);

-в ряде случаев выйти из программы можно с помощью функции exit (требуется для получения заголовочный файл cstdlib). Эту функцию используют только в крайней

необходимости, когда требуется завершить работу программы. Указанная функция exit прекратит выполнение итерации и приведет к выходу из программы. Далее приводится пример использования конструкции for без параметров.

```
#include <iostream>
#include <cstdlib>
#include <stdio.h>
int main()
{
    char ch = '\0';
    for( ; ; )
    {
      ch = getchar(); /* считывание символа */
      if(ch=='A') break; /* выход из цикла */
    }
    std::cout<<"You are enter symbol 'A'"<<std::endl;
    system("PAUSE");
    return 0;
}
```

В языке C++ допускаются некоторые варианты оператора for, позволяющие во многих случаях увеличить мощность и гибкость программы. Один из распространенных способов усиления мощности цикла for — применение оператора "запятая" для создания двух параметров цикла. Оператор "запятая" связывает несколько выражений, заставляя их выполняться вместе. В примере обе переменные (x и y) являются параметрами цикла for и обе инициализируются в этом цикле:

```
for(x=0, y=0; x+y<10; ++x)
{
  y = getchar();
  y = y - '0'; /* Вычитание из y ASCII-кода нуля */
    .
    .
    .
}
```

Здесь запятая разделяет два оператора инициализации. При каждой итерации значение переменной x увеличивается, а значение y вводится с клавиатуры. Для выполнения итерации x и y должны иметь определенное значение. Несмотря на то, что значение y вводится с клавиатуры, оно должно быть инициализировано таким образом, чтобы выполнилось условие цикла при первой итерации. Если y не инициализировать, то оно может случайно оказаться таким, что условие цикла примет значение ЛОЖЬ, тело цикла не будет выполнено ни разу.

6.3. Цикл while

Цикл while в C++ - условный цикл, в котором операции выполняются до тех пор, пока условие имеет значение true. Т.о. цикл while может не выполнить ни одной операции, если проверяемое условие изначально имеет значение false. Общая форма цикла while имеет следующий вид:

```
while (условие) оператор;
```

Здесь оператор (тело цикла) может быть пустым оператором, единственным оператором или блоком. Условие (управляющее выражение) может быть любым допустимым в языке выражением. Условие считается истинным, если значение выражения не равно нулю, а оператор выполняется, если условие принимает значение ИСТИНА. Если условие принимает значение ЛОЖЬ, программа выходит из цикла и выполняется следующий за циклом оператор. Для рассмотрения работы цикла while приводится пример по вычислению x в степени n.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
```

```
{
    int i=1,pow=1,x,n;
    cout<<"Enter a base number x and element n"<<endl;
    cin>>x>>n;
    while(i<=n)
    {
     pow*=x;
     ++i;
    }
    cout<<pow<<endl;
    system("PAUSE");
    return 0;
}
```

6.4. Цикл do-while

Основным отличием от рассмотренных выше циклических конструкций связки do-while является факт проверки условия в конце приведенной записи цикла. Поэтому цикл do-while всегда выполняется как минимум один раз. Общая форма цикла do-while следующая:

```
do
{
  оператор;
} while (условие);
```

Если оператор не является блоком, фигурные скобки не обязательны, но их почти всегда ставят, чтобы оператор достаточно наглядно отделялся от условия. Итерации оператора do-while выполняются, пока условие не примет значение ложь.

Цикл do-while часто используется в функциях выбора пунктов меню. Если пользователь вводит допустимое значение, оно возвращается в качестве значения функции. В противном случае цикл требует повторить ввод. Следующий пример демонстрирует возможности по выбору пункта меню некоторой программной подсистемы:

```
#include <iostream>
#include <cstdlib>
```

```
#include <stdio.h>
using namespace std;
int main()
{
      char ch;
      cout<<"1. Action 1"<<endl;
      cout<<"2. Action 2"<<endl;
      cout<<"3. Action 3"<<endl;
      cout<<"  Enter you actions?"<<endl;
      do {
      ch = getchar(); /* чтение выбора с клавиатуры */
        switch(ch) {
          case '1':
            cout<<"Run 1 action"<<endl;
            break;
          case '2':
             cout<<"Run 2 action"<<endl;
            break;
          case '3':
            cout<<"Run 3 action"<<endl;
            break;
        }
      } while(ch!='1' && ch!='2' && ch!='3');
    system("PAUSE");
    return 0;
}
```

В этом примере применение цикла do-while является технически правильным, потому что итерация, как уже упоминалось, всегда должна выполниться как минимум один раз. Цикл повторяется, пока его условие не станет ложным, т.е. пока пользователь не введет один из допустимых ответов.

Задачи на самостоятельное решение

1. Дана последовательность из n целых чисел. Найти количество элементов этой последовательности, кратных числу К.

2.Дана последовательность целых чисел. Найти сумму нечетных элементов этой последовательности.

3. С клавиатуры вводится ряд чисел. Составить программу, которая определяет количество отрицательных, количество положительных и количество нулей в набранной последовательности чисел.

7. МАССИВЫ ДАННЫХ

7.1. Понятие о массивах, порядок их объявления и доступ к элементам массива

Массив-это набор переменных одного типа, имеющих одно и то же имя. Доступ к конкретному элементу массива осуществляется с помощью индекса. В языке С и С++ все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний - по самому большому.

Для объявления и использования массива в программе необходимо определить имя и тип будущего массива, а также указать его размер. Общий вид записи при объявлении массива в программе имеет следующий вид:

```
тип имя_массива [размер];
```

Если требуется определить массив данных с именем “array”, состоящих из десяти целочисленных значений, форма объявления массива будет выглядеть следующим образом:

```
int array[10];
```

В рассмотренном примере массив однозначно определен. В 32-разрядных системах под массив будет выделена память в 40 байт, так как типу int (тип int является аппаратно-зависимым типом) соответствует 4 байта информации.

Объявить и присвоить значение элементам массива можно следующим образом:

```
int array[10]={14,5,3,7,5,-3,16,7,8,7};
```

Согласно стандарту С++ размер массива должен быть указан явно с помощью выражения-константы. Таким образом,

в программе размер массива определяется во время компиляции и впоследствии остается неизменным. Доступ к элементу массива осуществляется посредством индекса, заданного константой или ссылочной переменной. Пример вывода значения массива по указанному индексу следующий:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int array[10]={14,5,3,7,5,-3,16,7,8,7};
    cout<<array[9]<<endl;
    for (int i=3;i<9;i+=2) cout<<array[i]<<endl;
    system("pause");
    return 0;
}
```

Задать размерность массива можно другими способами, при этом индекс количества элементов указывать необязательно. В предлагаемом примере размер памяти под массив будет выполнен автоматически.

```
int array[]={1,2,3,7,-6};
```

Символьные массивы могут инициализироваться как обычный массив или могут вводиться как строка символов char str[15]=”QTCreator C”.

```
#include <iostream>
#include <cstdlib>
#include <conio.h>
int main()
{
     char str[15]="QTCreator C";
     std::cout<<str<<std::endl;
     system("PAUSE");
     return 0;
}
```

7.2. Двухмерные и многомерные массивы

Объявление двухмерных массивов не отличается от формы записи одномерных, которые были рассмотрены в

предыдущем разделе. Создание двухмерного массива mass с размерами 10 на 10, который состоит из вещественных чисел, выглядит следующим образом:

```
float mass[10][10];
```

В отличных от C++ других языках определения количества вхождений элементов отделяются друг от друга запятой. В C++ каждое количественное значение заключено в свои квадратные скобки. Двухмерные массивы размещаются в матрице, состоящей из строк и столбцов. Первый индекс указывает номер строки, а второй — номер столбца. Доступ к заданному элементу массива посредством указания индексов, например 3 строка и 4 столбец, выглядит следующим образом (необходимо помнить, что отсчет индексов осуществляется с 0, а не с 1):

mass [2][3];

В следующем примере элементам двухмерного массива присваиваются числа от 1 до 12 и значения элементов выводятся на экран построчно:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  int t, i, num[3][4];
  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;
  /* вывод на экран */
  for(t=0; t<3; ++t)
  {
    for(i=0; i<4; ++i)
      cout<<num[t][i]<<"  ";
    cout<<endl;
  }
    system("pause");
    return 0;
}
```

При объявлении массивов с неизвестным количеством элементов можно не указывать размер только в самых левых квадратных скобках:

```
    int arr[][3]={1, 2, 3,
                  5, 6, 7,
                  8, 9, 0};
// Пример задачи: Дан двухмерный массив. Необходимо,
сделать ввод/вывод двумерного массива. Найти
максимальный элемент двумерного массива.
#include <iostream>
#include <cstdlib>
#include <conio.h>
const int n = 3; //объявляем константу для массива
int main ()
{
    int X[n][n]; //объявляем массив целого типа
    int i,j; //переменные для цикла
    int max = 0; //Переменная для вычисления
максимального числа
    for (i = 0; i < n ; i++)
    for (j = 0; j < n ; j++)
    { //цикл ввода массива
    std::cout<<"X["<<i<<","<<j<<"] = "; //На экран
выводится 'X[i][j] = '
    std::cin>>X[i][j]; //вводим с клавиатуры целые
числа
    }
    std::cout<<"\n"; //Переход на следующую строку
    for (i = 0; i < n ; i++)
    { //цикл вывода массива
    std::cout<<"\n"; //Переход на следующую строку
    for (j = 0; j < n ; j++)
    {
    std::cout<<"X["<<i<<"]"<<"["<<j<<"] =
"<<X[i][j]<<"\t"; //На экран выводится результат
нашего ввода
    }
    }
    max = X[0][0]; //Допустим маскимальное число - это
первое число массива
    for (i = 1; i < n ; i++)
    { //цикл находжение максимального элемента массива
```

```
    std::cout<<"\n"; //Переход на следующую строку
    for (j = 0; j < n ; j++)
    {
    if (X[i][j]> max) max = X[i][j]; //если есть число
большее, чем max
    //тогда max принимает значение этого числа
    }
    }
    std::cout<<"\n"; //Переход на следующую строку
    std::cout<<"Max = "<<max<<std::endl; //На экран
монитора выводится максимальное число
    _getch();
//Экран не закрывается, пока не нажата любая клавиша
     return 0;
    }
```

Многомерный массив может быть определен следующим образом, однако такого рода способы создания массивов применяют нечасто:

```
double my_data[2][3][5][7][11];
```

Задачи на самостоятельное решение

1.Отыскать в заданном массиве все простые числа.

2.Получить разность между максимальным и минимальным значением последовательности, которая хранится в массиве.

3. Подсчитать количество отрицательных и количество положительных чисел, кратных 3 и 5.

8. ПРЕДСТАВЛЕНИЕ СИМВОЛЬНЫХ И СТРОКОВЫХ ДАННЫХ

8.1. Символьные формы записи данных

Символы в языке С++ могут применятся по отдельности или входить в состав строк. Строкой называется некоторая последовательность символов. Строки в языке С++ могут быть представлены массивами или указателями. Указатели и операции с ними будут рассмотрены в последующих разделах.

Если строка в массиве представлена символами в один байт, массив имеет тип char. При этом существуют и другие кодировки, в которых символ представляется, например, двумя байтами. Для работы с такими строками требуются специальные функции. Понятие кодировки следующее - это способ представления символов для заданной среды разработки или операционной системы. Применяются однобайтные и многобайтные кодировки. Для однобайтных кодировок каждый символ однозначно определен одним байтом. Многобайтное представление символов заключается в использовании при описании данных двух или более байтов информации, которые соответствуют 1 символу. В свою очередь многобайтные кодировки можно разделить на кодировки с фиксированным количеством байтов - каждому символу соответствует одинаковое количество байтов, и «плавающие», в которых один символ может представляться разным количеством байтов в зависимости от его содержимого. К первым относятся кодировки типа Unicode, в которой каждый символ представлен двумя байтами, ко вторым - UTF-8 и др.

Необходимость в многобайтных кодировках возникла из-за того, что одним стандартным байтом можно представить не так много символов, например восьмибитный байт способен принимать значения от 0 до 255, а значит в такой кодировке не может существовать более 256 различных символов.

Между однобайтными и фиксированными многобайтными строками принципиальной разницы нет. В C/C++ существует специальный тип для многобайтных символов - wchar_t и специальные функции для работы со строками, состоящими из таких символов. Размер wchar_t не фиксирован в стандарте и определяется реализацией компилятора. На многих платформах и компиляторах это два байта, соответствующих кодировке Unicode. Кроме того, существует специальная форма для записи строковых литералов, в которых символы представлены несколькими

байтами: перед кавычками ставится буква L. Т, вызов функции MessageBox при подключенном заголовке <windows.h> в Unicode-программе будет выглядеть так:

```
MessageBox(NULL,L"Message", L"Новый заголовок окна",MB_OK);
```

Однако если речь идет о строках типа char, то строка – это массив однобайтных элементов, которая завершается ограничивающим символом с кодом 0. Без объявленного массива тип char используется как символьный. В отличие от строк, символ — это встроенный интегральный тип в C/C++, для него допустимы все операции, допустимые для интегральных типов. Существуют символьные литералы, они записываются в одинарных кавычках (прямых апострофах). Пример символьного литерала:

```
char code;
code='A'; //Символ
```

В вышеприведенном примере значением sym является 65 в кодовой таблице ASCII. В этом случае строка sym=’A’ абсолютно эквивалентна строке sym=65. Однако в случае переноса разрабатываемого приложения на другие архитектуры лучше всегда использовать запись в апострофах, где у символа А другой код. Для записи символьных литералов типа wchar_t используется запись, аналогичная записи для строковых литералов этого типа:

```
wchar_t code;
code=L'ab'; //Символьный многобайтовый литерал
```

Количество символов между апострофами зависит от размера типа wchar_t. Кроме того, в C++ имеется возможность для записи символьных литералов – слеш, за которым идет код символа. Такая форма записи необходима, если требуется использовать элемент, не отображающийся в печатный символ, например признак окончания строки (’\0’).

8.2. Операции со строками

Существуют следующие варианты объявления строковых массивов:

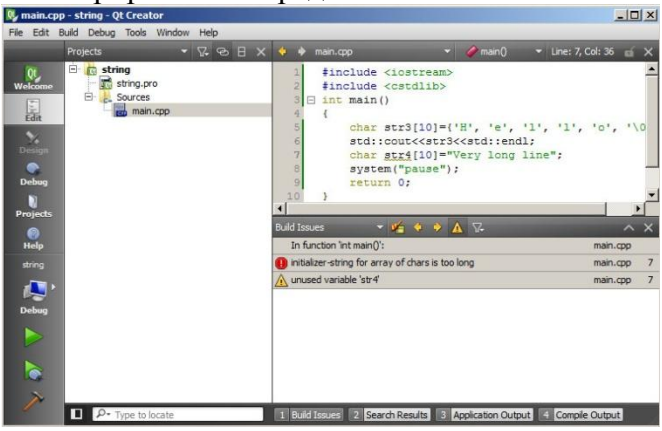
```
     char str1[10];
//Строка - массив из 10 символов.
     char str2[10]="Hello";
// В первые 5 символов записывается “Hello”,в 6- ‘\0’,
// значение трех последних не определено.
     char str3[10]={'H', 'e', 'l', 'l', 'o', '\0'};
```

//Запись соответствует предыдущей записи из приведенного примера.

Реализация в программе выглядит следующим образом:

```
#include <iostream>
#include <cstdlib>
int main()
{    char str3[10]={'H', 'e', 'l', 'l', 'o', '\0'};
     std::cout<<str3<<std::endl;
     system("pause");
     return 0;
}
```

В следующем фрагменте компилятор выдаст ошибку, так как количество символов, определенное для заданного массива, превышено. На рисунке 1 показана ошибка, которая выявлена интегрированной средой.



Ошибки при сборке проекта, если длина записи в массив превышена его размером

```
    char str4[10]="Very long line";
//Ошибка. Массив из 10 элементов нельзя
//инициировать более длинной последовательностью.
    char str5[]="Very long line";
//Компилятор автоматически определяет длину массива
//(в указанном случае 15) и инициализирует его
//последовательностью символов.
```

8.3. Присваивание строк

Основной способ присваивания строк – присваивание отдельных символов. Присваивание можно выполнить следующим образом:

```
str1[0]='H';
str1[1]='e';
str1[2]='l';
str1[3]='l';
str1[4]='o';
str1[5]='\0';
```

Однако форма заполнения, предложенная в тексте фрагмента программы, не является удобной. Для корректной работы со стоками существует целый ряд функций, которые определены в заголовке <cstring>. Для копирования одной строки в другую можно использовать функцию strcpy(), формат которой имеет вид:

```
char* strcpy(char* dest, const char* src)
```

Функция посимвольно копирует содержимое строки, на которую указывает src в строку, на которую указывает dest и возвращает dest. Пример использования приведен в следующем фрагменте:

```
char str1[10], str2[10];
strcpy(str1, "Hello");
strcpy(str2, str1);
```

8.4. Сравнение строк

Для сравнения строк используются функции strcmp и stricmp. Первая сравнивает строки с учетом регистра, вторая –

без. Однако данные функции предназначены для работы с латинскими символами. Если требуется сравнивать без учета регистра кириллические строки, придется использовать заголовок библиотеки <clocale>. Прототипы функций следующие:

```
int stricmp(const char *string1, const char *string2);
int strcmp(const char *string1, const char *string2);
```

Обе функции возвращают число, меньшее 0, если первая строка меньше второй, большее нуля, если первая строка больше второй и 0, если строки равны.

Для вычисления длины строки используется функция

```
size_t strlen(const char *string);
```

Функция возвращает длину строки, не включая нулевой символ в конце строки.

8.5. Преобразования строк

Зачастую требуется преобразовать число в строку и наоборот. Для этих целей доступно целое семейство функций atof, atoi, atol и itoa, ltoa. Все они имеют схожую структуру. Функции из первой группы преобразуют строку в число (float, int или long) в зависимости от окончания. Функции из второй группы выполняют обратное преобразование. Описание функций из первой группы имеет следующий вид:

```
double atof(const char* string);
int atoi(const char* string);
long atol(const char* string);
```

Вторая группа представлена следующими функциями:

```
char* itoa(int value, char* string, int radix);
char* ltoa(long value, char* string, int radix);
```

Функции из второй группы могут создавать строковое представление чисел в любой системе (в зависимости от

системы счисления) от 2 до 36. Основание передается в третьем параметре. Чтобы получить строковое представление числа в десятичной системе, необходимо установить значение, равное 10.

Пример преобразования типов показан на примере программы:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    char str1[5];
    itoa(12, str1, 10); //str1=”12”
    cout<<str1<<endl;
    itoa(12, str1, 16); //str1=”C”
    cout<<str1<<endl;
    itoa(12, str1, 2); //str1=”1100”
    cout<<str1<<endl;
    system("pause");
    return 0;
}
```

Для перевода чисел с плавающей точкой в строковый набор предназначен следующий метод:

```
char* _gcvt(double value, int digits, char* buffer);
```

Первым параметром является число для перевода, вторым - система счисления и последним - строка, в которую будет записано число после преобразования

Пример использования предложен в листинге:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{   char str[20];
    double val;
    std::cin>>val;
    _gcvt(val, 10, str);
    std::cout<<str<<std::endl;
    system("pause");
    return 0;
}
```

8.6. Конкатенация (объединение) строк

Для объединения содержимого строк существуют функции, которые имеют следующее описание.

```
char* strcat(char* dest, const char* source)
char* strncat(char* dest, const char* source, size_t size)
```

Эти функции добавляют к строке, на которую указывает dest, символы из строки source. Первая функция добавляет все символы до признака окончания строки, вторая – максимум size символов. Результирующая строка завершается ‘\0’.

Задачи на самостоятельное решение

1.Определить количественное вхождение латинского символа ‘A’ в предложении, состоящем из 100 знаков.

2.Определить символ, который чаще чем остальные встречается в указанной строке.

3.Из десяти введенных строк определить строки, содержащие наибольшее и наименьшее количество символов, а также произвести их объединение.

9. УКАЗАТЕЛИ И ОБЛАСТИ ИХ ПРИМЕНЕНИЯ

9.1. Общие сведения об указателях

Для работы с данными в программе используются переменные, которые вызываются по мере необходимости. В языке C++ есть возможность ссылаться на данные посредством ссылок и указателей, тем самым экономя ресурсы памяти ЭВМ.

Указатель - это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера. Например, если одна переменная

содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую.

Как и переменную перед использованием, указатель необходимо задать, объявив соответствующим типом. Объявление указателя состоит из имени базового типа, символа *и имени переменной. Общая форма объявления указателя следующая:

```
тип *имя;
```

Здесь тип - это базовый тип указателя, им может быть любой из имеющихся в системе типов. Указатель выбранного типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа int *, компилятор предполагает, что любой адрес, на который он ссылается, содержит переменную типа int, хоть это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

В языке C++ определены две операции для работы с указателями: * и &. Оператор & — это унарный оператор, возвращающий адрес своего операнда. Для взятия адреса значения переменной и присвоения его указателю используется следующее выражение:

```
int *m,count;
m = &count;
```

Вторая операция для работы с указателями (*) выполняет действие, обратное по отношению к &. Оператор * - это унарный оператор, возвращающий значение переменной, расположенной по указанному адресу. В следующем примере показано, как подготовить вывести значение указателя, если известен адрес переменной.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
```

```
 int p=15;
 int *n;
 n=&p;
    cout<<*n<<endl;
    system("pause");
    return 0;
      }
```

Указатель можно использовать в правой части оператора присваивания для присваивания его значения другому указателю. Если оба указателя имеют один и тот же тип, то выполняется простое присваивание, без преобразования типа.

```
#include <iostream>
#include <cstdlib>
#include <stdio.h>
using namespace std;
int main()
{
    int value = 50;
    int *p1, *p2;
    p1 = &value;
    p2 = p1;
    //Значения, которые содержат указатели
    cout<<*p1<<" "<<*p2<<endl;
    //Адрес в памяти, на который ссылаются указатели
    cout<<p1<<" "<<p2<<endl;
    system("pause");
    return 0;
}
```

9.2. Адресная арифметика

Для языка С допустимы только две арифметические операции над указателями: суммирование и вычитание. Предположим, текущее значение указателя p1 типа int * равно 2000. Известно, что переменная типа int занимает в памяти 4 байта для 32 разрядных систем. Тогда после операции увеличения p1++; указатель p1 принимает значение 2004, а не

2001. То есть, при увеличении на 1 указатель p1 будет ссылаться на следующее целое число. Это же справедливо и для операции уменьшения. Например, если p1 равно 2000, то после выполнения оператора p1--; значение p1 будет равно 1996.

Операции адресной арифметики подчиняются следующим правилам. После выполнения операции увеличения над указателем, данный указатель будет ссылаться на следующий объект своего базового типа. После выполнения операции уменьшения — на предыдущий объект. Следующий пример демонстрирует возможности изменения адреса с помощью указателей.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    int *p1,value=2000;
    //Выведен адрес 0x405416 (произвольный), так как
указатель не связан с заданным значением
    cout<<p1<<"\n";
    p1=&value;
    //Выведен адрес 0x28ff38, так как указателю
присвоен адрес переменной value
    cout<<p1<<"\n";
    p1++;
    //Выведен адрес 0x28ff3c, так как адрес был
увеличен на 1, значение увеличивается на тип int-4
байта
    cout<<p1<<"\n";
    cin.get();
    return 0;
}
```

9.3. Указатели и массивы

Понятия указателей и массивов тесно связаны. Рассмотрим следующий фрагмент программы:

```
char str[80], *p1;
p1 = str;
```

Здесь p1 указывает на первый элемент массива str. Обратиться к пятому элементу массива str можно с помощью любого из двух выражений:

```
str[4]
*(p1+4)
```

Массив начинается с нуля. Поэтому для пятого элемента массива str нужно использовать индекс 4. Можно также увеличить p1 на 4, тогда он будет указывать на пятый элемент (имя массива без индекса возвращает адрес первого элемента массива).

В языке C++ существуют два метода обращения к элементу массива: адресная арифметика и индексация массива. Стандартная запись массивов с индексами наглядна и удобна в использовании, однако с помощью адресной арифметики иногда удается сократить время доступа к элементам массива. Поэтому адресная арифметика часто используется в программах, где существенную роль играет быстродействие.

В следующей программе приведены два варианта вывода строки на экран. В первой версии используется индексация массива, а во второй — адресная арифметика:

```
 #include <cstdlib>
 #include <iostream>
  using namespace std;
int main()
{
   char *s;
   char buf[]="Base string";
   //Получение адреса первого элемента массива
   s=buf;
   /* Индексация указателя s как массива. */
   for(int t=0; s[t]; ++t) cout<<(s[t]);
   cout<<"\n";
   /* Использование адресной арифметики. */
   while(*s) cout<<(*s++);
   cin.get();
   return 0;
}
```

Вторая форма записи, предложенная в программе, является более наглядной и удобной. Для большинства компиляторов она также более быстродействующая. Поэтому в процессе разработки программ приемы адресной арифметики используются довольно часто.

9.4. Функции динамического распределения

Указатели используются для динамического выделения памяти компьютера для хранения данных. Динамическое распределение означает, что программа выделяет память для данных во время своего выполнения. Память для глобальных переменных выделяется во время компиляции, а для нестатических локальных переменных — в стеке. Во время выполнения программы ни глобальным, ни локальным переменным не может быть выделена дополнительная память. Но довольно часто такая необходимость возникает, причем объем требуемой памяти заранее неизвестен. Такое случается, например, при использовании динамических структур данных, таких как связные списки или двоичные деревья. Такие структуры данных при выполнении программы расширяются или сокращаются по мере необходимости. Для реализации таких структур в программе нужны средства, способные по мере необходимости выделять и освобождать для них память.

Приведенный пример показывает работу с двумерным динамическим массивом. Размерность массива выбрана 3 на 3.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
 int m, n;
 //Определяется размерность массива
 m=3;
 n=3;
 //Выделение памяти под массив
 int **matr = new int*[m];
```

```
   for(int i = 0; i<m; i++)
   matr[i] = new int[n];
 //Ввод значений
    for(int i = 0; i<m; i++)
        for(int k = 0; k<n; k++)
            cin>>matr[i][k];
 //Вывод значений массива
    for(int i = 0; i<m; i++)
    {
        for(int k = 0; k<n; k++)
        {
            cout<<matr[i][k]<<" ";
        }
        cout<<endl;
    }
//Удаление
    for(int i = 0; i<m; i++)
    delete []matr[i];
    delete []matr;
    system("pause");
   return 0;
}
```

Задачи на самостоятельное решение

1.В динамическом массиве из 10 чисел проверить, есть ли в нем два одинаковых значения, и вывести их на экран.

2.Задан массив из 15 значений. Получить адреса с третьего по 10 элементы и получить значения на экране.

3.Задан массив на 20 значений. В случае если в исходном массиве количество положительных значений меньше, чем отрицательных, то создавать новый массив для отрицательных значений, в противном случае - создавать массив для положительных значений и переписывать в него соответствующие значения исходного массива.

10. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ - СТРУКТУРЫ, ОБЪЕДИНЕНИЯ И ПЕРЕЧИСЛЕНИЯ

10.1. Структуры данных

Структуры языка C++ представляют поименованную совокупность компонентов, называемых полями, или элементами структуры. Элементами структуры являются:

-переменная указанного типа;

-битовое поле;

-функция

Объявление структуры имеет следующее формальное описание:

```
struct [имя_структуры] {
тип_элемента_структуры  имя_ элемента1;
тип_элемента_структуры  имя_ элемента2;
...
тип_элемента_структуры  имя_ элементаN;
} [список_объявляемых_переменных];
```

Объявление структуры с битовыми полями имеет следующее формальное описание:

```
struct [имя_структуры] {
тип_элемента_структуры
[имя_ элемента1] : число_бит;
тип_элемента_структуры
[имя_ элемента2] : число_бит;
...
тип_элемента_структуры
[имя_ элементаN] : число_бит;
} [список_объявляемых_переменных];
```

Возможно неполное объявление структуры, имеющее следующее формальное описание:

struct имя_структуры;

При отсутствии имени объявляемой структуры создается анонимная структура. При создании анонимной структуры обычно указывается список объявляемых переменных.

Список объявляемых переменных типа данной структуры может содержать:

- имена переменных;
- имена массивов;
- указатели

Пример заполнения структур и работа с ними представлены в следующей программе.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
//Определяем структуру
struct sA
 {
   char a[2];
   int i;
//Связываем структуру с переменной, массивом и
//указателем
 }struA, struB[10], *struC;
struA.i=5;
struB[0].i=20;
struC=&struA;
cout<<struC->i<<endl;
cin.get();
return 0;
}
```

Для использования указателя на структуру ему необходимо присвоить адрес переменной типа структуры.

Размер структуры с битовыми полями всегда кратен байту. Битовые поля можно определять для целочисленных переменных типа int, unsigned int, char и unsigned char. Одна структура одновременно может содержать и переменные, и битовые поля. Если для битового поля не задано имя элемента, то доступ к такому полю не разрешен, но количество указанных бит в структуре размещается.

Структура не может содержать в качестве вложенной структуры саму себя, но она может содержать элемент, являющийся указателем на объявляемую структуру.

Например:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
 struct structA
  {
    //pA указатель на структуру
    struct structA *pA; int iA;
   } sA;
cin.get();
return 0;
}
```

При одновременном объявлении структурного типа, объявлении переменной данного типа и ее инициализации список значений указывается в фигурных скобках в последовательности, соответствующей последовательности определения элементов структуры. Пример объявления структуры с заданными параметрами представлен на следующем листинге:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
// Объявление структурного элемента
  struct position
   {
    int x;      // Объявление элементов x и y
    int y;
   } p_screen = { 50, 100 };
//Выводим значения уже заполненной структуры
cout<<p_screen.x<<endl;
cout<<p_screen.y<<endl;
cin.get();
return 0;
```

При создании переменной типа структуры производятся следующие действия:

-память под все элементы структуры выделяется последовательно для каждого элемента;

-для битовых полей память выделяется, начиная с младших разрядов;

-память, выделяемая под битовые поля, кратна байту;

-общая выделяемая память может быть больше, чем сумма размеров полей структуры.

Выделение памяти под структуру приведенного ниже варианта будет равно 7 байтам информации, так как тип char занимает 1 байт (таких элементов будет 3) и тип float занимает 4 байта.

```
struct structA
 {
  char cA;
  char sA[2];
  float fA;
 };
```

Доступ к элементам структуры осуществляется посредством указания переменной (указателя) и заданного элемента этой структуры. Элементы структуры могут иметь модификаторы доступа: public, private и protected. По умолчанию все элементы структуры объявляются как общедоступные (public). В классе, как ключевом элементе ООП, все члены по умолчанию объявляются как защищенные (private).

Для обращения к отдельным элементам структуры используются операторы: . и ->. Следующий пример демонстрирует обращение к элементам структуры.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    struct structA {
    char c1;
```

```
    char s1[4];
    float f1;} aS1,
    // aS1 - переменная структурного типа
    *prtaS1=&aS1;
    // prtaS1 - указатель на структуру aS1
    struct structB {
    struct structA aS2;
    // Вложенная структура
    } bS1,*prtbS1=&bS1;
    aS1.c1= 'A';
    // Доступ к элементу c1 структуры aS1
    prtaS1->c1= 'A';
    // Доступ к элементу c1 через
    // указатель prtaS1
    (*prtaS1).c1= 'A';
    // Доступ к элементу c1
    (prtbS1->aS2).c1='A';
    cout<<(prtbS1->aS2).c1<<endl;
  cin.get();
  return 0;
}
```

10.2. Объединения

Объединение позволяет размещать в одном месте памяти данные, доступ к которым реализуется через переменные разных типов. Использование объединений значительно экономит память, выделяемую под объекты.

При создании переменной типа объединение память под все элементы объединения выделяется исходя из размера наибольшего его элемента. В каждый отдельный момент времени объединение используется для доступа только к одному элементу данных, входящих в объединение.

Так, компилятор выделит 4 байта под следующее объединение (по максимальному типу из двух предложенных)

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    union unionA {
```

```
    char ch1;
    float f1;
    } a1;
    cin.get();
    return 0;
}
```

Объединения, как и структуры, могут содержать битовые поля.

Инициализировать объединение при его объявлении можно только заданием значения первого элемента объединения. Фрагмент программы представлен далее:

```
   #include <iostream>
   #include <cstdlib>
     using namespace std;
   int main()
    {
      union unionA
      {
      char ch1;
      float f1;
      } a1={ 'M' };
      cout<<a1.ch1;
      cin.get();
     return 0;
    }
```

Доступ к элементам объединения, аналогично доступу к элементам структур, выполняется с помощью операторов . и ->. В следующем фрагменте программы показаны возможности доступа к элементам.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    union TypeNum
    {
    int    i;
    long  l;
    float  f;
    };
    union TypeNum vNum = { 1 };
```

```
 // Инициализация первого элемента объединения i = 1
    cout<< vNum.i;
    vNum.f = 4.13;
    cout<< vNum.f;
    cin.get();
    return 0;
}
```

Элементы объединения не могут иметь модификаторов доступа и всегда реализуются как общедоступные (public).

10.3. Перечисления

Перечисление определяет множество, состоящее из значений, указанных через запятую в фигурных скобках. Перечисление задает для каждого мнемонического названия в указываемом множестве свой индекс.

Перечисление может иметь следующее формальное описание:

```
enum имя_типа {список_значений}
список_объявляемых_переменных;
enum имя_типа список_объявляемых_переменных;
enum (список_элемент=значение);
```

Перечислимый тип описывает множество, состоящее из элементов-констант, иногда называемых нумераторами или именованными константами.

Значение каждого нумератора определяется как значение типа int. По умолчанию первый нумератор определяется значением 0, второй - значением 1 и т.д. Для инициализации значений нумератора не с 0, а с другого целочисленного значения, следует присвоить это значение первому элементу списка значений перечислимого типа.

Пример использования перечислений приведен в следующей программе

```
   #include <iostream>
   #include <cstdlib>
     using namespace std;
   int main()
 {
```

```
    enum eDay{sn, mn, ts, wd, th, fr, st} day1;
    // переменная day1 будет принимать
    // значения в диапазоне от 0 до 6
    day1=st;
    // day1 - переменная перечислимого типа
    int i1=sn;
    // i1 будет равно 0
    day1= eDay(0);
    // eDay(0) равно значению sn
    enum eDay2{sn1=1, mn1, t1s, wd1, th1, fr1, st1}
day2;
    // переменная day2 будет принимать
    // значения в диапазоне от 1 до 7
  cin.get();
  return 0;
      }
```

Для перечислимого типа существует понятие диапазона значений, определяемого как диапазон целочисленных значений, которые может принимать переменная данного перечислимого типа.

Для перечислимого типа можно создавать указатели.

Задачи на самостоятельное решение

1. Подготовить структуру, в которой поместить информацию о группе студентов 1 курса (для примера взять 10 человек), включающую в себя ФИО, возраст и место проживания. Вывести на экран данные о студентах, фамилии которых начинаются с буквы ‘А’.

2. Посредством использования объединений подготовить описание блока данных, который хранит данные о температуре в градусах Цельсия, Кельвина и Фаренгейта, а также осуществляет перевод температур из одного формата в другой.

3. Определить с помощью перечислений значения функции sin(x) для углов 0, 15,30,45,60,90 градусов. Полученные значения вывести на экран.

11. ПОДПРОГРАММЫ В ЯЗЫКЕ С++

Подпрограммы в языке С++ направлены на решение определенной задачи и носят название функции. Функцию можно рассматривать как операцию, определенную пользователем. В общем случае она задается своим именем. Операнды функции, или формальные параметры, задаются в списке параметров, через запятую. Такой список заключается в круглые скобки. Результатом функции может быть значение, которое называют возвращаемым. Об отсутствии возвращаемого значения сообщают ключевым словом void. Действия, которые производит функция, составляют ее тело; оно заключено в фигурные скобки. Тип возвращаемого значения, ее имя, список параметров и тело составляют определение функции.

Объявление функции состоит из типа возвращаемого значения, имени и списка параметров. Вместе эти три элемента составляют прототип. Объявление может появиться в файле несколько раз. Описание прототипа может, например, выглядеть следующим образом:

```
void in(int l1,int l2);
```

В следующем примере приводится функция по определению наибольшего общего делителя для указанных переменных.

```
#include <iostream>
#include <cstdlib>
using namespace std;
// возвращает наибольший общий делитель
int del( int v1, int v2 )
{
    while ( v2 )
    {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

```
int main()
{
    cout<<del(10,15);
    cin.get();
    return 0;
}
```

Выполнение функции происходит тогда, когда в тексте программы встречается оператор вызова. Если функция принимает параметры, при ее вызове должны быть указаны фактические параметры, аргументы. Их перечисляют внутри скобок, через запятую. В рассмотренном примере такой функцией является объект del с параметрами 10 и 15.

Вызов функции может обрабатываться двумя разными способами. Если она объявлена встроенной (inline), то компилятор подставляет в точку вызова ее тело. Во всех остальных случаях происходит нормальный вызов, который приводит к передаче управления ей, а активный в этот момент процесс на время приостанавливается. По завершении работы выполнение программы продолжается с точки, непосредственно следующей за точкой вызова. Работа функции завершается выполнением последней инструкции ее тела или специальной инструкции return.

Функция должна быть объявлена до момента ее вызова, попытка использовать необъявленное имя приводит к ошибке компиляции. Определение функции может служить ее объявлением, но ему разрешено появиться в программе только один раз. Поэтому обычно его помещают в отдельный исходный файл. Иногда в одном файле находятся определения нескольких функций, логически связанных друг с другом. Чтобы использовать их в другом файле, необходимо подключить файл с функциями в указанный файл посредством директивы предпроцессора include.

Тип возвращаемого функцией значения бывает встроенным, как int или double, составным, как int& или double*, или определенным пользователем – перечислением или классом. Можно также использовать специальное

ключевое слово void, которое говорит о том, что функция не возвращает никакого значения. Указанная функция или встроенный массив не могут быть типом возвращаемого значения, однако можно вернуть указатель на первый элемент массива в качестве возвращаемого параметра.

Список параметров не может быть опущен. Функция, которая не требует параметров, должна иметь пустой список либо список, состоящий из одного ключевого слова void. Например, следующие объявления эквивалентны:

```
int func();
int func(void);
```

Список параметров функции состоит из названий типов, разделенных запятыми. После имени типа может находиться имя параметра, хотя это и необязательно. В списке параметров не разрешается использовать сокращенную запись, соотнося одно имя типа с несколькими параметрами:

```
int manip( int v1, v2 );     //ошибка
int manip( int v1, int v2 ); //правильно
```

C++ является строго типизированным языком. Компилятор проверяет аргументы на соответствие типов в каждом вызове функции. Если тип фактического аргумента не соответствует типу формального параметра, то производится попытка неявного преобразования. Если же это оказывается невозможным или число аргументов неверно, компилятор выдает сообщение об ошибке. Именно поэтому функция должна быть объявлена до того, как программа впервые обратится к ней: без объявления компилятор не обладает информацией для проверки типов.

Функции используют память из стека программы. Некоторая область стека отводится функции и остается связанной с ней до окончания ее работы, по завершении которой отведенная ей память освобождается и может быть занята другой функцией. Иногда эту часть стека называют областью активации. Каждому параметру функции отводится место в данной области, причем его размер определяется

типом параметра. При вызове функции память инициализируется значениями фактических аргументов.

Стандартным способом передачи аргументов является копирование их значений, т.е. передача по значению. При этом способе функция не получает доступа к реальным объектам, являющихся ее аргументами. Вместо этого она получает в стеке локальные копии этих объектов. Изменение значений копий никак не отражается на значениях самих объектов. Локальные копии теряются при выходе из функции. Значения аргументов при передаче по значению не меняются. Следовательно, программист не должен заботиться о сохранении и восстановлении их значений при вызове функции.

Массив в C++ никогда не передается по значению, а только как указатель на его первый, точнее нулевой, элемент. Например, объявление

```
void putValues( int[ 10 ] );
```

рассматривается компилятором так, как будто оно имеет вид

```
void putValues( int* );
```

Размер массива неважен при объявлении параметра. Все три приведенные записи эквивалентны:

// три эквивалентных объявления putValues()

```
void putValues( int* );
void putValues( int[] );
void putValues( int[ 10 ] );
```

Передача массивов как указателей имеет следующие особенности:

-изменение значения аргумента внутри функции затрагивает сам переданный объект, а не его локальную копию. Если такое поведение нежелательно, программист должен позаботиться о сохранении исходного значения. Можно также при объявлении функции указать, что она не должна изменять значение параметра, объявив этот параметр константой:

```
void putValues( const int[ 10 ] );
```

-размер массива не является частью типа параметра. Поэтому функция не знает реального размера передаваемого массива.
В качестве примера предлагается листинг программы, в котором представлен механизм передачи заданного массива в функцию пользователя.

```
#include <iostream>
#include <cstdlib>
#define count 7
using namespace std;
// функция поиска максимального элемента в массиве
int f1( int array[],int n )
{
    int max=array[0];
    for (int i=1;i<n;i++)
        if (max<array[i]) max=array[i];
    return max;
}
int main()
{
    int array[count]={4,-5,78,34,1,250,56};
    cout<<f1(array,count)<<endl;
    cin.get();
    return 0;
}
```

Аналогичный пример, но с использованием указателя на массив в функции пользователя.

```
#include <iostream>
#include <cstdlib>
#define count 7
using namespace std;
// функция поиска максимального элемента в массиве
int f1( int *array,int n )
{
    int max=*array;
    for (int i=1;i<n;i++)
        if (max<*(array+i)) max=*(array+i);
    return max;
}
int main()
{
    int array[count]={400,-5,78,34,1,250,560};
```

```
    cout<<f1(array,count)<<endl;
    cin.get();
    return 0;
}
```

Иногда нельзя перечислить типы и количество всех возможных аргументов функции. В этих случаях список параметров представляется многоточием (...), которое отключает механизм проверки типов. Наличие многоточия говорит компилятору, что у функции может быть произвольное количество аргументов неизвестных заранее типов. Пример использования функции с неопределенным числом параметров представлен следующим листингом по суммированию значений перечисленных параметров.

```
#include <iostream>
#include <cstdlib>
using namespace std;
long PP(int n ...)
{
 int *pPointer = &n;
  // Настроились на область памяти с параметрами...
 int Sum = 0;
 for ( ; n; n--) Sum += *(++pPointer);
 return Sum;
}
int main()
{
    long RR;
    RR = PP(6, 1, 2, 3, 4, 5, 6 );
    /*
    Вызвали функцию с 7 параметрами. Единственный
обязательный параметр
    определяет количество передаваемых параметров.
    */
    cout << RR << endl;
    cin.get();
    return 0;
}
```

Задачи на самостоятельное решение

1. В функции произвести вычисление суммы индексов массива, элементами которого являются отрицательные числа.

2. В функции получить индекс максимального элемента массива, минимального и вычислить выражение- индекс минимального элемента в степени максимального без использования готовых методов математической библиотеки.

3. В функции отсортировать массив по убыванию. Использовать указатели по передаче данных в функцию и в теле функции.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Довбуш Г.Ф. Visual C++ на примерах / Г.Ф. Довбуш, А.Д. Хомоненко. – СПб.: БХВ-Петербург, 2007. – 528 с.
2. Пахомов Б.И. C/C++ и MS Visual C++ 2010 для начинающих / Б.И. Пахомов. – СПб.: БХВ-Петербург, 2011. – 736 с.
3. Мюссер Д.Р. C++ и STL: справочное руководство / Д.Р. Мюссер, Ж.Дж. Дердж, А. Сейни. 2-е изд. - М.: ООО ”И.Д. Вильямс”, 2010. – 430 с.
4. Прата С. Язык программирования C++. Лекции и упражнения / С. Прата. 5-е изд. – М.: ООО ”И.Д. Вильямс”, 2007. – 1184 с.
5. Страуструп Б Язык программирования C++ /Б. Страуструп. - М.: Бином, 2011. – 1136 с.
6. Шилдт Г. C++ Базовый курс / Г. Шилдт. 3-е изд. – М.: ООО ”И.Д. Вильямс”, 2010. – 624 с.
7. Марченко А.Л. C++. Бархатный путь / А.Л. Марченко. – М.: Горячая Линия - Телеком, 2005. – 400 с.
8. Коплиен Дж. Программирование на C++ / Дж. Коплиен. – СПб.: Питер, 2005. – 480 с.
9. Roberge J. A laboratory course in C++ structures. 2ed./ J. Roberge, S. Brandl, D. Whittington. Jones and Bartlett, 2003. -411 p.
10. London J. Modeling Derivatives in C++ / London J. Wiley, 2005. -841p.

СОДЕРЖАНИЕ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к практическим работам по дисциплине для студентов направления 230100.64 «Информатика и вычислительная техника» профиля «Системы автоматизированного проектирования в машиностроении» очной формы обучения
Часть 2

Составители
Юров Алексей Николаевич
Паринов Максим Викторович
Чижов Михаил Иванович
Рыжков Владимир Анатольевич

В авторской редакции

Компьютерный набор А.Н. Юрова

Подписано в изданию 10.11.2011.
Уч.-изд. л. 2,7. «С»

ФГБОУВПО «Воронежский государственный технический университет»
394026 Воронеж, Московский просп., 14