Министерство науки и высшего образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.В. ДУРКИН, О.Н. ШЛЫКОВА

ИНФОРМАТИКА

ФУНКЦИИ, ВЕКТОРЫ, СТРОКИ, СТРУКТУРЫ, ФАЙЛЫ

Учебно-методическое пособие

НОВОСИБИРСК
2021

УДК 004(075.8)
Д 841

Рецензенты:

д-р техн. наук, профессор *М.А. Степанов*
д-р техн. наук, профессор *В.П. Разинкин*

Работа подготовлена кафедрой радиоприемных
и радиопередающих устройств и утверждена
Редакционно-издательским советом университета
в качестве учебно-методического пособия

Дуркин В.В.

Д 841 Информатика. Функции, векторы, строки, структуры, файлы: учебно-методическое пособие / В.В. Дуркин, О.Н. Шлыкова. – Новосибирск: Изд-во НГТУ, 2021. − 92 с.

ISBN 978-5-7782-4356-9

Материал пособия поделен на пять тем: функции, векторы, строки, структуры, файлы.

Контрольные задания по темам представлены в 20 вариантах. Каждая тема начинается с изложения теории рассматриваемого вопроса в таком объеме, который позволяет выполнять задание без обращения к дополнительным источникам.

УДК 004(075.8)

Дуркин Валерий Вячеславович
Шлыкова Ольга Николаевна

ИНФОРМАТИКА

ФУНКЦИИ, ВЕКТОРЫ, СТРОКИ, СТРУКТУРЫ, ФАЙЛЫ

Учебно-методическое пособие

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *Л.Н. Киншт*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *С.И. Ткачева*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 19.02.2021. Формат 60 × 84 1/16. Бумага офсетная. Тираж 50 экз.
Уч.-изд. л. 5,34. Печ. л. 5,75. Изд. № 13. Заказ № 229. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20

ISBN 978-5-7782-4356-9

1. ФУНКЦИИ

Необходимо ознакомиться с правилами записи функции, с механизмом передачи параметров в функцию, с перегрузкой функций.

ОБЪЯВЛЕНИЕ И ОПИСАНИЕ ФУНКЦИЙ

Функция – это изолированный блок кода, выполняющий конкретную задачу.

Функции являются первичными «кирпичиками», из которых строится любая программа на языке C++. Имена функций подчиняются тем же правилам, что и имена переменных. Имя функции должно отражать ее предназначение.

Каждая из функций должна быть *определена* или, по крайней мере, *объявлена* до ее использования в программе.

Определение функции имеет вид

тип_возвращаемого_значения имя_функции(список_параметров)

{

операторы тела функции

}

Первая строка определения, содержащая тип возвращаемого значения, имя функции и список параметров, называется *заголовком* функции. Тип возвращаемого значения может быть любым, кроме массива и функции. Могут быть также функции, не возвращающие никакого значения. В заголовке таких функций тип возвращаемого значения объявляется *void.*

Список параметров, заключаемый в скобки, в простейшем случае представляет собой разделяемый запятыми список вида

тип_параметра иденификатор_параметра

Параметры могут быть исходными данными для выполнения алгоритма функции или результатами, возвращаемыми функцией.

Например, заголовок

double FSum(double X1,double X2, int A)

объявляет функцию с именем *FSum* с тремя параметрами *X1, X2* и *A*, из которых первые два имеют тип *double*, а последний – *int*. Тип возвращаемого результата *double*. Имена параметров *X1, X2* и *A* – локальные, т. е. они имеют значение только внутри данной функции и никак не связаны с именами аргументов, переданных при вызове функции. Значения этих параметров в начале выполнения функции равны значениям аргументов на момент вызова функции.

Если функция вызывается раньше, чем она описана в программе, то помимо описания функции в текст программы включается также *прототип* функции – ее предварительное объявление. Прототип представляет собой тот же заголовок функции, но с точкой с запятой в конце. Кроме того, в прототипе можно не указывать имена параметров. Прототипы, размещенные обычно в начале модуля, делают программу более наглядной. Прототип приведенного выше заголовка функции имеет вид

double FSum(double X1,double X2, int A);

или

double FSum(double, double, int);

Теперь рассмотрим описание тела функции. Тело функции пишется по тем же правилам, что и любой код программы, и может содержать объявления типов, констант, переменных и любые выполняемые операторы. Не допускается объявление и описание в теле других функций. Таким образом, функции не могут быть вложены друг в друга.

Надо иметь в виду, что все объявления в теле функции носят локальный характер. Объявленные переменные доступны только внутри данной функции.

Локальные переменные не просто видны только в теле функции, но по умолчанию они и существуют только внутри функции, создаваясь в момент вызова функции и уничтожаясь в момент выхода из функции.

Очень важным оператором тела функции является оператор возврата в точку вызова:

return выражение;

или

return;

Выражение в операторе *return* определяет возвращаемое функцией значение, т. е. результат обращения к функции. Тип возвращаемого значения определяется типом функции. Если функция не возвращает никакого значения, т. е. имеет тип *void*, то выражение в операторе *return* опускается, да и сам оператор *return* необязателен. В теле функции может быть и несколько операторов *return.*

Примеры описаний функций

1. *void print (char *name, int value)* // Ничего не возвращает
   {cout <<"\n"<<name<<value;}// Нет оператора *return*

2. *double min (double a, double b)* // В функции два оператора возврата.
   { *if(a<b) return a*; // Функция возвращает минимальное
   return b; // из значений аргументов
   }

3. *void write (void*) // Ничего не возвращает, ничего не получает
   {*cout<<"\n* НАЗВАНИЕ:";} // Всегда печатает одно и то же.

Здесь заголовок можно упростить *void write()*. Тип возвращаемого значения должен быть указан явно, т. е. написание заголовка в виде *write()* было бы ошибкой.

ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИИ ПО ЗНАЧЕНИЮ И ПО ССЫЛКЕ

Функции могут вызываться из разных частей программы. Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Параметры, перечисленные в заголовке определения функции, называются *формальными*, а записанные в операторе вызова функции – *фактическими* (или *аргументами*).

При обращении к функции (ее вызове) формальные параметры заменяются фактическими (аргументами), причем язык *C++* требует соответствия этих параметров по типам. Если же типы формальных и

фактических параметров не совпадают, то там, где это возможно, происходит преобразование типов. Если же такое преобразование невозможно, то компилятор выдает сообщение об ошибке.

Согласование по типам между формальными и фактическими параметрами требует, чтобы до первого обращения к функции было помещено либо ее определение, либо ее объявление (прототип). Наличие прототипа позволяет компилятору выполнять контроль соответствия типов параметров.

Существует два способа передачи параметров в функцию: по **значению** и по **ссылке**. Рассмотрим первый из них. Список параметров, передаваемый в функции, как было показано в предыдущем разделе, состоит из имен параметров и указаний на их тип. Например, в заголовке

double FSum(double X1,double X2, int A)

указано три формальных параметра *X1, X2, A* и определены их типы. Обращение к такой функции может иметь вид

double Z = FSum(Y, Y2, 5);

Это только один из способов передачи параметров в функцию, называемый **передачей по значению**. Работает он так. В момент вызова функции в памяти создаются временные переменные с именами *X1, X2, A*, и в них копируются значения аргументов *Y, Y2* и константы *5*. На этом связь между аргументами (фактическими параметрами) и переменными *X1, X2, A* (формальными параметрами) разрывается. Вы можете изменять внутри функции значения *X1, X2* и *A*, но это никак не отразится на значениях аргументов, т. е. аргументы надежно защищены от непреднамеренного изменения формальных параметров внутри функции.

К недостаткам такой передачи параметров по значению относятся, во-первых, затраты времени на копирование значений и затраты памяти для хранения копии и, во-вторых, невозможность из функций изменять значения некоторых аргументов, что во многих случаях желательно. Действительно, рассмотрим классический пример по обмену значениями между двумя переменными.

Пример 1

```
#include <iostream>
 #include <iomanip>
```

```
 using namespace std;
//Прототип функции swap. Передача параметров по значению
void swap(int x,int y);
void main ()
{
 setlocale(LC_ALL,"rus_rus.1251");
 int i,j;
 i = 10;
 j = 20;
 cout<<"Исходные значения переменных i и j:
"<<i<<setw(5)<<j<<endl;
 swap(i,j);   //Вызов функции. Фактические параметры – значения
переменных i и j
 cout<<"Значения переменных i и j после обмена:
"<<i<<setw(5)<<j<<endl;
}
// Определение функции swap
void swap(int x,int y)
{   int temp;
   temp =x;
   x = y;
   y = temp;
}
```

Результат

Исходные значения переменных *i* и *j*: 10 20
Значения переменных *i* и *j* после обмена: 10 20

В этом примере переменной *i* было присвоено начальное значение 10, а переменной *j* – 20. Затем была вызвана функция *swap,* в которую были переданы значения этих переменных. Как вы видите, никакого обмена не произошло, поскольку функция *swap* работает с копиями переменных *i u j*, поэтому фактические параметры не изменились.

Возможен и другой способ передачи параметров – **вызов по ссылке.** В случае вызова по ссылке в функцию передается адрес элемента.

Вызов по ссылке можно осуществить двумя способами: с помощью *указателей* и с помощью *ссылочных параметров.*

При использовании *указателей* в качестве формальных параметров им передаются адреса соответствующих фактических параметров. Рассмотрим решение предыдущего примера, используя указатели.

Пример 2

```
//Прототип функции swap с использованием указателей
void swap(int  *x,int  *y);// Формальные параметры – указатели
void main ()
{
 . . . . .
 swap(&i, &j); // Вызов функции. Фактические параметры – адреса
                // переменных i и j
 . . . . .
}
// Определение функции swap
void swap(int  *x,int  *y)
{  int temp;
 temp =*x; // Временно сохраняем значение, расположенное по адресу x
 * x = *y;     // Помещаем значение, хранимое по адресу y, по адресу x
  *y = temp; // Помещаем значение, которое раньше хранилось
             // по адресу x, по адресу y
}
```

Результат

Исходные значения переменных *i* и *j*: 10 20

Значения переменных *i* и *j* после обмена: 20 10

Обмен произошел, так как функции *swap* при вызове были переданы адреса переменных, а не их значения.

Ссылочный параметр – это специальный тип указателя, который позволяет работать с указателем как с обычной переменной. При использовании *ссылочного параметра* функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции, а именно при выполнении операций над ссылочным параметром, обеспечивается его автоматическое разыменование, и поэтому программисту не нужно использовать операторы, работающие с указателями.

Ссылочный параметр может объявляться с помощью символа «&», который должен предшествовать имени параметра в объявлении функции, естественно, такое же обозначение используется в списке типов параметров в заголовке функции.

Операции, выполняемые над ссылочным параметром, влияют на аргумент (фактический параметр), а не на ссылочный параметр.

Чтобы лучше понять механизм действия ссылочных параметров, рассмотрим предыдущий пример.

Пример 3

```
//Прототип функции swap с использованием ссылочных параметров
void swap(int &x,int &y);// x и y – ссылочные параметры
void main ()
{ . . . . .
  swap(i,j);//Вызов функции
  . . . . .
 }
// Определение функции swap
void swap(int &x,int &y)
{   int temp;
   temp = x; // Сохраняем значение, расположенное по адресу x
   x = y;     //  Помещаем значение , хранимое по адресу y, по адресу x
   y = temp; //Помещаем значение, которое раньше хранилось
                 // по адресу x, по адресу y
}
```

Обратите внимание, что объявление *x* и *y* ссылочными параметрами освобождает вас от необходимости использовать оператор «*» при организации обмена. Кроме того, вызывается функция *swap* обычным способом – передачей в нее имен аргументов. Результат работы программы тот же, что и в примере 2, т. е. обмен произведен.

Как уже отмечалось, функция может возвращать только одно значение. Передача же аргументов по ссылочному параметру или с помощью указателей эквивалентна возврату нескольких значений. В этом случае функция по-прежнему возвращает одно значение, но теперь переданные в функцию параметры можно изменять. Таким образом, фактически возвращаемых значений может быть несколько!

ПЕРЕДАЧА В ФУНКЦИЮ ОДНОМЕРНОГО МАССИВА

Массивы могут быть параметрами функций, и функции могут возвращать указатель на массив в качестве результата.

Не надо путать размер массива и размерность массива. Размер массива – количество элементов, размерность массива – количество индексов. Например, mass[3][4][5] – трехмерный массив размером $3 \times 4 \times 5 = 60$ элементов.

Для передачи в функцию одномерного массива достаточно передать адрес его нулевого элемента, носителем которого является само имя этого элемента, т. е. если *x* – имя массива, то выражения *&x[0]* и *x* эквиваленты. В качестве формального параметра в определении функции и в прототипе нужно использовать указатели на переменные соответствующего типа. Можно в этом качестве использовать и описание массива (без указания размера и без списка начальных значений). Например, необходимо написать и протестировать функцию, которая определяет произведение и сумму членов целочисленного одномерного массива.

Пример 4

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

 //Прототип функции
```

```
void fun(int *pt,int n,int &sum,int &pr);
void main ()
{  setlocale(LC_ALL,"rus_rus.1251");
  int mass[5]={1,2,3,4,5},s,p;
  fun(mass,5,s,p);
  cout<<"Сумма и произведения равны
:"<<setw(3)<<s<<setw(5)<<p<<endl;
 }
// Определение функции
void fun(int *pt,int n,int &sum,int &pr)
{          int i;   sum=0, pr=1;
          for(i = 0; i<n; i++)
             {
                    pr*= pt[i];
                     sum+=pt[i];
             }
}
```

Результат

Сумма и произведения равны: 5 120

Функция *fun* имеет четыре формальных параметра: *pt* – указатель на переменную такого же типа, что и обрабатываемый массив *mass[]*; *n* – количество элементов массива; *sum, pr* – сумма и произведение элементов массива. Причем и *sum* и *pr* являются ссылочными параметрами, при обращении к функции *fun*() в них автоматически передаются адреса переменных *s* и *p*. Поэтому операции над *sum* и *pr* изменяют фактические параметры *s* и *p*.

В этом примере мы использовали все три способа передачи параметров в функцию: через указатель (*pt→mass*), по значению (*n→5*) и с помощью ссылочных параметров, который мы обсудили выше.

Запись *pt[i]* следует понимать как взятие значения по адресу *pt+i*, т. е.

*pt[i] == *(pt + i) или &pt[i] == pt + i.*

Ничего в программе не изменится, если прототип и заголовок функции в определении записать следующим образом:

void fun(int pt[],int n,int &sum,int &pr).

ПЕРЕДАЧА В ФУНКЦИЮ ДВУМЕРНОГО МАССИВА (МАТРИЦЫ)

Многомерный массив в соответствии с синтаксисом языка есть массив массивов, т. е. массив, элементами которого служат массивы. Например, двумерный массив

int mass[4][3]

состоит из четырех элементов, каждый из которых – одномерный массив из трех элементов. Как и в случае одномерных массивов, доступ к элементам многомерных массивов возможен с помощью индексированных переменных и с помощью указателей. Необходимо помнить, что при добавлении целой величины к указателю его внутреннее значение изменится на «длину» элемента соответствующего типа. Для приведенного массива *mass* – указатель, поставленный в соответствие элементу типа *int[3]*. Добавление единицы к указателю *mass* приводит к изменению значения адреса на величину

*sizeof(int)*3== 6* байт

Именно поэтому **(mass + 1)* есть адрес элемента *mass[1],* т. е. указатель на одномерный массив, состоящий из трех элементов и отстоящий от начала массива на 6 байт. Если рассматривать двумерный массив как матрицу, то *mass[0]* – это адрес нулевой строки (ее первого байта) матрицы, *mass[1]* – адрес первой строки и т. д.

Для передачи в функцию двумерного массива достаточно передать адрес его начала в виде *mass[0]* и его размеры (*n* и *m* – количество строк и столбцов), т. е. фактически вместо работы с двумерным массивом мы будем работать с одномерными массивами – строками матрицы, где *n* – количество этих одномерных массивов, *m* – их размер, *а mass[i]* – адреса. Тогда для передачи массива в качестве формального параметра можно использовать указатель на переменную такого же типа, что и массив.

Значит, если *int *pt* – формальный параметр, которому при обращении к функции передается адрес начала массива *mass[0],* то доступ

к любому элементу двумерного массива в теле функции осуществляется согласно выражению

**(pt + i*m + j) == pt[i*m + j],*

где *i* – индекс строки, *j* – индекс столбца.

ПЕРЕГРУЖЕННЫЕ ФУНКЦИИ

Перегруженные функции обычно используются для выполнения сходных операций над различными типами данных.

Предположим, вам требуется написать функцию, которая определяет максимальное значение в одномерном массиве типа *double.* Прототип этой функции будет иметь вид

*double maxdouble (double *x, int len);*

Затем возникает аналогичная задача, но для массива типа *int.* Вы создаете другую функцию, похожую на первую, с прототипом

*int maxint(int *x, int len);*

Естественно встает вопрос: нельзя ли, вне зависимости от типов аргументов, использовать одну функцию, но с разными версиями? При вызове такой функции компилятор находит подходящую для конкретного случая версию, в зависимости от списка аргументов, который вы передаете в функцию. Оказывается, такой механизм в C++ существует, и он носит название – **перегрузка функций**.

В нашем случае функции *maxdouble* и *maxint* целесообразно объединить в одну перегруженную функцию *max.* Типами аргументов таких функций часто служат **шаблоны**.

Объявление и определение шаблона функции начинается ключевым словом *template*, за которым следует заключенный в угловые скобки и разделенный запятыми непустой список *параметров* шаблона. Эта часть объявления или определения обычно называется *заголовком шаблона.*

Каждый параметр шаблона состоит из служебного слова *typename,* за которым следует идентификатор. В заголовке шаблона имена параметров шаблона должны быть уникальны.

Стоит упомянуть, что вместо *typename* можно использовать *class*, и они полностью эквивалентны.

Следом за заголовком шаблона располагается прототип или определение функции − все зависит от контекста программы. Как известно, у прототипа и определения функции также имеется собственный заголовок. Этот заголовок состоит из типа возвращаемого значения, имени функции и списка параметров. В этом списке они играют роль спецификаторов типа. Объявление параметров, у которых в качестве спецификатора типа используется идентификатор из списка параметров шаблона, называется *шаблонным* параметром. Наряду с шаблонными параметрами в список параметров функции могут также входить параметры основных и производных типов.

Шаблон функции *max* можно определить следующим образом.

```
template<typename T> //Заголовок шаблона. T – параметр шаблона
T max( T  *x,  int len ) //  Заголовок функции. *x – шаблонный
                          // параметр
    {
     T max=x[0]   ;
        for(int i=1;i<len;i++)
              if(max<x[i])
             max=x[i];
        return max;
    }
```

В заголовке шаблона этой функции объявляется единственный параметр шаблона *T* как тип данных, который должен проверяться функцией *max*. В следующем далее заголовке функции этот параметр *T* использован для задания типа возвращаемого значения (*T max*) и для задания типа указателя *x*. В теле функции этот же параметр *T* использован для указания типа локальной переменной *max*.

В приведенном ниже примере показано использование шаблона функции *max* для решения поставленной задачи.

Пример 5

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

```
//Шаблон функции для поиска максимального элемента
template<typename T> T max(T *x,int len)
        {…… }
ume main()
    {    int small[] ={1,24,34,22,33};
          double large[] = {23.0, 1.4,2.456,345.5,12.0,2981.1};
          int lensmall(sizeof small/sizeof small[0]); // Определение
                                                      // количества
         int lenlarge(sizeof large/sizeof large[0]); //элементов в массивах
        cout<<max(small,lensmall)<<endl;
        cout<<max(large,lenlarge)<<endl;
        return 0;
       }
```

Результат

34

2981.1

Когда компилятор обнаруживает вызов функции *max* в исходном коде программы, то тип данных, переданных в функцию, подставляется всюду вместо *T* в определении шаблона и C++ создает законченную функцию для определения максимального значения в одномерном массиве. Затем эта созданная функция компилируется.

Например, при вызове функции *max* для целочисленного массива *small[]* компилятор сгенерирует функцию:

```
int   max( int  *x,  int len )
        {  int  max=x[0];
          for(int i=1;i<len;i++)
                    if(max<x[i])
                   max=x[i];
           return max;
      }
```

ГЕНЕРАЦИЯ (ПСЕВДО) СЛУЧАЙНЫХ ЧИСЕЛ

В обучающих задачах с массивами часто возникает необходимость заполнения этих массивов случайными числами. Наиболее просто это можно решить с помощью функции *rand,* ее прототип:

int rand (void);

Она генерирует псевдослучайное целое число на интервале значений от 0 до *RAND_MAX*. Последнее является константой, которая варьируется в зависимости от реализации языка, но в большинстве случаев составляет 32767.

Если же вам нужны целые случайные числа в диапазоне от 0 до *n*, то необходимо использовать операции деления по модулю

rand() % (n+1).

Если же нижняя граница диапазона не 0, а *m*, то предыдущее выражение примет вид

rand() % (n+1-m)+m.

Например, вам нужно сгенерировать псевдослучайные числа в интервале от –5 до 5, тогда

rand() % (5+1+5) – 5 = rand() % 11 – 5.

В этом примере функция *rand()* генерирует случайные числа от 0 до 10, и после вычитания 5 они преобразуются в случайные числа от –5 до 5.

К сожалению, функция *rand* генерирует последовательность чисел хоть и псевдослучайную, но всегда одинаковую, что делает тестирование программы не совсем корректным. Чтобы эту проблему решить, необходимо воспользоваться функцией *srand.* Ее прототип:

void srand (unsigned int seed);

Задавая разные значения переменной *seed*, мы и получим разные последовательности от функции *rand.* Как задать случайные значения переменной *seed*? Можно использовать функцию *time.* Эта функция, будучи вызвана с нулевым указателем в качестве параметра, возвращает количество секунд, прошедших с 1 января 1970 года на данный момент.

Значение этой функции мы можем в качестве параметра передать в функцию *srand* (при этом выполняется неявное приведение типа) и при каждом запуске программы получать разные псевдослучайные последовательности.

Для использования функций *rand* и *srand* нужно подключить заголовочный файл <*cstdlib*>, а для использования *time* – файл <*ctime*> (пример 6).

Пример 6

```
#include<iostream>
#include<cstdlib>
#include<ctime>
#define RND  (rand()/double(RAND_MAX))
using namespace std;
int main()
{   setlocale(LC_ALL,"rus_rus.1251");
   srand(time(NULL));
   cout << "10 целых случайных чисел в интервале от 1 до 100: " << endl;
   for(int i=0;i<10;i++) cout << rand() % 100 + 1 << " ";
  cout << "\n\n10 вещественных случайных чисел в интервале от –5 до 5: " << endl;
  for(int i=0;i<10;i++) cout<< RND*10 - 5<<"  ";
}
```

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Составить программу, состоящую из двух функций. В главной функции с помощью генератора случайных чисел задать два массива указанного типа. Создать функцию, реализующую алгоритм вашей задачи, и протестировать ее для обоих массивов. Результаты тестирования вывести на экран монитора в главной функции.

2. Решить поставленную задачу при условии, что один из заданных массивов будет иметь тип *int,* а второй – *double*, т. е. создать перегруженную функцию, которая и позволила бы реализовать нужный алгоритм для массивов обоих типов. Результаты тестирования перегруженной функции вывести на экран монитора в главной функции.

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Заданы вещественные массивы *A[6]* и *B[7]*. Найти максимальный и минимальный элемент в каждом из массивов и сумму элементов между ними.

Вариант 2

Заданы целые массивы *X[5], Y[6]*. Найти число нулевых элементов и число отрицательных элементов в каждом из массивов.

Вариант 3

Заданы вещественные массивы *G[10], R[8]*. Найти число элементов >2 и число элементов <1 в каждом из массивов и их произведение.

Вариант 4

Заданы целые массивы *C[8], D[12]*. Найти сумму положительных элементов и произведение отрицательных элементов в каждом из массивов.

Вариант 5

Заданы массивы *E[11], F[14]*. Найти произведение отрицательных и произведение положительных элементов в каждом из массивов.

Вариант 6

Заданы массивы *K[10], M[12]*. Найти произведение элементов, расположенных после первого отрицательного элемента в каждом из массивов.

Вариант 7

Заданы целые массивы *N[8], M[12]*. Найти минимальный элемент и сумму элементов, расположенных до него, в каждом из массивов.

Вариант 8

Заданы вещественные массивы *T[14], R10[]*. Изменить массивы так, чтобы значение каждой ячейки было разностью между предыдущим значением и номером ячейки (например, *T[i]=T[i] – i*, где *i* – номер ячейки массива).

Вариант 9

Заданы целые массивы *B1[4][3], B2[3][4]*. Найти сумму и произведение минимальных элементов в каждой строке в каждом из массивов.

Вариант 10

Заданы вещественные массивы *Z1[8], Z2[14]*. Найти сумму элементов в ячейках с четными номерами и произведение элементов в ячейках с нечетными номерами в каждом из массивов.

Вариант 11

Заданы вещественные массивы *A1[4][3], A2[3][4]*. Найти сумму элементов второго столбца и произведение элементов второй строки в каждом из массивов.

Вариант 12

Заданы целые массивы *A3[5][5], A4[5][5]*. Найти сумму и произведение элементов на главной диагонали в каждом из массивов.

Вариант 13

Заданы вещественные массивы *B1[4][3], B2[3][4]*. Найти минимальный и максимальный элементы в каждом из массивов.

Вариант 14

Заданы целые массивы *C1[3][4], C2[2][3]*. Найти сумму элементов после первого нулевого элемента (элементы массивов рассматривать последовательно по строкам) в каждом из массивов.

Вариант 15

Заданы вещественные массивы *D1[5][2], D2[4][2]*. Поменять в них вторую и третью строки.

Вариант 16

Заданы целые массивы *E1[3][3], E2[4][4]*. Вычислить сумму и произведение элементов во второй строке каждого из массивов.

Вариант 17

Заданы целые массивы *K1[3][2], K2[2][3]*. Найти произведение и сумму максимальных элементов в каждом столбце в каждом из массивов.

Вариант 18

Сформировать единичные матрицы в массивах *L1[4][4], L2[3][3]*.

Вариант 19

Заданы вещественные массивы *M1[3][4], M2[2][3]*. Найти сумму элементов, больших 2, и произведение элементов, меньших 1, в каждом из массивов.

Вариант 20

Заданы массивы *N1[3][3], N2[3][3]*. Найти сумму элементов под главной диагональю в каждом из массивов.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как записать заголовок функции?
2. Назначение оператора *return.*
3. Как вызвать функцию?
4. Какое соотношение между формальными и фактическими параметрами должно выполняться обязательно?
5. Форма записи и назначение прототипа функции.
6. Как вернуть из вызываемой функции несколько значений?
7. В чем разница между ссылочным параметром и указателем?
8. Как передать в функцию одномерный массив?
9. Как передать в функцию двумерный массив?
10. Что такое перегрузка функций?
11. Как записывается заголовок шаблона?
12. Какой параметр называется шаблонным?
13. Как сгенерировать последовательность вещественных псевдослучайных чисел в диапазоне от –2 до 7?

2. ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ *VECTOR*

Необходимо ознакомиться со структурой стандартной библиотеки шаблонов C++ STL и получить практические навыки работы с последовательным контейнером *vector*.

ДИНАМИЧЕСКИЕ МАССИВЫ

Динамическим массивом называют массив с переменным размером, т. е. массив, количество элементов которого может изменяться во время выполнения программы.

Для создания динамических массивов применяются операции ***new*** и ***delete***. Рассмотрим фрагмент кода создания одномерного динамического массива на 10 элементов:

```
double *pt = new double [10];
```

Здесь *pt* – указатель на выделенный участок памяти под массив вещественных чисел типа *double*.

После того как динамический массив стал ненужным, необходимо освободить участок памяти, который под него выделялся.

```
delete [] pt;
```

Это простейший случай, когда программа непосредственно получает требуемый размер массива и создает его. Но чаще размер массива заранее не известен, он должен меняться при его заполнении. Как это происходит? Если по мере заполнения массива вся выделенная память окажется занятой, то при добавлении очередного элемента выделенную ранее память нужно освободить, все хранящиеся в массиве значения сохранить во временном массиве. Затем выделить память под массив большего размера, в него поместить сохраненные значения и старую память освободить.

При традиционном подходе к работе с динамическими массивами (операции *new* и *delete*) вся эта процедура возлагается на пользователя.

Кроме того, что она громоздкая, она еще и ответственная. Некорректное использование операторов *new* и *delete* может привести к непредсказуемым последствиям: зависаниям программы, порче содержимого памяти и др.

В настоящее время практически отпала необходимость использовать подобные динамические массивы, поскольку в стандартной библиотеке шаблонов C++ (STL) разработаны такие структуры данных, называемые ***контейнерами***, которые по существу являются динамическими массивами, но с значительно расширенными функциональными возможностями. Речь идет о последовательном контейнере *vector*. В этом контейнере изменение размера массива при его заполнении происходит автоматически, невидимо для пользователя. Кроме того, *vector* входит в семейство контейнеров STL, поэтому в вашем распоряжении оказывается весь мощный арсенал алгоритмов STL, работающих с этими контейнерами.

Чтобы убедить вас в преимуществе *vector* перед традиционными динамическими массивами, рассмотрим простой пример.

Пример 1

Создать одномерный динамический массив произвольного размера. Размер массива должен изменяться при его заполнении.

```
// Традиционный подход
const int M =  5;
    int temp,   * Z=new int[M] ; // Массив начального размера
   for(int i=0;1;i++)
    {  cout<<i+1<<"-й  элемент ";
       cin>>temp;
      if(temp==0)
            break;   // Ограничитель ввода
     else(i%M==0) // Массив уже заполнен?
      {     int *q=new int[i+M]; // Создаем новый массив и переписываем
           for(int j=0;j<i;j++) // в него элементы массива p
```

```
            q[j]=Z[j];
          delete[] Z;   //Уничтожаем старый массив
          Z=q;          // Z теперь указывает на новый массив
      }
      Z[i]=temp;
    }
    delete[] Z;// Освобождаем память
}
// Код программы при использовании контейнера vector
#include <vector>
vector<int>v;
 int num;
  for(int i=0;;i++)
    {  cout<<i+1<<"-й  элемент ";
       cin>>num;
  if(num==0)
        break;
       v.push_back(num);// Добавить num в конец вектора
}
```

Преимущества второго подхода настолько очевидны, что в комментариях не нуждаются. Все процессы, связанные с изменением размеров массива и перераспределением памяти, скрыты от пользователя и выполняются автоматически.

СТРУКТУРА СТАНДАРТНОЙ БИБЛИОТЕКИ ШАБЛОНОВ (STL)

Стандартная библиотека шаблонов (*Standard Template Library*) является существенной частью стандартной библиотеки C++. Использование STL дает возможность создавать более надежные, более переносимые и более универсальные программы, а также сократить расходы на их разработку.

Ядро стандартной библиотеки шаблонов включает три основных элемента:

контейнеры,
алгоритмы,
итераторы.

Они работают совместно один с другим, предоставляя тем самым готовые решения различных задач программирования.

Контейнеры – это объекты, которые могут хранить множество элементов – объектов некоторого типа.

Каждый раз, когда в программе возникает необходимость оперировать множеством элементов, в дело вступают контейнеры. В языке С (не в С++) существовал только один встроенный тип контейнера: массив. В STL имеется несколько различных типов контейнеров. Простейшим и самым распространенным контейнером является *vector.*

Алгоритмы обрабатывают содержимое контейнеров. Их возможности включают средства инициализации, сортировки, поиска и преобразования содержимого контейнеров.

Итераторы – это объекты, которые в той или иной степени действуют подобно указателям. Итераторы обеспечивают интерфейс между контейнерами и алгоритмами. Они позволяют циклически опрашивать содержимое контейнера практически так же, как это делается с помощью указателя при циклическом опросе элементов массива. Итераторы обрабатываются аналогично указателям

ПОСЛЕДОВАТЕЛЬНЫЙ КОНТЕЙНЕР *VECTOR*

Последовательными называются контейнеры, в которых каждый элемент занимает определенную позицию.

Вектор – единственный в STL обратно-совместимый с чистым С-контейнером. Это означает, что вектор, по сути дела, является обычным динамическим массивом, но с рядом дополнительных функций.

Для доступа к его элементам можно использовать стандартное обозначение индексации массивов. Объектами вектора могут быть как базовые типы переменных (*int, float, char* и т. д.), так и строки.

Каждый контейнер, в том числе и вектор, имеет свои собственные функции (методы), позволяющие выполнять определенный круг базовых операций. Однако *алгоритмы* STL, обрабатывающие данные, содержащиеся в контейнерах, позволяют выполнять более расширенные или более сложные действия.

Вектор обеспечивает *произвольный доступ* к своим элементам. Итераторы векторов являются итераторами произвольного доступа, что позволяет применять к векторам все алгоритмы STL.

Чтобы использовать вектор в программе, необходимо включить в нее заголовочный файл *<vector>:*

#include <vector>

Объявление вектора выглядит следующим образом:

vector<тип_элемента> *имя;*

Например,

vector<int> n;

n – вектор, который может хранить целые числа.

При объявлении вектора его можно одновременно и инициализировать (табл. 1).

Таблица 1

Способы инициализации векторов

Операция	Описание
vector <Type> v	Создает пустой вектор, не содержащий ни одного элемента
vector<Type> v1(v2)	Создает копию другого вектора того же типа (с копированием всех элементов)
vector<Type> v(n)	Создает вектор из *n* элементов, создаваемых конструктором по умолчанию
vector<Type> v(n,elem)	Создает вектор, инициализируемый *n* копиями элемента *elem*

Здесь *Type* – тип элементов вектора.

Пример 2

```
#include <vector>
int main()
{  setlocale(LC_ALL,"rus_rus.1251");
   vector<int> v; // Создаем пустой вектор
   vector<int> v1(10); // Создаем вектор из десяти элементов,
                       // начальные значения которых равны нулю
```

```
    for(int i = 0; i < 10; i++)   v1[i] = (i+1)*(i+1);//Заполняем вектор v1
    vector<int>v3(v1);// Копируем вектор v1 в вектор v3;
    v=v3;// Присваиваем элементам вектора v значения элеметов
          // вектора v3
   cout<<"\nЭлементы вектора v"<<endl;
    for(int i = 0; i < 10; i++) cout<<setw(5)<<v[i];
   returen 0;
}
```

Результат

```
Элементы вектора v
    1    4    9   16   25   36   49   64   81 100
```

Одной из самых замечательных особенностей контейнеров STL является автоматическое наращивание памяти в соответствии с объемом внесенных данных. Чтобы разобраться с этим вопросом, остановимся более подробно на функциях *size, capacity* и *reserve(n)* (табл. 2).

Таблица 2

Немодифицирующие операции над векторами

Операция	Описание
v.size()	Возвращает фактическое количество элементов
v.empty()	Возвращает *true*, если контейнер пуст, и *false* в противном случае
v.max_size()	Возвращает максимально возможное количество элементов
v.capacity()	Возвращает количество зарезервированной (выделенной) памяти
v.reserve(n)	Резервирует память, необходимую для добавления *n* элементов в вектор

Функция *size* возвращает текущее количество элементов в контейнере. Она *не сообщает,* сколько памяти контейнер выделил для хранящихся в нем элементов.

Функция *capacity* сообщает, сколько элементов поместится в выделенной памяти. Речь идет об *общем* количестве элементов, а не о том, сколько *еще* элементов можно разместить без расширения контейнера. Если *size* и *capacity* возвращают одинаковые значения, значит, в контейнере не осталось свободного места и следующая вставка (*insert, push_back* и другие, см. пример 6) даже одного элемента вызовет процедуру перераспределения памяти.

Таким образом, динамическое увеличение контейнера порой обходится довольно дорого (при больших объемах). Естественно, эту операцию хотелось бы выполнять как можно реже. Тем более что при каждом выполнении перечисленных операций все итераторы, указатели и ссылки на содержимое *vector* становятся недействительными. Для решения этой проблемы надо использовать функцию *reserve(n).*

Функция *reserve(n)* устанавливает минимальную емкость контейнера равной *n* при условии, что *n* не меньше текущего размера. Таким образом, для предотвращения лишних затрат следует установить достаточно большую емкость контейнера функцией *reserve,* причем сделать это нужно как можно раньше, желательно сразу же после конструирования контейнера.

Чтобы разобраться с этими тонкостями, рассмотрим пример 3.

Пример 3

```
vector<int> v; // Создаем пустой вектор
cout<<"Пустой вектор"<<endl;
cout<<"Емкость вектора  "<<v.capacity()<<"\nКоличество элементов
 "<<v.size()<<endl;
   int s = 0;
// Заполняем вектор
  for (int i=1; i<=1000; ++i)
{    if(v.size()==v.capacity()) s++;
    v.push_back(i);}
cout<<"\nПосле заполнения вектора"<<endl;
cout<<"Емкость вектора  "<<v.capacity()<<"\nКоличество элементов
"<<v.size()<<endl;
cout<<"Число перерасределения памяти = "<<s<<endl;
```

*cout<<"\n*Добавим в конец вектора еще 67 элементов*"<<endl;*

//v.reserve(1067);

v.insert(v.end(),67,1);

*cout<<"*Емкость вектора *"<<v.capacity()<<"\n*Количество элементов *"<<v.size()<<endl;*

Результат

Пустой вектор
Емкость вектора 0
Количество элементов 0

После заполнения вектора
Емкость вектора 1066
Количество элементов 1000
Число перераспределений памяти 18

Добавим в конец вектора еще 67 элементов
Емкость вектора 1599
Количество элементов 1067

Анализ

Как и ожидалось, в пустом векторе число элементов и его размер равны нулю. В процессе заполнения вектора 18 раз произошло перераспределение памяти, т. е. 18 раз происходила процедура, описанная выше. В результате размер (емкость) вектора превысил количество элементов в нем на 66 элементов. Предположим, что нам нужно добавить в вектор еще 67 элементов, т. е. необходимое количество элементов в векторе превышает его текущий объем всего на один элемент. В этом случае опять происходит перераспределение памяти и объем памяти увеличивается в 1,5 раза. Таким образом, вместо нужного объема 1067 мы получили 1599.

При известном количестве элементов вектора возникшие трудности (большое количество перераспределений памяти и избыточный размер вектора) можно решить просто:

vector<int> v ;

v.reserve(1067);

Тогда число перераспределений памяти будет равно нулю, а емкость – 1067, т. е. равной количеству элементов. Однако вектор – это динамический массив, и размеры массива заранее не известны. Тогда можно подстраховаться и зарезервировать заведомо больший размер вектора (например, *v.reserve(2000)*), а чтобы вектор не удерживал ненужную память (2000 – 1067), необходимо сократить емкость от максимальной (2000) до используемой в настоящий момент (1067). Подобное сокращение емкости обычно называется «сжатием по размеру». Это сжатие в векторе *v* можно выполнить с помощью оператора: *vector<int>(v).swap(v);* (табл. 3).

Таблица 3

Операции присваивания для векторов

Операция	Описание
v1 = v2	Присваивает *v1* все элементы *v2*
v.assign(n,elem)	Присваивает *n* копий заданного элемента
v.assign(v1.beg,v1.end)	Присваивает элементы интервала *[beg,end)*
v1.swap(v2)	Меняет местами содержимое *v1* и *v2*

Следующий пример показывает, как *assign* может быть использована для копирования одного вектора в другой.

Пример 4

```
vector<int> v1;
for( int i = 0; i < 10; i++ )    v1.push_back( i ); // Заполнение вектора
vector<int> v2;
// Копирование вектора v1 в вектор v2
 v2.assign( v1.begin(), v1.end() );
 for( int i = 0; i < v2.size(); i++ )       cout << v2[i] << " ";
```

Результат

0 1 2 3 4 5 6 7 8 9

В табл. 4 перечислены все операции прямого обращения к элементам векторов. Как принято в C и C++, первому элементу вектора соответствует индекс 0, а последнему – индекс *size()-1*. Таким образом, *n*-элементу соответствует индекс *n-1*.

Таблица 4

Операции обращения к элементам вектора

Операция	Описание
v.at(idx)	Возвращает элемент с индексом *idx* (при недопустимом значении индекса генерируется исключение *out_of_range*)
v[idx]	Возвращает элемент с индексом *idx* (без интервальной проверки!)
v.front()	Возвращает первый элемент (без проверки его существования!)
v.back()	Возвращает последний элемент (без проверки его существования!)

Векторы поддерживают стандартный набор операций для получения итераторов (табл. 5). Итераторы векторов относятся к категории итераторов произвольного доступа. Это означает, что с векторами в принципе могут использоваться все алгоритмы *STL*.

Таблица 5

Операции получения итераторов

Операция	Описание
v.begin()	Возвращает итератор произвольного доступа для первого элемента
v.end()	Возвращает итератор произвольного доступа для позиции за последним элементом

Пример 5

```
vector<char> v; // Создание массива нулевой длины
//Помещаем значения в вектор
```

```
for (int i = 0; i < 10; i++)  v.push_back('A' + i);
// Получаем доступ к содержимому вектора с помощью итератора
vector<char>::iterator it;
for (it = v.begin(); it != v.end(); it++)     cout << *it <<" ";
```

Результат

A B C D E F G H I J

v.end() возвращает итератор, указывающий не на последний элемент контейнера, а на непосредственно следующий за ним элемент. Это часто бывает удобно. Например, для любого контейнера *v* разность *(v.end() - v.begin())* всегда равна *v.size()* (табл. 6).

Таблица 6

Операции вставки и удаления для векторов

Операция	Описание
v.insert(pos,elem)	Вставляет в позицию итератора *pos* копию элемента *elem* и возвращает позицию нового элемента
v.insert(pos,n,elem)	Вставляет в позицию итератора *pos n* копий элемента *elem* (и не возвращает значения)
v.insert(pos,beg,end)	Вставляет копию всех элементов интервала *[beg,end)* в позицию итератора *pos* (и не возвращает значения)
v.push_back(elem)	Присоединяет копию *elem* в конец вектора
v.pop_back()	Удаляет последний элемент (не возвращая его)
v.erase(pos)	Удаляет элемент в позиции итератора *pos* и возвращает позицию следующего элемента
v.erase(beg,end)	Удаляет все элементы из интервала *[beg,end)* и возвращает позицию следующего элемента
v.clear()	Удаляет все элементы (контейнер остается пустым)

Пример 6

```
vector<char> v;
 for (int i=0; i<10; i++)  v.push_back('A' + i);
 // Отображаем исходное содержимое вектора.
 cout << "Размер = " << v.size() << endl;
 cout << "Исходное содержимое вектора:\n";
 for (int i=0; i<v.size(); i++)  cout << v[i] << " ";
 cout << endl << endl;
// Вводим итератор it и присваиваем ему адрес первого элемента
// в векторе
 vector<char>:: iterator it= v.begin();
 it += 2; // указатель на 3-й элемент вектора
 // Вставляем 10 символов 'X' в вектор v.
 v.insert(it, 10, 'X');

 // Отображаем содержимое вектора после вставки символов
 cout << "Размер вектора после вставки = "<< v.size() << endl;

 cout << "Содержимое вектора после вставки:\n";
 for (i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl << endl;

 // Удаление вставленных элементов.
it = v.begin();
 it += 2; // Указатель на 3-й элемент вектора
 v.erase(it, it+10); // Удаляем 10 элементов подряд.

 // Отображаем содержимое вектора после удаления символов.
 cout << "Размер вектора после удаления символов = " << v.size() << endl;
 cout << Содержимое вектора после удаления символов:\n";
 for (i=0; i<v.size(); i++) cout << v[i] << " ";
```

Результат

Размер = 10
Исходное содержание вектора:
A B C D E F G H I J
Размер вектора после вставки = 20
Содержимое вектора после вставки:
A B X X X X X X X X X C D E F G H I J
Размер вектора после удаления символов = 10
Содержимое вектора после удаления символов:
A B C D E F G H I J

МНОГОМЕРНЫЕ ВЕКТОРЫ

Статический массив в соответствии с синтаксисом языка есть массив массивов, т. е. массив, элементами которого служат массивы.

Аналогично многомерный вектор – это вектор векторов, т. е. вектор, элементами которого являются векторы.

Пример 7

```
/*Создание двумерного вектора (матрицы) целого типа при неизвестном количестве строк и столбцов*/
   vector<vector<int>> dvec;
   // Инициализация матрицы
   int N, M;
   cout<<"Введите число строк и столбцов";
   cin>>N>>M;
for(int i = 0; i < N; i++)
  {
     vector<int> vec;            // Это вектор-строка
     for(int j = 0; j < M; j++)
        vec.push_back(i);        // Заполняем очередную строку
     dvec.push_back(vec);        // Загружаем в матрицу очередную строку
  }
```

Если количество строк *N* и количество столбцов *M* матрицы нам известно, то для создания матрицы надо использовать конструкцию
int N, M;

.....

vector< vector<int> > matrix(N, vector<int>(M));

// Между двумя угловыми скобками обязательно должен быть пробел!

Инициализировать такую матрицу можно обычным способом:

```
for (int i=0; i<N; i++)
  for ( int j=0; j<M; j++)
    matrix[i][j]=rand()%5;
```

Таким образом, данная матрица состоит из *N* строк, а каждая строка – это вектор из *M* элементов, т. е. со строками матрицы можно работать как с обычными одномерными векторами, используя весь аппарат STL. Если вы будете вставлять в строку или удалять из нее элементы, то размер строки будет естественно меняться.

Значит, в отличие от статических массивов, в двумерном векторе (матрице) количество элементов в строках может быть разным.

Вектор может быть параметром функции и может возвращаться в виде результата работы функции.

Контейнеры *STL* всегда копируются при любых попытках передать их в качестве параметра.

Таким образом, если вы передаете вектор из миллиона элементов функции, описанной следующим образом:

void function(vector<int> v) { // Старайтесь никогда так не делать ...},

то весь миллион элементов будет скопирован в другой, временный, вектор, который будет освобожден при выходе из функции *function.* Если эта функция вызывается в цикле, о производительности программы можно забыть сразу.

Если вы не хотите, чтобы контейнер создавал временный вектор каждый раз при вызове функции, используйте передачу параметра по ссылке. Хорошим тоном считается использование при этом модификатора *const*, если функция не намерена изменять содержимое контейнера.

void function(const vector<int>& v) { // OK ...}

Если содержимое контейнера может измениться по ходу работы функции, то модификатор const писать не следует:

```
int (vector<int>& v) {    // Так держать
  v[0]++;  }
```

Двумерный вектор, так же как и одномерный, передается в функцию по ссылке.

Пример 8

```
.....
void function(vector<vector<int> > &dvec) // Формальный параметр –
{   for(int i = 0; i < 4; i++)                    // ссылка на двумерный вектор
    {  vector<int> vec;              // Это вектор-строка
      for(int j = 0; j < 5;j++)
         vec.push_back(i);  // Заполняем очередную строку
      dvec.push_back(vec); // Заносим очередную строку в матрицу
    }
}
 void main()
{  vector<vector<int>> dv;
   function(dv);
   for(int i = 0; i < dv.size(); i++)// dv.size() – число строк
    { for(int j = 0; j < dv[i].size(); j++)// dv[i].size() – число элементов
                                          // в i-й строке
         cout << dv[i][j] << " ";
        cout << "\n";
    }
```

Результат

```
0  0  0  0  0
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
```

АЛГОРИТМЫ STL

Алгоритмы STL – это специальные шаблонные функции, которые занимаются разного рода обработкой данных в указанных им интервалах. Их использование практически не отличается от использования обычных функций. Они позволяют искать, сортировать и изменять данные, т. е. алгоритмы в STL занимаются задачами, так или иначе связанными с обработкой однотипных данных, хранящихся в контейнерах. При этом алгоритмам передается не сам контейнер (в виде объекта), а диапазон обрабатываемых данных в виде двух итераторов: один указывает на начало данных *first*, которые должен обработать алгоритм, а второй – на конец *last.* При этом полагается, что последовательность указана на интервале [*first, last*), т. е. рассматриваются элементы, начиная с того, на который указывает *first*, и до элемента, предшествующего позиции *last.* Если в качестве *first* указать *begin*(), а в качестве *last – end()*, то будет рассматриваться все содержимое контейнера.

Чтобы работать с итераторами, надо их объявить. Например, следующий оператор создает итератор *it* для работы с векторами целых чисел:

vector<int>:: iterator it;

Некоторые из алгоритмов *STL* (применительно к векторам) приведены в табл. 7.

Чтобы эти алгоритмы работали, необходимо использовать директиву

#include <algorithm>

Таблица 7

Алгоритмы STL

Алгоритм	Назначение
1	2
copy(v.begin(), v.end(),v1.begin)	Копирование вектора *v* в вектор *v1*
count(v.begin(), v.end(),val)	Подсчет количества элементов со значением *val*
max_element(v.begin(), v.end())	Возвращает итератор, указывающий на элемент с наибольшим значением
min_element(v.begin(), v.end())	Возвращает итератор, указывающий на элемент с наименьшим значением

Окончание табл. 7

Алгоритм	Назначение
1	2
replace(v.begin(), v.end(),val,rval)	Замена *val* на *rval*
reverse(v.begin(), v.end())	Реверсирование вектора
sort(v.begin(), v.end())	Сортировка по возрастанию
sort(v.rbegin(), v.rend())	Сортировка по убыванию
**unique(v.begin(), v.end())*	Удаляет повторяющиеся смежные элементы из заданного диапазона
***generate(v.begin(), v.end(),gen)*	Заполняет вектор значениями, генерируемыми функцией *gen*. Это функция пользователя без аргументов
**** template<class ii, classT>* *T accumulate(v.begin(), v.end(),T nit)* ***** template< class ii, classT >* *T accumulate(v.begin(), v.end(),T init, multiplies<T>())* *ii -InputIterator*	Возвращает сумму элементов вектора плюс значение *init*. Если *init* = 0, то возвращает сумму значений элементов вектора Возвращает произведение элементов вектора и *unit*, если *init* = 1, то возвращает произведение значений элементов вектора

* *unique* не изменяет размера контейнера. Вместо этого каждый уникальный элемент помещается в очередную свободную позицию, начиная с первой. Дубликаты же элементов заносятся в конец контейнера. Это, так сказать, «отходы (остаток)» алгоритма. *Unique* возвращает итератор, указывающий на начало этого остатка. Как правило, этот итератор затем передается алгоритму *erase* для удаления ненужных элементов (пример 9).

** См. пример 10.

*** Для работы алгоритма *accumulate* необходима директива:
#include <numeric> (см. пример 11).

**** Для работы алгоритма *multiplies* необходима директива:
#include <functional > (см. пример 11).

Рассмотрим примеры использования некоторых из перечисленных алгоритмов.

Пример 9

```
#include <algorithm>
……….
int main()
{setlocale(LC_ALL,"rus_rus.1251");
// Вектор для хранения целых чисел c
  vector<int> v;
  vector<int>::iterator pos;// Итератор для вектора целых чисел
   cout<<"Исходный вектор"<<endl;
   for(int i=0;i<10;i++)
   {       v.push_back(rand()%7);// Занесение чисел в  вектор
           cout<<v[i]<<"  ";
   }
  // Поиск и вывод минимального  элемента
  pos = min_element (v.begin(),v.end());
  cout << "\nmin: " << *pos << endl;
  // Сортировка всех элементов по возрастающей
  sort (v.begin(), v.end());
  cout << "Отсортированные элементы:\n";
 for (pos = v.begin(); pos != v.end(); ++pos)    cout << *pos << ' ';
cout<<"\n Вектор без дубликатов "<<endl;
  //Убрать дубликаты
    v.erase(unique(v.begin(), v.end()),v.end());
 for(int i=0;i<v.size();i++)
   cout<<v[i]<<"  ";
return 0;
   }
```

Результат

Исходный вектор
6 1 6 5 3 2 5 0 5 6
min: 0

Отсортированные элементы:
0 1 2 3 5 5 5 6 6 6
Вектор без дубликатов
0 1 2 3 5

Пример 10

```
……
int rnd(void)
{return rand()%6 ;}
void main()
{   int N=20;
  vector<int >v(N);
  vector<int >::iterator it;
   generate(v.begin(),v.end(), rnd) ;
   for(it = v.begin(); it != v.end(); it++)
       cout << *it << " " ;
}
```

Результат

5 5 4 4 5 4 0 0 4 2 5 5 1 3 1 5 1 2 3 0

Пример 11

К коду примера 10 добавим директивы

```
#include <functional >
#include <numeric>
```

и операторы

```
it=v.begin();
cout<<"Сумма равна:  "<<accumulate(it+2,it+6,0);
cout<<"\n Произведение равно:
"<<accumulate(it+2,it+6,1,multiplies<int>());
```

Результат

Сумма равна: 17
Произведение равно: 320

ПОТОКОВЫЕ ИТЕРАТОРЫ

Потоковые итераторы позволяют интепретировать устройства ввода/вывода (потоки *cin/cout*) как итераторы. Это значит, что можно использовать устройства ввода/вывода в качестве параметров алгоритмов STL!

Основное назначение входных и выходных итераторов – как раз поддержка потоковых итераторов. С их помощью можно применять соотвествующие алгоритмы напрямую к потокам ввода/вывода.

Потоковые итераторы – это, на самом деле, объекты шаблонных классов разных типов ввода/вывода. Существует два потоковых итератора: *ostream_iterator* и *istream_iterator*. Наибольшее применение нашел первый из них, ему и уделим свое внимание.

Объект класса *ostream_iterator* может использоваться в качестве параметра любого алгоритма с выходным итератором. В примере 12 он используется как параметр метода *copy*.

Пример 12

```
int arr[] = {10,20,30,40};
vector<int>v(arr,arr+4);  //Переносим массив в вектор
ostream_iterator<int>it(cout,", ");// Создаем объект it класса
ostream_iterator
cout<<"Содержимое вектора"<<endl;
copy(v.begin(),v.end(),it); // Вывод вектора
```

Результат

Содержимое вектора
10, 20, 30, 40,

Анализ

Мы определили итератор *it* как объект класса *ostream_iterator* для чтения значений типа *int*. Конструктор этого объекта имеет два параметра. Первым является поток, в который будут записываться значения (в данном случае это *cout*); вторым – *C*-строка, которая будет выводиться после каждого значения. Здесь используем запятую и пробел.

Алгоритм *copy* копирует содержимое вектора в поток *cout* (в поток вывода на экран).

Если мы хотим содержимое вектора записать в файл, то вместо *cout* надо использовть выходной файловый поток класса *ostream* (работа 5).

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Составить шаблон функции печати сформированного вами вектора для любых типов его элементов. Размеры вектора в функцию не передавать!
2. Реализовать требуемый алгоритм задания методами контейнера *vector* и алгоритмами STL.
3. Если это не противоречит условию задачи, то заполнять вектор надо с помощью функций *rand* и *generate.*

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Сформировать и напечатать произвольную квадратную матрицу с целыми элементами, вводя с клавиатуры ее размер *N*. Определить сумму и произведение элементов каждой строки, исключая первый и последний элемент.

Вариант 2

Ввести и напечатать в обратном порядке последовательность целых чисел, количество которых заранее не известно. Сделать концом последовательности ввод нулевого значения. Определить, сколько еще чисел поместится в выделенной памяти без расширения контейнера. Найти и вывести на печать максимальный и минимальный элементы и поменять их местами.

Вариант 3

Сформировать и напечатать матрицу, вводя ее размеры (число строк и число столбцов) с клавиатуры. Определить максимальный элемент в каждой строке, а также элемент, отстоящий вправо от максимального на две позиции. Если при этом может произойти выход за пределы вектора, выдать соответствующее сообщение.

Вариант 4

Вычислить значения $y*\sin(y+\Delta)$ и записать их в массив *x*. Вычисления прекратить, если $|y*\sin(y+\Delta)|<10^{-2}$, y и Δ вводить с клавиатуры. Определить число незаполненных ячеек. Сжать вектор по размеру, т. е. незаполненные ячейки удалить. Отсортировать массив по убыванию.

Вариант 5

Вычислить в отдельной функции модуль и фазу комплексной переменной *z*, если $z=\dfrac{ae^{jk}}{b\cos(jk)+a\sin(jk)}$. Результат распечатать в главной функции, *a* меняется от $a_{\min}$ до $a_{\max}$ с шагом a_h, *b* и *k* не меняются. Все параметры задать с клавиатуры.

Вариант 6

Задано целое число $N \geq 60$. В отдельной функции сформировать массив чисел Фибоначчи, значение последнего из которых не превосходит *N*. Вывести массив на экран монитора с помощью второй функции. Поменять местами второй от начала и третий от конца элементы. «Сжать» массив, чтобы не осталось свободных ячеек.

Вариант 7

Ввести с клавиатуры целое число $N \geq 2$. Сформировать вектор произвольного размера из простых чисел, не превосходящих *N*. Определить число перераспределений памяти при заполнении вектора, а также сумму последних трех и произведение первых двух элементов вектора.

Вариант 8

Написать программу формирования массива *N* чисел арифметической прогрессии, первый член которой равен *A1*, а знаменатель прогрессии равен *Q*. Числа *N*, *A1* и *Q* вводить с клавиатуры компьютера. Определить, сколько *еще* элементов поместится в выделенной памяти. Реверсировать полученный массив и скопировать со второго по шестой его элементы в другой массив. Распечатать все полученные массивы.

Вариант 9

Создать шаблон функции формирования массива $M > 7$ чисел геометрической прогрессии, первый член которой равен *B1*, а знаменатель прогрессии равен *Q*. Числа *M, B1* и *Q* вводить с клавиатуры компьютера. В главной функции определить сумму его элементов с третьего по седьмой включительно.

Вариант 10

Написать программу формирования и печати массива *N* случайных целых чисел от *L* до *R* включительно. Значения *N, L* и *R* вводить с клавиатуры компьютера.

Определить количество свободных ячеек, которые могут быть заполнены без перераспределения памяти. Убрать из массива дубликаты.

Вариант 11

Создать и заполнить случайными числами два одномерных массива разных размеров. Вставить второй массив в первый после *k*-го элемента. В объединенном массиве найти максимальный и минимальный элементы и сумму элементов, расположенных между ними. Операторы цикла в коде программы не использовать!

Вариант 12

Создать и заполнить случайными числами два целочисленных массива. Преобразовать эти массивы таким образом, чтобы в них не было повторяющихся чисел. Найти, сколько членов первого массива совпадает с членами второго массива.

Вариант 13

Дан массив, состоящий из натуральных чисел (ввести с клавиатуры). Признаком окончания ввода служит ввод натурального числа *M*. Образовать новый массив, элементами которого будут элементы исходного массива, оканчивающиеся на цифру, записанную в переменной *k*. Найти произведение элементов нового массива, не используя оператора цикла.

Вариант 14

Создать и заполнить случайными числами целочисленный массив. Получить в порядке возрастания все целые числа, которые не совпадают с числами, расположенными в позициях между максимальным и минимальным значением.

Вариант 15

Сформировать прямоугольную матрицу, вводя ее размеры (число строк и число столбцов) с клавиатуры. Переставляя ее строки и столбцы, добиться того, чтобы наибольший элемент (или один из них) оказался в верхнем левом углу. Распечатать обе матрицы.

Вариант 16

Сформировать прямоугольную матрицу, вводя ее размеры (число строк и число столбцов) с клавиатуры. Найти строку с наибольшей и наименьшей суммой элементов. Вывести на печать найденные строки и суммы их элементов.

Вариант 17

Сформировать прямоугольную матрицу, вводя ее размеры (число строк и число столбцов) с клавиатуры. Определить количество особых элементов матрицы, считая элемент особым, если он больше суммы остальных элементов его столбца.

Вариант 18

Сформировать произвольную действительную квадратную матрицу, вводя с клавиатуры ее размер N. Найти наибольший по модулю элемент. Получить квадратную матрицу размером $N - 1$ путем отбрасывания в исходной матрице строки и столбца, на пересечении которых расположен элемент с найденным значением.

Создать перегруженные функции удаления строки и столбца в матрице с произвольным типом элементов.

Вариант 19

Сформировать и заполнить случайными числами прямоугольную матрицу размером $n \times m$ (n и m ввести с клавиатуры). Вычислить суммы элементов в столбцах до k-й строки. Полученные суммы вставить как строку перед k-й строкой. Создать перегруженную функцию вставки строки в матрицу с произвольным типом элементов.

Вариант 20

Добавить в двумерный динамический массив строку из одних нулей после каждой строки, сумма элементов которой больше заданного числа S.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите основные элементы *STL*.
2. В чем преимущества контейнера *vector* перед обычными динамическими массивами?
3. Что такое итератор?
4. Как объявить и инициализировать вектор?
5. Что возвращает функция *size()* и функция *capacity()*?
6. В чем разница в операциях *v[index]* и *v.at(index)?*
7. Как правильно передать одномерный вектор в функцию? Напишите прототип такой функции.
8. Как правильно передать двумерный вектор в функцию? Напишите прототип такой функции.
9. Как определить, сколько раз произошло перераспределение памяти при заполнении массива?
10. Как найти сумму и произведение элементов вектора, не используя операторов цикла?

3. ОБРАБОТКА СИМВОЛЬНЫХ ДАННЫХ

Необходимо ознакомиться с функциями класса *string* для создания строк, их модификации и поиска. Получить на их основе навыки обработки символьной информации, текстов.

При работе со строками часто возникают недорозумения. Как правило, это происходит из-за того, что термин «*строка*» может означать совершенно разные вещи – обычный символьный массив типа *char** и экземпляр класса *string*. Обычно термином «*строка»* обозначают объект строкового типа стандартной библиотеки *C++ (string)*. «Традиционные» же строки типа *char** называют *C-строками***.**

Строки типа *string* значительно удобнее для работы, чем *C*-строки. Это выражается в следующем.

1. Так как строка *string* – это последовательный контейнер с символами (*vector*<*char*>), то при изменении ее размеров происходит автоматическое выделение памяти, что исключает ошибки с распределением памяти, характерные для строк типа *char**.

2. Класс *string* содержит почти все методы, изложенные в работе 2 по отношению к векторам. Кроме того, в этом классе можно использовать множество других полезных методов, например *substr, find* и др.

3. Класс *string* позволяет работать со строками как с обычными типами, не создающими проблем для пользователей. Это означает, что строки можно копировать, присваивать и сравнивать как базовые типы (*int, float* и т. д.), не беспокоясь о возможной нехватке памяти или размере внутреннего блока, пердназначенного для хранения символов.

Строки типа *string* в отличие от строк *char** не требуют завершающего нулевого символа. К отдельным сиволом строки можно получить доступ операцией *[ind]* (см. табл. 4 в работе 2 и замечание к ней).

СОЗДАНИЕ И ИНИЦИАЛИЗАЦИЯ СТРОК

Приведенный ниже пример 1 демонстрирует способы инициализации строк.

Пример 1

```
string str1;// Пустая строка
string str2(«Люблю грозу в начале мая…»); // Инициализация через
                                          // конструктор
string str3 = «Разделяй и властвуй»;
string str4(str3); // Строка str4 инициализируется строкой str3
```

Это простейшие способы инициализации объектов *string*. Их разновидности обладают большей гибкостью и лучше поддаются контролю (пример 2).

Пример 2

```
string s1 ("Точность – вежливость королей.");
string s2 ("Хаос – это порядок, который нам не понятен.");
string s3("Идея выше факта.");
// Копирование 7 символов из середины источника
string s4(s2, 9, 7);
cout<<"s4: " << s4 << endl;
// Комбинированное копирование
string s5 =  " - " + s1.substr(11,10)+s1.substr(21,50);
cout<<"s5: "<< s5 << endl;
// Функция substr() также может копировать отдельные символы
string s6 = s3+s1.substr(0, 1);
cout<<"s6: "<< s6 << endl;
```

Результат

s4: порядок
s5: – вежливость королей.
s6: Идея выше факта.Т

Анализ

В первом аргументе функции *substr()* передается начальная позиция, а во втором – длина подстроки в символах. У обоих аргументов имеются значения по умолчанию. Функция *substr()* с пустым списком аргументов возвращает копию всего объекта *string*: это удобный способ копирования строк в C++.

ПРИСОЕДИНЕНИЕ, ВСТАВКА И КОНКАТЕНАЦИЯ (СЦЕПЛЕНИЕ) СТРОК

Функции ***append, insert*****,** операторы: **+, +=**

Одно из самых ценных и удобных свойств строк C++ (как любого вектора) состоит в том, что они автоматически растут по мере надобности, не требуя вмешательства со стороны программиста. Работа со строками не только становится более надежной, из нее почти полностью устраняются «*нетворческие*» операции: отслеживание границ памяти, в которой хранятся данные строки.

Пример 3

```
string str1("Hello, ");
string str2("world!");
cout <<"Исходные строки:\n";
cout << "str1: " << str1 << endl;
cout << "str2: " << str2 << endl ;
// Конкатенация(сцепление) двух string-объектов.
string str3;
str3 = str1 + str2;
cout<<"str3: " << str3 << "\n\n";
// Демонстрация использования функции append() и оператора += .
string s("Служить ");
string s1("бы ");
string s2(" прислуживаться  ");
s +=s1;          // Присоединение строки "бы "
cout<<"s: "<<s<<endl;
```

```
  s += "рад";      // Присоединение C-строки
  cout<<"s: "<<s<<endl;
  s += ',';              // Присоединение отдельного символа
  cout<<"s: "<<s<<endl;
  s.append(s2);      // Присоединение строки " прислуживаться  "
(эквивалент +=)
  cout<<"s: "<<s<<endl;
  s.append("тошно"); // Присоединение C-строки (эквивалент +=)
  cout<<"s: "<<s<<endl;
  s.append(3,'.');        // Присоединение 3 символов:...
  cout<<"s: "<<s<<endl;
// Демонстрация использования функции insert()  для вставки строки.
    string s4("Счастливые  наблюдают");
    cout <<"Исходная строка:\n";
    cout << "s4: " << s4 << endl;
    string s5("часов ");
    s4.insert(11,s5);  // Вставка строки "часов "
    cout<<"s4: "<<s4<<endl;
     s4.insert(17,"не"); //Вставка C-строки
     cout<<"s4: "<<s4<<endl;
```

Результат

```
Исходные строки:
str1: Hello,
str2: world!

str3: Hello, world!

s: Служить бы
s: Служить бы рад
s: Служить бы рад,
s: Служить бы рад, прислуживаться
s: Служить бы рад, прислуживаться тошно
s: Служить бы рад, прислуживаться тошно …
```

Исходная строка:
s4: Счастливые наблюдают.

s4: Счастливые часов наблюдают.
s4: Счастливые часов не наблюдают.

Анализ

Операторы + и += обеспечивают гибкие и удобные средства для объединения строковых данных. В правой части оператора может использоваться практически любой тип, интерпретируемый как один или несколько символов. Оператор + можно использовать для сцепления одного *string*-объекта с другим или *string*-объекта с *C*-строкой.

Оператор +=, как и функция *append*, может присоединять к исходной строке как *C*-строки, так и *string*-объекты. Но оператор += присоединет строковые значения, определяемые одним аргументом, функция же *append* может иметь несколько аргументов (табл. 1). Первый аргумент функции *insert* – это индекс, с которого вставляется новая *C*-строка или строка *string*. Функция *insert* широко используется для вставки символов (см. следующий раздел), в этом случае она имеет более двух аргуметов.

ВСТАВКА, ЗАМЕНА И УДАЛЕНИЕ СИМВОЛОВ В СТРОКАХ

Функции ***insert, replace, erase.***

Функция вставки символов ***insert*** очень удобна: вам не придется беспокоиться, чтобы вставляемые символы не вышли за пределы текущего блока памяти. Строка расширяется, и существующие символы вежливо подвигаются, уступая место новым.

Если вы хотите, чтобы существующие символы были заменены новыми, воспользуйтесь функцией перезаписи ***replace.*** Существует несколько перегруженных версий ***replace,*** но простейшая форма получает три аргумента:

- начальную позицию в строке;
- количество символов, заменяемых в исходной строке;
- строку замены (длина которой может не совпадать с вторым аргументом).

Строка замены может быть как типа *char*,* так и типа *string.*

Функция ***erase*** легко и эффективно удаляет символы из строк. Функция получает два аргумента: начальную позицию удаления (по умолчанию 0) и количество удаляемых символов (по умолчанию все). Если заданное количество символов больше количества оставшихся символов в строке, стираются все символы до конца (таким образом, вызов *erase* без аргументов удаляет из строки все символы).

Пример 4

```
string s1("Это простой тест.");
 string s2("ABCDEFG");
 cout <<"Исходные строки:\n";
 cout << "s1: " << s1 << endl;
 cout << "s2: " << s2 << "\n\n";
// Демонстрация использования функции insert для вставки символов
 cout << "Вставляем в строку s1 с 5 позиции 4 символа строки
s2,начиная с 0 позиции:\n";
 s1.insert(5, s2,0,4);
 cout <<s1 << "\n\n";
 // Демонстрация использования функции erase
 cout << "Удаляем 4 символов из строки s1, начиная с 5 позиции:\n";
 s1.erase(5,4);
 cout << s1<< "\n\n";
 // Демонстрация использования функции replace
 cout <<"Заменяем 2 символа в s1, начиная с 6 позиции, 3 символами
строки s2 с 1 позиции:\n";
 s1.replace(6, 2, s2,1,3);
 cout << s1 << endl;
```

Результат

Исходные строки:
s1: Это простой тест.
s2: ABCDEFG

Вставляем в строку *s1* с 5 позиции 4 символа строки *s2*, начиная с 0 позиции:

Это п*ABCD*ростой тест.

Удаляем 4 символа из строки *s1*, начиная с 5 позиции:

Это простой тест.

Заменяем 2 символа в *s1*, начиная с 6 позиции, 3 символами строки *s2* с 1 позиции:

Это пр*BCD*той тест.

Анализ

В данной программе в четвертый аргумент функции ***replace*** передается 3 символа строки *s2*, которые заменяют 2 символа строки *s1*, т. е. 2 символа заменили на 3 символа, а часть строки, оставшаяся после вставки, просто подвинулась!

ПОИСК В СТРОКАХ

Функции группы ***find*** класса *string* предназначены для поиска символа или группы символов в заданной строке. Приведем две наиболее применяемые функции этой группы.

find

Ищет в строке символ или группу символов. Возвращает начальную позицию первого найденного экземпляра или *npos* при отсутствии совпадений.

find_first_of

Ищет в строке и возвращает позицию первого символа, совпадающего с любым символом из заданной группы. При отсутствии совпадений возвращает *npos.*

Если передать в качестве параметра *find_first_of* один символ, вызов *find_first_of* будет полностью аналогичен вызову *find.* Однако если передать в качестве параметра *find_first_of* строку, метод вернет индекс первого вхождения любого символа из данной строки:

```
string s = "Hello, World!";
int index = s.find_first_of(' '); // поиск первого пробела
```

int index2 = s.find_first_of(", !"); // поиск первого пробела, знака восклицания или

//запятой

Особенно *find_first_of* удобно использовать для того, чтобы разделить строку на множество подстрок (слов) (см. пример 8).

СРАВНЕНИЕ СТРОК

Сравнение строк принципиально отличается от сравнения чисел. Для сравнения двух строк требуются лексические сравнения. Иначе говоря, когда вы проверяете символ и определяете, «*больше*» или «*меньше*» он, чем другой, вы в действительности сравниваете числовые представления этих символов в выбранной кодироовке. Чаще всего используется кодировка *ASCII*, в которой печатные символы английского языка представляются десятичными числами в интервале от 32 до 127, причем буквы в начале алфавита имеют меньшие *ASCII*-коды, чем буквы в конце алфавита. Это относится и к буквам русского алфавита.

В C++ предусмотрено несколько способов сравнения строк, каждый из которых обладает своими достоинствами и недостатками. Проще всего использовать перегруженные внешние (т. е. не класса *string*) операторы *operator*==, *operator*!=, *operator*>, *operator*<, *operator*>= и *operator*<=.

Пример 5

string s2("Что"), *s1*("Это");

// В левой части находится литерал в кавычках, в правой части – объект *string*

if("Что" == *s2*)

cout << "Есть совпадение" << *endl;*

// В левой части находится объект *string*, а в правой – указатель на строку в стиле C, завершенную нуль-терминатором.

if(s1 != s2.c_str())

cout << "Нет совпадения" << *endl;*

Результат

Есть совпадение
Нет совпадения

Анализ

Для повышения эффективности в классе *string* определены перегруженные операторы для прямых сравнений строковых объектов, литералов в кавычках и указателей на строки С;

Функция *c_str* возвращает *const char** – указатель на строку С, завершенную нуль-символом, которая эквивалентна текущему содержимому объекта *string!* Другими словами, преобразует строку *sting* в С-строку. Это позволяет для объектов *string* использовать стандартные функции С, например функцию *atoi* или любую функцию, определенную в заголовочных файлах *<string.h>, <stdlib.h>* или *ctype.h* (см. пример 7).

Функция *compare* класса *string* позволяет выполнять гораздо более изощренные и точные сравнения, чем набор внешних операторов. Она существует в нескольких перегруженных версиях для сравнения:

- двух полных строк;
- части одной строки с полной строкой;
- подмножеств двух строк.

Пример 6

```
string s(“abcd");
s.compare("abcd")      // Возвращает 0
s.compare("dcba")      // Возвращает значение <0 (s меньше)
s.compare("ab")        // Возвращает значение >0 (s больше)

s.compare(s)           // Возвращает 0 (s равно s)
s.compare(0,2,s,2,2)   // Возвращает значение <0 ("ab" меньше "cd")
s.compare(1,2,"bcx",2) // Возвращает 0 ("bc" равно "bc")
```

При обращении к отдельным символам строк обычно используется синтаксис индексирования ([]) в стиле массивов С. Строки С++ так же поддерживают другой вариант: функцию *at.* Если все проходит

нормально, эти два механизма индексации приводят к одинаковым результатам:

string s(“abcd");

cout<<s[1]; // Выводит *b*

cout<<s.at(1); // Выводит *b*

Однако все же между оператором [] и функцией *at* существует одно важное различие. При попытке обратиться к элементу по индексу, выходящему за границы массива, функция *at* выдает исключение, тогда как обычный синтаксис **[]** приводит к непредсказуемым последствиям.

В табл. 1 приведены наиболее распространенные функции для работы со строками.

Так как строки *string* являются обычными контейнерами STL, то они могут работать и с итераторами, так же как и рассмотренные ранее в работе 2 контейнеры *vector.*

Таблица 1

Функции работы со строками

Функция	Описание
*s.append(str, start, num)**	Добавляет в конец строки *s num* символов из строки *str*, начиная с индекса *start*
*s.append(С-строка, start, num)**	Добавляет в конец строки *s* первые *num* символов из С-строки, начиная с индекса *start*
*s.assign(str, start, num)**	Присваивает строке *s num* символов из строки *str*, начиная с индекса *start*
s.assign(С-строка, num)	Присваивает строке *s* первые *num* символов С-строки (строки с завершающим нулем)
*s.compare(start,num,str)**	Сравнивает со строкой *s num* символов строки *str*, начиная с индекса *start*
s.c_str()	Преобразует строку *s* в С-строку
*s.erase(start, num)**	Удаляет *num* символов из строки *s*, начиная с индекса *start*

Окончание табл. 1

Функция	Описание
*s.find(str,start)**	Ищет первое вхождение строки *str* в строку *s*, начиная с индекса *start*
*s.find(sim, start)**	Ищет первое вхождение символа *sim* в строку *s*, начиная с индекса *start*
*s.find_first_of(str,start)**	Ищет первое вхождение любого символа строки *str* в строку *s*, начиная с индекса *start*
s.insert(start,str)	Вставляет в строку *s*, начиная с индекса *start*, строку *str*
s.insert(start,str,start1, num)	Вставляет в строку *s*, начиная с индекса *start*, *num* символов строки *str*, начиная с индекса *start1*
s.length()	Возвращает длину строки
s.push_back(sim)	Добавляет символ *sim* в конец строки *s*
s.replace(start,num,str)	Заменяет *num* символов строки *s*, начиная с индекса *start*, строкой *str*
s.replace(start,orgNum,str, repStart, repNum,)	Заменяет *orgNum* символов в строке *s*, начиная с индекса *start*, *repNum* символами строки *str*, начиная с индекса *repStart*
*s.substr(start, num)**	Возвращает *num* символов строки *s*, начиная с индекса *start*
s.swap(str)	Меняет местами содержимое строк *s* и *str*

*По умолчанию *start* = 0, *num* – все символы строки *str*

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К РЕШЕНИЮ ЗАДАЧ

1. Для решения задач вариантов 12−15 необходимо с помощью функции **c_str** (табл. 1) преобразовать *string*-строку в С-строку и далее использовать функции проверки и преобразования символов языка С (табл. 2).

Таблица 2

Функции проверки символов языка *C*

Функция	Описание
isdigit(int c)	Возвращает значение не нуль, если *c* – цифра (0…9), и нуль в противном случае
isspace(int c)	Возвращает значение не нуль, если *c* – обобщенный пробел: символ пробела, символ табуляции, символ конца абзаца и другие, и нуль в противном случае
isalpha(int c)	Возвращает значение не нуль, если *c* – код буквы (*A…Z, a….z*), и нуль в противном случае
isalnum(int c)	Возвращает значение не нуль, если *c* – код буквы или цифры (*A…Z, a….z*, 0…9), и нуль в противном случае

Пример 7

Дана строка из цифр и букв латинского алфавита. Вывести число четных и нечетных цифр.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{       setlocale(LC_ALL,"rus_rus.1251");
      int i,n,chet=0,nechet=0,m;
      string s;
        cout<<"Введите строку, состоящую из цифр и букв латинского алфавита\n";
        cin>>s;
      // Количество символов в строке
         n =s.size();
       cout<<"Количество символов в строке:"<<n<<endl;
         for(i=0;i<n;i++)
```

```
        {
            m= s.c_str()[i]; // Преобразует строку s в С-строку
                             // и возвращает код символа С-строки
            if(m%2==0&&isdigit(m)!=0) chet++; // Если символ –
                                              // цифра с четным кодом
            if(m%2!=0&&isdigit(m)!=0) nechet++; // Если символ –
                                              // цифра с нечетным кодом
        }
      cout<<"\n Количество четных цифр:  "<<chet;
      cout<<"\n Количество нечетных цифр:  "<<nechet;
    system("pause");
      return 0;
}
```

Результат

Введите строку, состоящую из цифр и букв латинского алфавита
ytr568iu7777iuy135
Количество символов в строке: 18
Количество четных цифр: 2
Количество нечетных цифр: 8

Анализ

Если цифра четная, то и код ее четен. Например, код цифры 2 – 50, 4 – 52 и т. д.

2. При решении задач вариантов 2, 4, 5, 7, 10, 11, 17, 20 необходимо работать с отдельными словами исходной строки. Ниже приведен один из вариантов кода программы, которая позволяет разбивать исходную строку на подстроки (слова) согласно символам-разделителям. Причем эти символы могут находиться не только внутри строки, но и в ее начале и конце.

Пример 8

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

```
#include <string>
 using namespace std;
int main()
{    setlocale(LC_ALL,"rus_rus.1251");
   string s("  И/дым: Отечества, нам сладок и приятен!++");// Исходная
                                                              // строка
   string delim(": ,/!++ ");// Символы-разделители
   vector<int> WordBegPos,WordLength;//Массивы начала слов и их длин
  int CurBeg = 0;// Позиция начала строки на текущей итерации

        cout<<"Исходная строка:\""<<s<<"\"";
   while(1)
        { // Пропускаем все разделители перед началом текущего слова
              while(s.find_first_of(delim,CurBeg)==CurBeg)
                    ++CurBeg;
         // Помещаем в массив позцию начала текущего слова
           WordBegPos.push_back(CurBeg);

        //Позиция, следующая за последним символом текущего слова
           int CurFin = s.find_first_of(delim,CurBeg);
              if(CurFin == string::npos)// Условие выхода из внешнего
                                        // цикла while(1)
                {// Определяем длину последнего слова и заносим
                  // ее в массив
                       WordLength.push_back(s.size()-CurBeg);
                  break;
                }
              // Определяем длину текущего  слова и заносим
              // ее в массив
      WordLength.push_back(CurFin-CurBeg);
```

```
    CurBeg = CurFin+1;// Определяем позицию, с которой начнется
    //следующая итерация
        }
    //Нулевая длина последнего слова говорит о наличии разделителей
    //в конце  строки
    if(WordLength.back() == 0)
            { WordBegPos.pop_back(); WordLength.pop_back();// Удаляем
     //записи о них
            }

   cout<<endl<<"В исходной строке "<<WordBegPos.size()<<"
слов"<<endl;
   for(unsigned int i = 0; i<WordBegPos.size();++i)

{cout<<"\""<<s.substr(WordBegPos[i],WordLength[i])<<"\""<<endl;
            }
   system("pause");
   return 0;

}
```

Результат

Исходная строка: "И/дым: Отечества, нам сладок и приятен!++"
В исходной строке 7 слов:
"И"
"дым"
"отечества"
"нам"
"сладок"
"и"
"приятен"

Анализ

Функция *str.find_first_of(sub,pos),* начиная с позиции *pos*, ищет в строке *str* и возвращает позицию первого символа, совпадающего с любым символом из подстроки *sub*. Если значение *pos* не задано, то оно равно нулю, т. е. поиск будет вестись с начала строки.

Функция *str.substr(pos,length)* выделяет подстроку строки *str*, начиная с позиции *pos* длиной *length.*

Исходная строка содержит символы-разделители в начале, внутри и в конце строки.

В массивы *WordBegPos* и *WordLength* помещаются позиции, с которых начинаются слова в строке, и их длины.

Используя информацию, хранящуюся в этих массивах, с помощью функции *s.substr* выделяем отдельные слова из строки.

Если функция поиска не нашла требуемого вхождения, то она возвращает константу *npos* класса *string.* Чаще всего она равна −1, но не стоит полагаться на это, поскольку числовые значения подобных констант не определены стандартом языка, а значит, могут зависеть от конкретного компилятора.

3. При решении задач вариантов 3, 6 и 9 необходимо из отдельных слов исходной строки сформировать массив. В примере 9 функция *vtok* как раз и решает эту проблему.

Пример 9

```
#include <iostream>
#include <vector>
#include <string>
 using namespace std;
// Функция разбиения строки на слова
vector<string> vtok(const string &s, const string &delim)
{   vector<int> WordBegPos,WordLength;//Массивы начал слов и их длин

     vector<string> vec; // Массив для хранения слов
    int CurBeg = 0;// Позиция начала строки на текущей итерации

    while(1)
```

```
{ // Пропускаем все разделители перед началом текущего слова
        while(s.find_first_of(delim,CurBeg)==CurBeg)
            ++CurBeg;
  // Помещаем в массив позицию начала текущего слова
    WordBegPos.push_back(CurBeg);

//Позиция, следующая за последним символом текущего слова
    int CurFin = s.find_first_of(delim,CurBeg);
    if(CurFin == string::npos)// Условие выхода из внешнего цикла while(1)
        {// Определяем длину последнего слова и заносим
        // ее в массив
            WordLength.push_back(s.size()-CurBeg);
      break;
        }
      // Определяем длину текущего  слова и заносим
      // ее в массив
  WordLength.push_back(CurFin-CurBeg);
        CurBeg = CurFin+1;// Определем позицию, с которой
                          // начнется следующая итерация
}

// Нулевая длина последнего слова говорит о наличии
// разделителей в конце строки
if(WordLength.back() == 0)
    { WordBegPos.pop_back(); WordLength.pop_back();// Удаляем
//записи о них
    }

//Заносим слова в массив vec
```

```
        for(unsigned int i = 0; i<WordBegPos.size();++i)
            { vec.push_back(s.substr(WordBegPos[i],WordLength[i]));
            }

    return vec;
}
int main()
{
        setlocale(LC_ALL,"rus_rus.1251");

  string s1("И/дым: Отечества, нам сладок и приятен!++");// Исходная
                                                          // строка
  string s2(": ,/!++ ");// Символы разделители
  cout<<"Исходная строка:\""<<s1<<"\"";

   vector<string> v;
   v = vtok(s1,s2);// Вызов фукции
 cout<<endl<<"В исходной строке "<< v.size()<<" слов"<<endl;
  for(unsigned int i=0;i<v.size();i++)
  cout<<"\""<<v[i]<< "\""<<endl;
  system("pause");
  return 0;
}
```

Результат

Аналогичен примеру 8

Анализ

В этом примере функция *vtok( )* имеет два формальных параметра: ссылку на исходную строку *s* и ссылку на строку *delim*, содержащую символы-разделители. Возвращает эта функция вектор, элементами которого являются отдельные слова исходной строки.

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Удалить из строки все комментарии вида /*….*/ и //….

Вариант 2

Дана строка, в которой слова разделены одним пробелом. Напечатать все слова, отличные от слова *hello*.

Вариант 3

Дана строка, в которой слова разделены одним пробелом. Напечатать то слово, которое по алфавиту предшествует всем другим словам (все слова различны; слово *aaa* предшествует слову *aaaa*).

Вариант 4

Дана строка, в которой слова разделены одним пробелом. Напечатать все слова, содержащие ровно две буквы *d*.

Вариант 5

Дана строка, в которой слова разделены одним пробелом. Напечатать текст, составленный из последних символов всех слов строки.

Вариант 6

Дана строка, в которой слова разделены одним пробелом и более. Напечатать их в алфавитном порядке.

Вариант 7

Дана строка из строчных русских букв (не более 50). Напечатать измененную строку, где после каждого фрагмента «город» добавлен фрагмент текста «пригород». Фрагменты разделены пробелами.

Вариант 8

Дана строка, в которой слова разделены одним пробелом. Удалить из нее все повторяющиеся слова.

Вариант 9

Дана строка, в которой слова разделены одним пробелом. Напечатать слово максимальной длины, в котором нет символов цифр (считается, что такое слово только одно).

Вариант 10

Дана строка, в которой слова разделены одним пробелом. В каждом слове перенести первую букву в конец слова. Длина строки при этом не должна изменяться, и слова по-прежнему должны быть разделены пробелами.

Вариант 11

Дана строка, в которой слова разделены следующими символами: ,:/+. Если слово нечетной длины, то удалить его среднюю букву.

Вариант 12

Дана строка из цифр и букв латинского алфавита. Найти цифры, обладающие минимальным и максимальным значением.

Вариант 13

Дана строка из цифр и букв латинского алфавита. Вывести сумму всех цифр.

Вариант 14

Дана строка из цифр и букв латинского алфавита в произвольном порядке. Вывести номер позиции, которая содержит символ ‘5’, при условии, что этот символ окружен буквами.

Вариант 15

Дана строка из цифр и букв латинского алфавита в произвольном порядке. Распечатать те группы цифр, в которых цифра 7 встречается не более двух раз (группа цифр – это последовательность цифр, обрамленная буквами).

Вариант 16

Дана строка, в которой слова разделены одним пробелом и более. Удалить из нее все пробелы.

Вариант 17

Дана строка из фамилии студентов и их оценок, сведения разделены пробелом (например, Иванов345 Петров233 Сидоров454). Удалить из строки всех неуспевающих студентов (имеющих хотя бы одну двойку).

Вариант 18

Написать и протестировать функцию *DELETE(s1, s2)*, которая удаляет из строки *s1* все символы, встречающиеся в строке *s2*.

Вариант 19

Дан произвольный текст. Отредактировать текст так, чтобы предложения в тексте разделялись ровно двумя пробелами.

Вариант 20

Составить программу, которая реверсирует каждое слово строки *str*.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем разница между строками C++ и C-строками?
2. Перечислите и приведите примеры способов инициализации объектов *string*.
3. Что означают записи *s3.insert(2, s4, 3, 5) и s2.erase()*?
4. string s("от великого до смешного один шаг ");
   *int i= s.find("*ог*");*
   cout<<i; // ?

   *i=s.find_first_of("*ог*");*
   cout<<i;
   s.erase(15,8); cout << s; // ?
5. Как сравнить две строки?
6. Как можно определить длину строки?

4. СТРУКТУРЫ

Необходимо приобрести навыки создания новых типов данных на примере структур.

ОБЩИЕ СВЕДЕНИЯ

Структуры – это совокупность логически связанных переменных, возможно, различных типов, сгруппированных под одним именем для удобства дальнейшей обработки. Именно тем, что в них могут храниться данные разных типов, они и отличаются от массивов, хранящих данные одного типа.

Прежде всего необходимо создать шаблон структуры или, говорят, надо описать структуру (пример 1).

Пример 1

```
struct DATE
{ int day;                  // День
  int month;                // Месяц
  int year;                 // Год
  string day_name;     //  Название дня недели
  string month_name; //  Название месяца
};
```

Описание структуры начинается с ключевого слова *struct*, за которым следует необязательное имя (в данном случае *DATE*), которое называется ***именем типа*** структуры (иногда его называют тэгом или ярлыком структуры). Этот ярлык именует структуру и в дальнейшем может использоваться для сокращения подробного описания. Переменные, упоминаемые в описании, называются ***элементами*.** Следом

за правой фигурной скобкой, заканчивающей список элементов, может следовать список переменных, так же как и в случае базисных типов. Вот почему в приведенном выше описании структуры после закрывающей фигурной скобки стоит точка с запятой; она завершает пустой список. Описание *struct {....} p1, p2, p3;* синтаксически аналогично *int p1, p2, p3;* в том смысле, что каждый из операторов описывает *p1, p2, p3* как переменные соответствующего типа и приводит к выделению для них памяти. Описание же структуры без последующего списка переменных не выделяет никакой памяти. Оно только определяет форму структуры и действует как шаблон. Если такое описание снабжено ярлыком (именем типа), то его можно позже использовать при определении фактических экземпляров структуры. Например, используя указанное выше описание *DATE*, можно с помощью строки

DATE a1 ,a2;

объявить структурные переменные *a1, a2,* каждая из которых строится по шаблону, введенному структурой *DATE*. Любая переменная *a1, a2* содержит в строго определенном порядке элементы *day, mont, year, day_name*.

После создания структурной переменной надо ее инициализировать, например:

a1 = {17, 3, 1989, «пятница», «февраль»};

Язык С++ позволяет одновременно объявлять и инициализировать структурные переменные

DATE a1 = {17, 3, 1989, «пятница», «февраль»};

Для доступа к элементам структуры используют операцию: имя_структуры. имя_элемента_структуры. Например, *a1.day* или *a1.day_name[9]* (десятый элемент массива *day_name*).

МАССИВЫ СТРУКТУР

Отдельные структурные переменные с произвольным шаблоном, как и обычные переменные любого типа, могут быть объединены в массивы фиксированной длины:

```
struct BOOK
  {     string author;     // Автор книги
        string title ; // Название книги
```

```
    int jear ;              // Год издания
    int n;          // Количество странниц
} catalog[10];              // Массив структур
```

Имя *catalog* объявлено как массив 10 структур с общим шаблоном *BOOK*. Обращение к элементам отдельной структуры производится, как и ранее, с использованием операции − точки. Например, обращение вида

```
catalog[3].title[4];
```

задает пятый символ массива *title* в составе четвертого элемента массива структур *catalog.*

УКАЗАТЕЛИ НА СТРУКТУРЫ

Тип *struct* является совершенно полноправным типом данных языка С++. Поэтому можно объявить указатель данного типа. В отличие от массива имя структурной переменной не эквивалентно ее адресу. Адрес можно получить известным образом:

```
DATE data_1;    //  Объявление структурной переменной
DATE  *pt;      //  Создание указателя на структуры данного типа
pt = &data_1; // Занесение в указатель адреса конкретной структуры типа DATA
```

При наличии указателя на структуру к элементам этой структуры можно обращаться с помощью операции ->. Например, *pt-> year*. Можно обращаться к элементу структуры и таким образом: (*имя_указателя). имя_элемента_структуры. Наличие круглых скобок обязательно. Таким образом, следующие три выражения эквивалентны:

```
(*pt).year;   pt-> year;   data_1.year.
```

Все они именуют один и тот же элемент *int year* структурной переменной *data_1*, имеющий тип *DATE.*

СТРУКТУРЫ И ФУНКЦИИ

Структуры могут быть параметрами функции и возвращаться в качестве результата работы функции. Структуры могут быть переданы в функцию по значению, через указатель и через ссылочный параметр, как обычные переменные основных типов (см. работу 1).

При передаче по значению формальному параметру (структуре) передается копия фактического параметра (структуры). Эта копия может занимать «довольно много места», а процесс копирования – «много времени».

Пример 2

```
 // Передача структуры в функцию по значению
#include <iostream>
 struct man
       { string name;
         int dd,mm,yy;
        string  zodiak;
        };
 void proc1(man B)// Формальный параметр – структура B с шаблоном man
       {B.dd++;}
int main()
   {     man A = {"Иванов И.И",1,10,1989,"Весы"};
              proc1(A); //Фактический параметр – копия структуры A
                              // с шаблоном man
              cout<<"A.dd="<<A.dd; //Выдача на экран – A.dd=1
             system("pause");
            return 0;
   }
```

При передаче структуры в функцию по значению изменить элементы этой структуры в функции невозможно. Обратите внимание, что программа определяет структуру *man* вне функции *main* и до функции *proc1*, поскольку функция объявляет переменную *B* типа *man,* и определение структуры *man* и должно располагаться до функции.

Если вы хотите с помощью функции изменять элементы структуры, то необходимо передавать ее адрес через указатель (пример 3) или ссылочный параметр (пример 4) на структуру.

Пример 3

```
……….
// Передача структуры в функцию через указатель
  void proc2(man *p) // Формальный параметр – указатель на структуру
                     // типа man
          {p->dd++;}
        int main()
   {    man A={"Иванов И.И",1,10,1989,"Весы"};
          proc2(&A);// Фактический параметр – адрес структуры A
          cout<<"A.dd = "<<A.dd; //Выдача на экран – A.dd = 2
   …….
        }
```

Пример 4

```
……….
// Передача структуры в функцию через ссылочный параметр
  void proc2(man &B) // Формальный параметр – ссылочный параметр
          {B.dd++;}
        int main()
   {    man A={"Иванов И.И",1,10,1989,"Весы"};
          proc2(A);
          cout<<"A.dd = "<<A.dd; //Выдача на экран – A.dd = 2
   …….
        }
```

Структуры позволяют достаточно эффективно решить вопрос возврата множества значений из функции. Если функция должна вернуть несколько значений в качестве результата, то их следует поместить в структуру и возвращать значение указанного типа. Например, создать функцию, которая находит и возвращает в функцию *main* максимальный элемент матрицы, а также номер строки и столбца, на пересечении

которых расположен этот элемент. С помощью структуры такая задача решается следующим образом:

```
struct coord  // Описание структуры с именем coord
   { float pik;
     int n_str,n_stl;
   }
coord max_m(float *pt,int n,int m);// Возвращает структуру типа coord

int  main()
{  coord man1;// Структура man1 построена по шаблону coord
  float c[4][5]={.....};
  man1=max_m(c[0],4,5);// Элементам структуры man1 присваиваются
                       //значения элементов структуры man
  cout<<" Максимальное значение равно: "<<man1.pik;
  cout<<"\nНомер строки : "<< man1.n_str <<"\nНомер столбца: "<< man1.n_stl ;
  return 0;
 }
 coord max_m(float *pt,int n,int m)
  {       int i,j;
          coord man;// Структура man имеет  шаблон типа coord
          man.pik  =pt[0];
          for( i = 0;I < n;i++)
            for( j= 0; j < m;j++)
              if( pt[i*m + j] > man.pik)
                    { man.pik = pt[i*m + j];
                      man.n_str = i+1;
                      man.n_stl = j+1;
           }
          return man;   //Функция max_m возвращает структуру
  }
```

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

В экзаменационной ведомости для восьми студентов указаны ФИО, оценка, число и месяц проведения экзамена. Выдать на экран монитора фамилии всех студентов, сдавших сессию на 4 и 5, и всех студентов с неудовлетворительной оценкой.

Вариант 2

В расписании 6 строк, в каждой из которых описан предмет, преподаватель, номер группы, день недели, часы занятия, аудитория. Определить, сколько занятий ведет один и тот же преподаватель.

Вариант 3

В игре в «дурака» козырь – черви. Достоинство карт каждой масти (шесть, семь,…, туз) описать как структуру. Описать логическую функцию: бьет ли карта К1 карту К2 в комбинации (К1, К2)? Сообщение выдать на экран монитора.

Вариант 4

С помощью структуры описать данные на группу из семи детей, указав их имя, пол, рост. Определить имя самой высокой девочки в группе.

Вариант 5

Задан список из десяти строк. В каждой строке сведения: ФИО человека, его пол, день и год рождения. Выдать на экран монитора сведения о самом старшем мужчине.

Вариант 6

Задан список из десяти строк. В каждой строке сведения: ФИО человека, его пол, день и год рождения. Выдать на экран монитора все фамилии людей из группы, начинающиеся с буквы Л, и даты их рождения.

Вариант 7

В записной книжке указаны ФИО, адрес и номер телефона 12 знакомых. Определить, есть ли в записной книжке сведения о знакомом с фамилией Иванов и именем Игорь (если есть, то напечатать их).

Вариант 8

В записной книжке указаны ФИО, адрес и номер телефона 12 знакомых. Определить, есть ли в записной книжке сведения об абоненте 46-14-13 (если есть, то вывести на экран монитора ФИО и адрес).

Вариант 9

Задан список группы из 11 человек, где указаны ФИО. Определить самое распространенное мужское и женское имя в группе.

Вариант 10

Задан список группы из 11 человек, где указаны ФИО. Вывести на экран монитора их в алфавитном порядке.

Вариант 11

В группе 10 юношей и девушек. Заданы их имя, год рождения, пол и вес. Определить имя самого тяжелого юноши, а также то, насколько его вес больше среднего веса юношей в группе.

Вариант 12

В прайс-листе на компьютерные процессоры заданы: наименование, частота, фирма-производитель и цена. Определить частоту самого дешевого процессора *Core i3* фирмы *Intel.*

Вариант 13

Оформить в виде структуры год, месяц, число. Описать функцию, вычисляющую количество дней в текущем месяце (учитывать високосный год).

Вариант 14

Оформить в виде структуры год, месяц, число. Описать функцию, проверяющую правильность даты (чтобы не было 31 июня и т. п.).

Вариант 15

Картотека видеотеки организована в виде массива структур с полями: название фильма, стоимость, режиссер. Ввести информацию по видеотеке и вывести информацию о фильмах, которые расположены между фильмами с максимальной и минимальной стоимостью.

Вариант 16

Имеется информация по итогам экзаменов в институте, всего в списке *N* человек. По каждому из студентов имеются следующие сведения: фамилия, оценка по математике, оценка по информатике и оценка по физике. Ввести информацию об экзаменах и напечатать фамилии студентов, у которых балл по каждому из предметов выше среднего по этому предмету.

Вариант 17

В телефонной книге даны фамилия, имя, улица, телефон шести человек. Найти, на какой улице живет Иванов Андрей и его телефон.

Вариант 18

Задан список из шести строк. В каждой строке указаны фамилия, телефон, фирма. Вывести на экран монитора всех сотрудников фирмы Intel.

Вариант 19

С помощью структуры описать данные пяти детей, указав их имя, пол, рост и вес. Определить имена всех мальчиков ростом выше 1 м 50 см.

Вариант 20

В структурном типе описаны характеристики шести машин: марка, год выпуска, стоимость. Определить самую дорогую машину.

Вариант 21

Определить новый тип данных – структуру, хранящую координаты точки на плоскости *x* и *y*. Написать функцию, возвращающую 1, если один из углов треугольника, заданного тремя переданными ей в параметрах точками, тупой, и 0 в противном случае.

Вариант 22

Определить новый тип данных – структуру, хранящую координаты точки в пространстве *x*, *y* и *z*. Написать функцию, вычисляющую расстояние между двумя переданными ей в параметрах точками. Продемонстрировать работу.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назначение структуры.
2. Дайте определение структуры.
3. Форма записи структуры.
4. Как объявить структурную переменную?
5. Доступ к структурной переменной.
6. Присвойте значение структурной переменной с помощью указателя.
7. Как передать структурную переменную из функции в функцию?
8. Объявите массив структурных переменных и инициализируйте его.

5. ОРГАНИЗАЦИЯ РАБОТЫ С ФАЙЛАМИ

Необходимо закрепить навыки создания и организации работы текстовых файлов и файлов данных.

Файл – это совокупность данных, размещенных на диске. Файлы – это основной способ долговременного хранения информации и документов (если не считать баз данных).

Работа с файлами в С++ может производиться как в стиле С, так и в стиле С++. И в том и другом случае файл рассматривается как поток (*stream*), представляющий собой последовательность считываемых или записываемых байтов.

До сих пор вы имели дело с двумя стандартными потоками:

cout – выходной поток, связанный с экраном (поток вывода на экран);

cin – входной поток, связанный с клавиатурой.

Потоки же для работы с файлами создаются как объекты следующих классов:

ofstream – для вывода (записи) данных в файл;

ifstream – для ввода (чтения) данных из файла.

Чтобы использовать эти классы, необходима директива

#include <fstream>.

После этого в программе можно определять конкретные файловые потоки соответствующих типов (объекты классов *ofstream* и *ifstream*), например:

ofstream outfile; // Определяется выходной файловый поток с именем *outfile;*

ifstream infile; // Определяется входной файловый поток с именем *infile.*

В классах *ifstream, ofstream* определены конструкторы, позволяющие одновременно с определением файловых потоков создавать и открывать файлы. Например,

ifstream infile ("file1.txt");

создается входной файловый поток с именем *infile* для чтения данных из файла. Разыскивается файл с названием *file1.txt*. Если такой файл не существует, то конструктор завершает работу аварийно.

ofstream outfile ("file2.txt ");

Создается выходной файловый поток с именем *outfile* для записи информации в файл. Если файл с названием *file2.txt* не существует, он будет создан, открыт и соединен с потоком *outfile*. Если файл уже существует, то предыдущий вариант будет удален и пустой файл будет создаваться заново.

Мы с вами будем работать с файлами в стиле C++, так как он имеет ряд достоинств по сравнению со стилем С. Пожалуй, основным из них является возможность применения очень удобных операций «поместить в поток» (<<) и «взять из потока» (>>).

ВЫПОЛНЕНИЕ ОПЕРАЦИЙ ЧТЕНИЯ И ЗАПИСИ

Для начала вы должны объявить объект типа *ofstream,* указав имя требуемого выходного файла как символьную строку, что показано ниже:

ofstream outfile("test.txt");

Аналогично может создаваться входной поток, связанный с файлом:

ifstream infile("test.txt");

К созданным таким образом потокам можно применять операции «поместить в поток» (<<) и «взять из потока» (>>). Преимуществом этих операций, работающих с текстовыми файлами, по сравнению с аналогичными функциями языка С является простота использования и автоматическое распознавание типов данных.

Рассмотрим следующий код.

Пример 1

```
int i = 1, j = 25, i1, j1;
   double a = 25e6, a1;
   string  s("Сидоров"), s1;
   ofstream outfile("test.txt");// создание файла как выходного потока
…
```

```
outfile << i << ‘  ‘ << j << ‘  ‘ << a << ‘  ‘ << s << endl;
outfile.close();// закрытие файла
ifstream infile("test.txt"); // открытие файла как входного потока
…
infile >> i1 >> j1 >> a1 >> s1;
infile.close();// закрытие файла
```

Результат

1 25 2.5e+007 Сидоров

В этом коде создается файл «*test.txt*» и в него записываются в текстовом виде два целых числа *i* и *j*, действительное число *a* и строка *s,* содержащая одно слово, после чего манипулятором потока *endl* выполняется перевод строки. Причем запись всех этих данных осуществляется одним оператором, содержащим операции сцепления.

После того как файл закроется, в нем будет записан текст "1 25 2.5e+07 Сидоров". Дальнейшие операторы создают входной поток, связанный с этим файлом и одним оператором, содержащим сцепленные операции «взять из потока», который читает все эти данные.

Продолжим рассмотрение операции <<. Последовательное применение операций «поместить в поток» (сцепленных или задаваемых самостоятельными операторами) приводит к занесению текстов в одну строку, как в рассмотренном выше примере. Если требуется перейти на новую строку, то можно или ввести в текст символ конца строки '\n', или применить манипулятор потока *endl* (сокращение от *end line* – конец строки). Например, оператор

```
outfile << "2 * 2 :\n" << (2 * 2) <<endl;
```

даст следующий результат: первая строка будет содержать текст "2 * 2 :", вторая – "4", а курсор файла будет переведен на третью строку.

В предыдущих примерах выводились константы и константные выражения. При выводе переменных все работает точно так же. Например, операторы

```
int i = 25, j = 2;
outfile << i << " * " << j << " = " << (i * j) << endl;
```

выводят в файл текст: "25 * 2 = 50".

Помимо этой операции выводить данные в поток можно еще двумя способами: методом *put* и методом *write*. Метод *put* выводит в поток один символ. Например, оператор

outfile.put('Я');

выведет в поток символ 'Я'.

Метод *write* выводит в файл из символьного массива, на который указывает его первый параметр, число символов, указанных вторым параметром. Например, оператор

outfile.write(s,5);

записывает в поток *outfile* 5 символов из массива *s*. Причем эти символы никак не обрабатываются, а просто выводятся в качестве сырых байтов данных. Среди этих символов, например, может встретиться в любом месте нулевой символ, но он не будет рассматриваться как признак конца строки.

Аналогичный оператор *read* может затем записать эти символы в какой-то другой символьный массив и тоже без всякой обработки.

Теперь остановимся на операции «взять из потока» (>>). Эта операция извлекает данные из потока, заданного ее левым операндом, и заносит их в переменную, заданную правым операндом. Операция возвращает поток, указанный как ее левый операнд. Благодаря этому допускаются сцепленные операции «взять из потока». Например, оператор

infile >> i >> j;

прочтет, начиная с текущей позиции файла, связанного с потоком *infile*, два целых числа и запишет в переменные *i* и *j*. Если в текущей позиции файла первому из чисел предшествуют пробельные символы или разделители, то они будут пропущены. За окончание числа операция примет первый отличный от цифры символ, в частности пробельный. Поэтому если эти два числа были ранее записаны в файл, например, оператором

outfile << i << ' ' << j << endl;

то они прочтутся нормально. Но если они были записаны оператором

outfile << i << j << endl;

т. е. без пробела, то их цифры будут слиты вместе и это составное число прочтется как *i*, а при чтении *j* произойдет ошибка.

Операцией >> можно вводить из файла строки в переменные типа *string*

Например, операторы:

string s;

infile >> s;

осуществляют чтение из файла в строку *s*. Но при этом читается не вся строка, а только одна лексема – последовательность символов, заканчивающаяся пробельным или разделительным символом.

Это удобно, если надо производить анализ текста или искать в нем какое-то ключевое слово. Но это становится недостатком, если надо просто прочесть строку целиком (пример 2).

Пример 2

```
 ofstream outfile("pr3.txt"); // Создание файла как выходного потока
// Пишем в файл
  outfile << "Учимся программировать на языке С++, " << "Часть 2" << endl;
  outfile << "НГТУ" << endl;
  outfile << "РЭФ" << endl;
  outfile.close();
  string one, two,three;
  ifstream infile("pr3.txt"); //Открытие файла как входного потока
// Читаем из файла
  infile>>one;
  infile >> two;
  infile >> three;
// Выводим прочитанное на экран
  cout<<one<<endl;
  cout << two << endl;
  cout << three << endl;
  infile.close();
```

Результат

```
Учимся
программировать
на
```

Таким образом, вместо того чтобы прочесть три введенные в файл строки, мы не прочли целиком даже первую строку. Но этого и следовало ожидать, так как после выполнения оператора *infile>>one;* из первой строки будет считана последовательность символов до первого пробела, т. е. слово «Учимся». Эти символы будут помещены в строку *one*. Далее оператором *infile >> two;* в строку *two* будет считано второе слово первой строки – «программировать» и т. д.

Для того чтобы прочитать всю строку, надо применить функцию *getline*. Существует две версии этой функции:

первая – *getline(input,str,delim);*

вторая – *getline(input,str).*

Здесь

input – поток, из которого считываются данные,

str – строка, в которую считываются данные,

delim – символ-разделитель.

Функция *getline* считывает неформатированные данные из потока в строку. Останавливается, как только найден символ, равный разделителю, или исчерпан поток. Первая версия использует в качестве разделителя *delim,* вторая – *'\n'*. Символ-разделитель удаляется из потока и не помещается в строку.

Таким образом, чтобы прочесть все строки в примере 2, необходимо операторы

```
infile>>one;
infile >> two;
infile >> three;
```

заменить на операторы

```
getline(infile,one);
getline(infile,two);
getline(infile,three);
```

Если бы мы применили первую версию этой функции во всех трех операторах, взяв в качестве *delim* пробел – ‘ ‘, то получили бы тот же результат, что и в примере 2.

Если в качестве входного потока, из которого считываются данные, используется поток *cin,* то оператор *getline(cin,str)* позволяет вводить с клавиатуры строку *str*, где отдельные слова разделены пробелами или другими разделительными символами.

В классе *ifstream* имеется еще один метод чтения из потока – *get*.

Функция *get* вводит одиночный символ из указанного потока (даже если это символ-разделитель) и возвращает этот символ в качестве значения вызова функции. Этот вариант функции *get* возвращает *EOF*, когда в потоке встречается признак конца файла.

Следующий код использует функцию *get* , чтобы определить количество символов в файле.

Пример 3

```
ifstream infile("test.txt");
int i = 0;
char c;
while((c = infile.get()) != EOF)
{    i++; }
infile.close();// закрытие файла
```

Функцию *get* удобно использовать для поиска в файле какого-то ключевого символа. Например, если в приведенный код добавить оператор *if(c=='H')break,* то программа найдет и запомнит номер позиции, в которой первый раз встретился символ *H*.

Если требуется записать в файл вектор, то это можно сделать, например, следующим образом:

```
vector<double>v;
…
 ofstream outfile("имя_фала");
/*Определение итератора it как объекта класса ostream_iterator для чтения значений типа double. Конструктор этого объекта имеет два параметра: outfile – это поток, куда записываются данные, и пробел, который будет выводиться после каждого значения. */
 ostream_iterator< double >it(outfile," ");
 copy(v.begin(),v.end(),it);// Вывод вектора в файл
```

ОПРЕДЕЛЕНИЕ КОНЦА ФАЙЛА И ПРОВЕРКА КОРРЕКТНОСТИ ЧТЕНИЯ ИЗ ФАЙЛА

Обычной файловой операцией в программах является чтение содержимого файла, пока не встретится конец файла. Для проверки корректности чтения из файла нужно использовать функцию *fail*. Она воз-

вращает значение *false*, если чтение из файла данной единицы (символа, слова, строки) произведено успешно, и *true,* если имеет место какая-то ошибка либо достигнут конец файла. Используя цикл *while,* можно непрерывно читать содержимое файла до тех пор, пока не найдется конец файла или не встретится другая ошибка, как показано ниже:

while (!infile.fail()) { / Чтение из файла */ }*

Чаще всего, как показано в примерах ниже, функцию *fail* используют неявно:

while (infile) { / Чтение из файла */ }// infile – это !infile.fail()*

Часто для определения конца файла используют и функцию *eof.* Эта функция возвращает значение *false*, если конец файла еще не встретился, и *true*, если встретился конец файла.

В примере 4 программа построчно читает содержимое файла *test.txt* до тех пор, пока не достигнет конца файла или не встретит иной ошибки, в случае чего выдается соответствующее сообщение об ошибке.

Пример 4

ifstream infile("test.txt");

string line;

while (getline(infile, line)) // Пока функция получения строки возвращает *true* (успех)

    cout << line << endl; // Выводим эту строку на экран

if (!infile.eof()) // Если выход из цикла произошел не из-за конца файла…

    // …выводим соответствующее сообщение

    *runtime_error("*Некорректные данные в файле *test.txt");*

Функция *runtime_error* («Текст ошибки») инициирует ошибку во время выполнения программы с соответствующим текстом для пользователя в виде, как правило, всплывающего окна сообщений о критической ошибке. После закрытия такого окна программа досрочно прерывает свое дальнейшее выполнение.

Аналогично, в примере 5 программа читает содержимое файла по одному слову до тех пор, пока не встретится конец файла.

Пример 5

```
ifstream infile("test.txt");
string word;
while (infile >> word) // Пока чтение строки проходит успешно
        cout << word << endl;
if (!infile.eof())
        runtime_error("Некорректные данные в файле test.txt");
```

И, наконец, в примере 6 программа читает содержимое файла по одному символу за один раз, используя функцию *get* (как в примере 3) до тех пор, пока не встретит конец файла.

Пример 6

```
ifstream infile("test.txt");
char letter = infile.get();
while (infile) // infile – это !infile.fail()
{       cout << letter;
        letter = infile.get();
}
if (!infile.eof())
        runtime_error("Некорректные данные в файле test.txt");
```

ВЫПОЛНЕНИЕ ОПЕРАЦИЙ ЧТЕНИЯ И ЗАПИСИ МАССИВОВ И СТРУКТУР

Чтобы читать и писать массивы и структуры, можно использовать функции *read* и *write.* При использовании функций *read* и *write* вы должны указать буфер данных, в который они будут читаться или из которого будут записываться, а также длину буфера в байтах, как показано ниже:

```
infile.read(buffer,sizeof(buffer));

outfile.write(buffer, sizeof(buffer));
```

Например, в примере 7 функция *write используется* для вывода содержимого структуры *worker* в файл *file.txt*, а функция *read* – для чтения из файла информации о служащем и записи ее структуры в *worker1*

Пример 7

```
#include <iostream>
 #include <fstream>
 using namespace std;
   ume main()
   { setlocale(LC_ALL,"rus_rus.1251");
     struct St
       {  string name;
          int age;
          double salary;
        } worker = { "Иванов Владимир",33,25000.0 },worker1;

     ofstream outfile("file.txt") ;
     outfile.write((char *) &worker, sizeof(St));// Запись в файл
     outfile.close();

    ifstream  infile("file.txt");
     infile.read((char *) &worker1, sizeof(St)); //Чтение из файла
     cout << worker1.name << endl;
     cout << worker1.age << endl;
     cout << worker1.salary << endl;
     infile.close();
    return 0;
  }
```

Функции *write* и *read* обычно получает указатель на символьную строку. Символы *(char *)* представляют собой оператор приведения типов, который информирует компилятор о том, что вы передаете указатель на другой тип.

Если же вам требуется записать в файл массив, например, из пяти структур *worker*, то оператор записи в файл примет вид

*outfile.write((char *) &worker[0], sizeof(St)*5);*

Аналогичные изменения надо внести и в оператор чтения из файла.

УПРАВЛЕНИЕ ОТКРЫТИЕМ ФАЙЛА

В примерах, приведенных выше, все файловые операции ввода и вывода выполнялись с начала файла. Однако часто возникает задача добавления информации в конец уже существующего файла. Для открытия файла в *режиме добавления* вы должны указать второй параметр, как показано ниже:

ifstream outfile("filenamE.txt", ios::app);

В данном случае параметр *ios::app* как раз и указывает режим открытия файла.

ЧТО ВАМ НЕОБХОДИМО ЗНАТЬ

1. Заголовочный файл *fstream* определяет классы *ifstream* и *ofstream,* с помощью которых ваша программа может выполнять операции файлового ввода и вывода.
2. Для открытия файла на ввод или вывод вы должны объявить объект типа *ifstream* или *ofstream,* передавая конструктору этого объекта имя требуемого файла.
3. После того как ваша программа откроет файл для ввода или вывода, она может читать или писать данные, используя операторы извлечения (>>) и вставки (<<).
4. Ваши программы могут выполнять ввод или вывод символов в файл или из файла, используя функции *get* и *put.*
5. Ваши программы могут читать из файла целую строку, используя функцию *getline.*
6. Большинство программ читают содержимое файла до тех пор, пока не встретится конец файла. Ваши программы могут определить конец файла с помощью функции *fail* или *eof.*
7. Когда ваши программы выполняют файловые операции, они должны проверять состояние всех операций, чтобы убедиться, что операции выполнены успешно. Для проверки ошибок ваши программы могут использовать функцию *fail.*
8. Если вашим программам необходимо вводить или выводить такие данные, как структуры или массивы, они могут использовать методы *read* и *write.*
9. Если ваша программа завершила работу с файлом, его следует закрыть с помощью функции *close.*

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К РЕШЕНИЮ ЗАДАЧ

1. Для неискаженного ввода текста, набранного кириллицей, с клавиатуры в строку, из строки в файл и из файла вновь в строку необходимо в исходный код программы включить функцию *setlocale(LC_ALL, ".866").*

2. Если операцией «поместить в поток» (<<) вы записали текст в файл кириллицей, то, для того чтобы посмотреть его, необходимо открыть этот файл в редакторе *Word,* установив кодировку символов *MS-DOS.*

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Ввести с клавиатуры шесть строк, слова в которых разделены пробелами. Записать их в текстовый файл. Найти максимальную длину строки в файле и распечатать все строки файла, имеющие такую длину.

Вариант 2

Вычислить значение *i*, *x*, *y*, если $y = 2\sin(x/3)\cdot e^{x}$, x изменяется от $\pi/10$ до $\pi/2$ с шагом $\Delta x = \pi/10$, i – номер шага по x. Результаты занести в файл данных *rez.txt*, где количество строк равно *i.*

Вариант 3

В программе создать файл, каждая строка которого содержит название горной вершины и ее высоту. Используя структуру для описания понятия «вершина», получить название самой высокой вершины по данным файла.

Вариант 4

Записать в программе текстовый файл из пяти строк. Подсчитать количество строк, которые оканчиваются буквой *'s'*, и считать их из файла.

Вариант 5

Записать в программе значения y_1, y_2 в файл *f1.txt*, а значения x_1, x_2 – в файл *f2.txt*, если $y_1 = \arcsin(t)$, $y_2 = \arccos(t)$, $x_1 = \sqrt{|1 - t^2|}$, $x_2 = |t+1| - |t-1|$, t изменяется от –0,5 до 0,5 с шагом $\Delta t = 0{,}1$.

Вариант 6

Записать в программе файл *ank.txt*, каждая из семи строк которого содержит следующие данные: пол, имя, рост. Распечатать средний женский рост и имя самого высокого мужчины по данным файла. Использовать структуру.

Вариант 7

Написать программу, которая работает в одном из двух режимов. Если в текущем каталоге имеется файл *tabl.txt*, то распечатать построчно его содержимое. В противном случае создать файл с таким именем и записать туда таблицу умножения для чисел от 2 до 9.

Вариант 8

Записать в программе текстовый файл *test.txt* из шести строк. Подсчитать количество строк в нем, которые начинаются с буквы '*f*'. Выдать эти строки на экран монитора.

Вариант 9

Из текстового файла удалить все символы пробела. Новый файл не создавать.

Вариант 10

Написать программу записи в файл и чтения из файла элементов массива структур для регистрации автомашин с полями: марка машины, год выпуска, цвет, номер.

Вариант 11

Ввести с клавиатуры в файл *ah.txt* произвольное количество строк. Ограничителем ввода является слово *end*. Подсчитать в нем количество строк, которые начинаются и оканчиваются одной и той же буквой. Выдать эти строки на экран монитора.

Вариант 12

В уже имеющемся файле из трех строк (предварительно его записать на диск) находятся вещественные числа (по три в каждой строке). Определить количество элементов файла, величина которых меньше среднего арифметического всех элементов данного файла.

Вариант 13

Записать в файл *ntr.txt* значения $x_1 = -1 + 1/3\sqrt{-5x^2 + 30x}$, $x_2 = -1 + 3/4\sqrt{x^2 + 4x + 20}$, если x изменяется от 0,1 до 1 с шагом 0,1. Переписать строки 2, 4, 6 из этого файла в файл *at.txt*.

Вариант 14

Создать два файла, содержащих сведения об игроках хоккейных команд «Динамо» и «Спартак». Структура записей файлов: фамилия, имя игрока; число заброшенных шайб; число сделанных голевых передач. По данным, извлекаемым из этих файлов, создать новый файл, содержащий данные о шести самых результативных игроках обеих команд (заброшенная шайба – 2 очка, передача – 1 очко).

Вариант 15

Используя структуру с элементами шаблона: фамилия, имя, возраст, распечатать количество с именем «Elena». Данные взять из файла, предварительно создав его.

Вариант 16

Написать программу записи в файл *f1.txt* программы на языке С++. Переписать в файл *f2.txt* содержимое *f1.txt* без комментариев.

Вариант 17

Сформировать файл из пяти строк. Каждая строка состоит из отдельных слов, разделенных пробелами. Строки ввести с клавиатуры. Считать в программу две последние строки и распечатать их.

Вариант 18

Сформировать файл из пяти строк, в каждой из которых задать ФИО и пол студентов группы. Считать и вывести на экран монитора ФИО всех девочек. Структуры не использовать.

Вариант 19

Из имеющегося файла *a.txt*, состоящего из шести строк, переписать в файл *b.txt* третью и четвертую строку. Строки могут иметь пробелы.

Вариант 20

Записать в файл *t.txt* четыре строки, введенные с клавиатуры компьютера. Переписать их в файл *e.txt* по алфавиту. Использовать алгоритмы *STL.*

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Файловый поток определен как *ifstream flow;*. Для каких целей его можно использовать?
2. Файловый поток определен как *ofstream flow1 ("file.txt ");*. Файл *file.txt* существует. Будут ли вновь записываемые в файл данные добавляться к уже имеющимся в нем данным?
3. Записанная в файл *f1.txt* строка имеет вид

НГТУ, кафедра РПУ

Как она будет выглядеть на экране после выполнения следующего фрагмента программы:

```
ifstream infile("f1.txt");
  string str;
infile>>str;
cout<<str;
```

4. Как будет выглядеть на экране строка в вопросе 3, если она будет считываться из файла оператором

```
getline(infile, str);
```

5. Как ввести с клавиатуры строку с пробелами?
6. Как определить количество символов в файле?
7. Что возвращает функция *eof*, если встретился конец файла*?*
8. Как записать в файл массив из *n* структур?
9. Как считать из файла массив из *n* структур?

ОГЛАВЛЕНИЕ