Лекция 4
Одномерные и двумерные массивы. Символьные строки и функции работы с ними

Массивы

В языке C/C++ ***массив*** – сложный тип данных, представляющий собой упорядоченную конечную совокупность элементов одного типа.

Размер массива − число его элементов.

Каждый элемент массива определяется идентификатором массива и своим порядковым номером – *индексом*.

Индекс – целое число, по которому производится доступ к элементу массива. Индексов может быть несколько. В этом случае массив называют ***многомерным***, а количество индексов одного элемента массива является его ***размерностью***.

Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива. Например, *double a*[10]; – описание массива из 10 вещественных чисел.

Одномерные массивы

В программе одномерный массив объявляется следующим образом:

тип ID_массива [размер] = {список начальных значений};

тип – базовый тип элементов массива (целый, вещественный, символьный);

размер – количество элементов в массиве.

Список начальных значений используется при необходимости инициализировать данные при объявлении, он может отсутствовать.

При декларации массива можно использовать также атрибуты «*класс памяти*» и *const*.

Размер массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, поэтому размер массива задается только константой или константным выражением. Нельзя задавать массив переменного размера, для этого существует отдельный механизм – *динамическое выделение памяти*.

Пример объявления массива целого типа: int *a*[5];

Индексы массивов начинаются с 0, т.е. в массиве *a* первый элемент: *a*[0], второй – *a*[1], … пятый – *a*[4].

Обращение к элементу массива в программе осуществляется с помощью записи операции обращения по индексу [] (квадратные скобки), например:

a[0]=1;

```
a[i]++;
a[3]=a[i]+a[i+1];
```

Пример объявления массива целого типа с инициализацией начальных значений:

```
int a[5]={2, 4, 6, 8, 10};
```

Если в группе {…} список значений короче, то оставшимся элементам присваивается 0.

Внимание! В языке C/C++ с целью повышения быстродействия программы отсутствует механизм контроля выхода за границы индексов массивов. При необходимости такой механизм должен быть запрограммирован явно.

Многомерные массивы

Декларация многомерного массива имеет следующий формат:

***тип ID*[*размер*1][*размер*2]…[*размерN*] =**
{
{*список начальных значений*},
{*список начальных значений*},
…
};

Списки начальных значений – атрибут необязательный.

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке C/C++ размещаются в памяти компьютера построчно друг за другом (см. раздел «Адресная функция»).

Рассмотрим особенности работы с многомерными массивами на конкретном примере двухмерного массива.

Например, пусть приведена следующая декларация двухмерного массива:

int m[3][4];

Идентификатор двухмерного массива – это указатель на массив указателей (переменная типа указатель на указатель: *int* ***m*;).

Поэтому двухмерный массив *m*[3][4]; компилятор рассматривает как массив трех указателей, каждый из которых указывает на начало массива со значениями размером по четыре элемента каждый. В ОП данный массив будет расположен следующим образом:

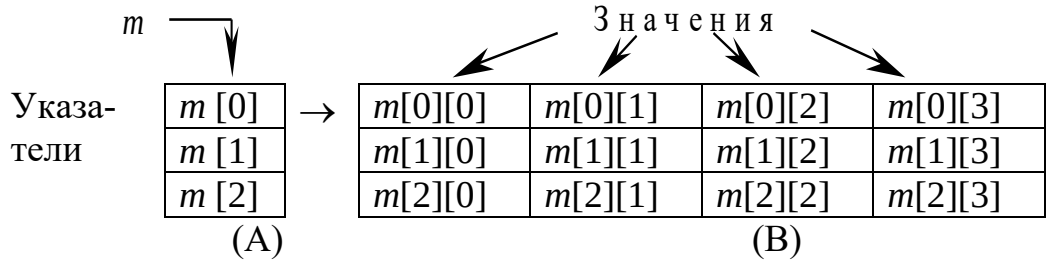


Рис.1. Схема размещения элементов массива *m* размером 3×4

Причем в данном случае указатель *m*[1] будет иметь адрес *m*[0]+4**sizeof*(*int*), т.е. каждый первый элемент следующей строки располагается за последним элементом предыдущей строки.

Приведем пример программы конструирования массива массивов:

```
#include <stdio>
void main()
{
int x0[4] = { 1, 2, 3,4};                  //  Декларация и инициализация
int x1[4] = {11,12,13,14};                 //  одномерных массивов
int x2[4] = {21,22,23,24};
int *m[3] = {x0, x1, x2,};                 // Создание массива указателей
int i,j;
for (i=0; i<3; i++) {
        printf("\n Строка %d) ", i+1);
        for (j=0; j<4; j++)
                printf("%3d", m[ i ] [ j ]);
        }
}
```

Результаты работы программы:

Строка 1) 1   2   3   4
Строка 2) 11 12 13 14
Строка 3) 21 22 23 24

Такие же результаты будут получены и в следующей программе:

```
#include <stdio>
void main()
{
int i, j;
int m[3][4] = { { 1, 2, 3, 4}, {11,12,13,14}, {21,22,23,24} };
for (i=0; i<3; i++) {
        printf("\n %2d)", i+1);
        for (j=0; j<4; j++)
                printf(" %3d",m[ i ] [ j ]);
        }
}
```

В последней программе массив указателей на соответствующие массивы элементов создается компилятором автоматически, т.е. данные массива располагаются в памяти последовательно по строкам, что является основанием для декларации массива *m* в виде

```
int m[3][4] = {1, 2, 3, 4, 11, 12, 13, 14, 21, 22, 23, 24};
```

Замена скобочного выражения *m*[3][4] на *m*[12] здесь не допускается, так как массив указателей не будет создан.

Таким образом, использование многомерных массивов в языке C/C++ связано с расходами памяти на создание массивов указателей.

Очевидна и схема размещения такого массива в памяти – последовательное (друг за другом) размещение «строк» – одномерных массивов со значениями (векторная организация памяти).

Обращению к элементам массива при помощи операции индексации $m[i][j]$ соответствует эквивалентное выражение, использующее адресную арифметику – $*(*(m+i)+j)$.

Адресная функция

Векторная память поддерживается почти всеми языками высокого уровня и предназначена для хранения массивов различной размерности и различных размеров. Каждому массиву выделяется непрерывный участок памяти указанного размера. При этом элементы, например, двухмерного массива X размерностью $n1 \times n2$ размещаются в ОП в следующей последовательности:

$X(0,0)$, $X(0,1)$, $X(0,2)$,... $X(0, n2–1)$, ..., $X(1,0)$, $X(1,1)$, $X(1,2)$,... $X(1, n2–1)$, ..., $X(n1–1,0)$, $X(n1–1,1)$, $X(n1–1,2)$, ..., $X(n1–1, n2–1)$.

Адресация элементов массива определяется некоторой адресной функцией, связывающей адрес и индексы элемента.

Пример адресной функции для массива X:

$$K(i, j) = n2*i + j;$$

где $i = 0,1,2,\ldots ,(n1–1)$; $j = 0,1,2,\ldots ,(n2–1)$; j – изменяется в первую очередь.

Адресная функция двухмерного массива $A(n,m)$ будет выглядеть так:

$$N1 = K(i, j) = m*i + j,$$

$i=0,1,\ldots, n–1$; $j=0,1,\ldots , m–1$.

Тогда справедливо следующее:

$$A(i, j) \leftrightarrow B(K(i, j)) = B(N1),$$

B – одномерный массив с размером $N1 = n*m$.

Например, для двухмерного массива $A(2,3)$ имеем:

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	– индексы массива A;
0	1	2	3	4	5	– индексы массива B.

Проведем расчеты:

$i = 0, j = 0$ $N1 = 3*0+0 = 0$ $B(0)$
$i = 0, j = 1$ $N1 = 3*0+1 = 1$ $B(1)$
$i = 0, j = 2$ $N1 = 3*0+2 = 2$ $B(2)$
$i = 1, j = 0$ $N1 = 3*1+0 = 3$ $B(3)$
$i = 1, j = 1$ $N1 = 3*1+1 = 4$ $B(4)$
$i = 1, j = 2$ $N1 = 3*1+2 = 5$ $B(5)$

Аналогично получаем адресную функцию для трехмерного массива $X(n1, n2, n3)$:

$$K(i, j, k) = n3*n2*i + n2*j + k ,$$

где $i = 0,1,2,\ldots ,(n1–1)$; $j = 0,1,2,\ldots ,(n2–1)$;); $k = 0,1,2,\ldots ,(n3–1)$; значение k – изменяется в первую очередь.

Для размещения такого массива потребуется участок ОП размером (*n*1**n*2**n*3)**sizeof*(*type*). Рассматривая такую область как одномерный массив *Y*(0,1,..., *n*1**n*2**n*3), можно установить соответствие между элементом трехмерного массива *X* и элементом одномерного массива *Y*:

$$X(i, j, k) \leftrightarrow Y(K(i, j, k)) .$$

Необходимость введения адресных функций возникает лишь в случаях, когда требуется изменить способ отображения с учетом особенностей конкретной задачи.

Передача одномерного массива в функцию:

```
void main(void)
{
int vect[20];
…
fun(vect);
…
 }
void   fun( int v[ ]) {
        …
}
```

При использовании в качестве параметра одномерного массива в функцию передается указатель на его первый элемент, т.е. массив всегда передается по адресу и параметр v[ ] преобразуется в *v. Поэтому этой особенностью можно воспользоваться сразу:

```
void   fun( int *v) {
        …
}
```

При этом информация о количестве элементов массива теряется, так как размер одномерного массива недоступен вызываемой функции. Данную особенность можно обойти несколькими способами. Например, передавать его размер через отдельный параметр. Если же размер массива является константой, можно указать ее и при описании формального параметра, и в качестве границы циклов при обработке массива внутри функции:

```
 void   fun( int v[20]) {
. . .
}
```

В случае передачи массива символов, т.е. строки, ее фактическую длину можно определить по положению признака окончания строки (нуль-символа) через стандартную функцию *strlen*.

Передача в функцию двухмерного массива:

Если размеры известны на этапе компиляции, то

```
void f1(int m[3][4])  {
int i, j;
for ( i = 0; i<3; i++)
```

```
        for ( j = 0; j<4; j++)
. . .                                   // Обработка массива
}
```

Двухмерный массив, как и одномерный, также передается как указатель, а указанные размеры используются просто для удобства записи. При этом первый размер массива не используется при поиске положения элемента массива в ОП, поэтому передать массив можно так:

```
void main(void)
{
        int mas [3][3]={{1,2,3}, {4,5,6}};
                …
        fun (mas);
                …
}
void fun( int m[ ][3]) {
…
}
```

Если же размеры двухмерного массива, например, вводятся с клавиатуры (неизвестны на этапе компиляции), то их значения следует передавать через дополнительные параметры, например:

```
        …
void fun( int**, int, int);
void main()
{
int **mas, n, m;
                ...
        fun (mas, n, m);
                …
}
void fun( int **m, int n, int m)  {
. . .                                   // Обработка массива
}
```

Пример создания одномерного динамического массива

В языке C/C++ размерность массива при объявлении должна задаваться константным выражением.

Если до выполнения программы неизвестно, сколько понадобится элементов массива, нужно использовать динамические массивы, т.е. при необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память).

Память под такие массивы выделяется с помощью функций ***malloc*** и ***calloc*** (операцией ***new*)** во время выполнения программы. Адрес начала массива хранится в переменной-указателе. Например:

```
                int n = 10;
```

```
double *b = (double *) malloc(n * sizeof (double));
```

В примере значение переменной *n* задано, но может быть получено и программным путем.

Обнуления памяти при ее выделении не происходит. Инициализировать динамический массив при декларации нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного – например *a*[3]. Можно обратиться к элементу массива и через косвенную адресацию – *(*a* + 3). В любом случае происходят те же действия, которые выполняются при обращении к элементу массива, декларированного обычным образом.

После работы захваченную под динамический массив память необходимо освободить, для нашего примера *free*(*b*);

Пример работы с динамическим массивом:

```
#include <alloc>
void main()
{
double *x;
int n;
printf("\nВведите размер массива – ");
scanf("%d", &n);
if ((x = (double*) calloc(n, sizeof(*x)))==NULL)  {        // Захват памяти
puts("Ошибка ");
        return;
}
     ...
// Работа с элементами массива
     ...
free(x);                                                  // Освобождение памяти
}
```

Пример создания двухмерного динамического массива

Напомним, что *ID* двухмерного массива – указатель на указатель. На рис. 10.1 приведена схема расположения элементов, причем в данном случае сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```
. . .
int  **m, n1, n2, i, j;
puts(" Введите размеры массива (строк, столбцов): ");
scanf(“%d%d”, &n1, &n2);          //вводим, например 3 и 4
m = (int**)calloc(n1, sizeof(int*)); // Захват памяти для указателей – A (n1=3)

for (i=0; i<n1; i++)                  // Захват памяти для элементов – B (n2=4)
        *(m+i) = (int*)calloc(n2, sizeof(int));
for ( i=0; i<n1; i++)
```

```
        for ( j=0; j<n2; j++)
                        m[i] [j] = i+j;                 //  *(*(m+i)+j) = i+j;
. . .
for(i=0; i<n; i++) free(m[i]);                          // Освобождение памяти
free(m);
. . .
```

Операции new и delete

В языке С++ для захвата и освобождения памяти используется более простой механизм – операции *new* и *delete*. Рассмотрим эти операции на простых примерах:

1) *type *p* = ***new*** *type* (*значение*); – захват участка памяти размером *sizeof*(*type*), путем установки на него указателя, и запись в эту область указанного значения;

```
. . .
```

delete *p*; – освобождение захваченной памяти.

2) *type *p* = ***new*** *type*[*n*]; – захват памяти на *n* последовательно размещенных объектов, возвращает указатель на начало участка ОП размером *n*sizeof*(*type*); используется для создания массива;

```
. . .
```

delete []*p*; – освобождение всей захваченной памяти.

Следует заметить, что операция *delete* не уничтожает значения, находящиеся по указанным адресам, а дает компилятору разрешение использовать ранее занятую память в дальнейшем.

Квадратные скобки в операции *delete* [ ] при освобождении памяти, занятой массивом, обязательны. Их отсутствие может привести к непредсказуемым результатам.

Пример создания одномерного динамического массива

Для примера приведем участок кода программы для одномерного динамического массива с использованием операций *new* и *delete*.

Напомним, что результатом операции *new* является адрес начала области памяти для размещения данных, указанного количества и типа. При нехватке памяти результат равен *NULL*.

```
…
double *x;
int   i, n;
puts(" Введите размер массива: ");
scanf(“%d”, &n);
x = new double [n] ;
if (x = = NULL) {
        puts(" Ошибка ! ");
return;
}
for (i=0; i<n; i++)                  // Ввод элементов массива
scanf(“%lf”, &x[i]);
```

```
…                                   // Обработка массива
delete [ ]x;                        // Освобождение памяти
…
```

Пример создания двухмерного динамического массива

Напомним, при создании двухмерного динамического массива сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```
. . .
int  **m, n1, n2, i, j;
puts(" Введите размеры массива (строк, столбцов): ");
scanf("%d%d", &n1, &n2);
m = new int*[n1];                   // Захват памяти для указателей – A (n1=3)
for (i=0; i<n1; i++)                // Захват памяти для элементов
        *(m+i) = new int[n2];
for ( i=0; i<n1; i++)
        for ( j=0; j<n2; j++)
                        m[i] [j] = i+j;             //  *(*(m+i)+j) = i+j;
. . .
for ( i=0; i<n1; i++)               // Освобождение памяти
        delete []m[i];
delete []m;
. . .
```

Примеры алгоритмов, использующих одномерные массивы.

а) Простейший пример

Задача 1. Ввести с клавиатуры массив, а затем вывести только его положительные элементы:

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

using namespace std;

int main()
{
int a[100],n,i;
M:  cout<<"Vvedite n<=100:";
cin>>n;
if(n>100) goto M;
cout<<"Vvedite massiv:\n";
for (i=0; i<n; i++)
  cin>>a[i];
```

```
cout<<"Polozhitelnye elementy:  ";
for (i=0; i<n; i++)
  if (a[i]>0) cout<<a[i]<<"  ";
cout<<endl;
getch();
return 0;
}
```

б) Нахождение суммы, произведения, количества

Задача 2. Найти сумму элементов массива.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
using namespace std;
int main()
{
int a[100],n,i,s;
M:  cout<<"Vvedite n<=100:";
cin>>n;
if(n>100) goto M;
cout<<"Vvedite massiv:\n";
for (i=0; i<n; i++)
   cin>>a[i];

s=0;
for (i=0; i<n; i++)
   s+=a[i];

cout<<"summa="<<s<<endl;
getch();
return 0;
}
```

Задача 3. Найти сумму положительных элементов массива.

От предыдущей задачи она отличается только добавлением оператора if:

```
s=0;
for (i=0; i<n; i++)
    if (a[i]>0)
       s+=a[i];
```

Задача 4. Найти произведение положительных элементов массива.
Содержательный участок программы имеет вид:

```
s=1;
for (i=0; i<n; i++)
   if (a[i]>0)
      s*=a[i];
```

Задача 5. Найти количество положительных элементов массива.

```
s=0;
for (i=0; i<n; i++)
   if (a[i]>0)
      s++;
```

Задача 6. Найти сумму четных (по значению) элементов массива.

```
s=0;
for (i=0; i<n; i++)
   if (a[i]%2==0)
      s+=a[i];
```

в) Нахождение порядковых номеров элементов

Задача 7. Найти сумму четных (по порядковому номеру) элементов массива.

Поскольку индексы элементов считаются с 0, а порядковые номера − с 1, четный порядковый номер означает нечетный индекс:

```
s=0;
for (i=0; i<n; i++)
   if (i%2==1)
      s+=a[i];
```

Задача 8. Найти индексы отрицательных элементов массива.

```
for (i=0; i<n; i++)
   if (a[i]<0)
      cout<<i<<"  ";
cout<<endl;
```

Задача 9. Найти индекс первого из отрицательных элементов массива.

```
for (i=0; i<n; i++)
   if (a[i]<0) {
      cout<<i<<endl;
            break;
   }
```

Задача 10. В массиве найти первый и последний нулевой элементы. Все элементы, лежащие между ними, увеличить на 1.

```
for (i=0; i<n; i++)
   if (a[i]==0) {
      cout<<i<<endl;
            break;
   }

for (k=n-1; k>=0; k--)
   if (a[k]==0) {
      cout<<k<<endl;
            break;
   }

for (p=i+1; p<k; p++)
   a[p]++;

for (i=0; i<n; i++)
  cout<<a[i]<<"  ";
cout<<endl;
```

г) Проверка условия, относящегося к массиву.

Задача 11. Проверить, все ли элементы массива меньше 5.

Для этого достаточно подсчитать **количество** элементов***, не меньших*** 5:

```
for (i=s=0; i<n; i++)
   if (a[i]>=5)
      s++;
if (s)
  cout<<"Ne vse elementy men\'she 5 !";
```

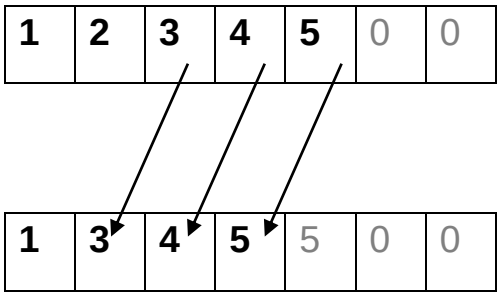
Заметим, что s++ можно заменить на s=1, т.к. конкретное количество элементов здесь несущественно.

Таким образом, для проверки условия, относящегося к массиву в целом, обычно нужна дополнительная переменная и цикл.

д) Удаление и вставка в массивах

Задача 11. Удалить из массива второй по счету элемент.

Поскольку полное количество элементов в массиве задано в его объявлении, физически "удалить" элемент из массива невозможно. Но с практической точки зрения, "удалить" элемент можно путем "сдвига" всех следующих за ним элементов "влево" (т.е. на предыдущую позицию, с меньшим на 1 индексом) и уменьшения на 1 количества *используемых* элементов:



Здесь используемые элементы выделены жирным цветом, а неиспользуемые - бледным.

Фрагмент программы имеет вид:

```
for (i=2; i<n; i++)
  a[i-1]=a[i];
n--;
```

Задача 12. Удалить из массива элементы, которые больше 10.

В этой задаче требуется удалить заранее неизвестное количество элементов; поэтому неясно, на сколько позиций нужно производить сдвиг. Потому лучше просто ввести две разные переменные: индекс элемента "до сдвига" и его индекс "после".

Сдвинуть нужно все *неудаляемые* элементы. Их количество и будет новым числом используемых элементов массива.

Индекс "до сдвига", очевидно, принимает все значения от 0 до n-1. А индекс "после" увеличивается на 1 после каждой записи очередного элемента:

```
for (k=i=0; i<n; i++)
  if (a[i]<=10)  a[k++]=a[i];
n=k;
```

Задача 13. Вставить в массив число 5 на вторую по счету позицию.

Подобно удалению, "вставка элемента" по сути означает "сдвиг" части элементов массива "вправо" (т.е. на последующие позиции, с б**о**льшими индексами) и помещение вставляемого числа на освободившуюся позицию. При этом число *используемых* элементов возрастает.

При программировании этого алгоритма нужно быть осторожным, чтобы не испортить последующие элементы массива раньше, чем они будут сдвинуты. Например, следующая программа:

```
for (i=0; i<n; i++)
  a[i+1]=a[i];
n++;
```

заполнит *все n элементов* массива одним и тем же значением, равным первоначальному значению a[0]. Действительно, при первой итерации цикла i=0, и будет выполнено a[1]=a[0]. При второй итерации будет a[2]=a[1] (а значение a[1] уже равно a[0]) и т.д.

Одним из выходов является использование второго массива:

```
b[0]=a[0];
b[1]=5;
for (i=1; i<n; i++)
  b[i+1]=a[i];
n++;
for (i=0; i<n; i++)
  a[i]=b[i];
```

Другой способ - изменить порядок сдвига:

```
for(i=n; i>1; i--)
  a[i]=a[i-1];
a[1]=5;
n++;
```

е) Обмен местами

Задача 14. Поменять местами первый и последний элемент массива.

При обмене, чтобы не потерять одно из значений, потребуется дополнительная переменная:

```
p=a[0];
a[0]=a[n-1];
a[n-1]=p;
```

ж) Поиск минимума/максимума

Задача 15. Найти в массиве наименьший элемент и его позицию.

В подобных задачах нахождения "самого-" в каком-нибудь смысле элемента используется та же идея, что и в спорте при определении рекорда: каждый очередной результат сравнивается с текущим рекордом, и если он "лучше", то он и становится новым значением рекорда, а иначе рекорд не меняется. За начальное значение рекорда принимается первый результат.

```
min=a[0]; k=0;
for (i=0; i<n; i++)
  if (a[i]<min)
    { min=a[i]; k=i; }
```

Эту же программу можно переписать и короче:

```
for (i=k=0, min=a[0]; i<n; i++)
  if (a[i]<min)
    min=a[k=i];
```

з) Сортировка массива

Задача 16. Отсортировать массив по возрастанию (т.е. расположить его элементы в порядке возрастания).

Для этой задачи придумано множество различных алгоритмов. Один из них − *сортировка методом прямого выбора:*

```
for(i=0; i<n; i++)
  for(k=i+1; k<n; k++)
    if (a[k]<a[i]) {
      p=a[k];          // Обмен a[k] и a[i]
      a[k]=a[i];
      a[i]=p;
    }
```

Здесь на каждой итерации внешнего цикла происходит помещение на i-ую позицию наименьшего из "оставшихся" (т.е. расположенных от i-ой позиции до конца массива) значений элементов.

Этот алгоритм включает в себя, таким образом, алгоритмы поиска минимума и обмена.

и) Поиск совпадений

Задача 17. Найти в массиве элемент, повторяющийся наибольшее количество раз. (Если таких элементов несколько, вывести любой из них).

```
for(max=i=0; i<n; i++){
  s=1;   // s - число повторений i-го элемента
  for(k=i+1; k<n; k++)
    if (a[k]==a[i]) s++;
  if (s>max) {
    max=s;       // max - число повторений x
    x=a[i];      // x - значение наиболее
  }              // частого элемента
}
```

Заметим, что x здесь не присвоено начальное значение, т.к. оператор x=a[i]; обязательно выполнится хотя бы раз (max вначале = 0, а s не меньше 1), а использовано x будет лишь впоследствии.

Примеры алгоритмов, использующих двумерные массивы.

а) Простейшие примеры

Задача 1. Ввести матрицу и увеличить все ее элементы на единицу.

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

using namespace std;

int main(){
int a[10][10],n,m,i,j;

cout<<"Vvedite n,m <=10:";  // Ввод матрицы
cin>>n>>m;
cout<<"Vvedite massiv:\n";
for (i=0; i<n; i++)
 for (j=0; j<m; j++)
   cin>>a[i][j];
                        // Увеличение на 1
for (i=0; i<n; i++)
 for (j=0; j<m; j++)
  a[i][j]++;
                        // Вывод матрицы
puts("Result:");
for (i=0; i<n; i++)
 for (j=0; j<m; j++)
  printf("%3d%c", a[i][j], j==m-1? '\n' : ' ');

getch();
return 0;
}
```

Задача 2. Найти в матрице наибольший элемент и его позицию.

```
max=a[0][0];
im=jm=0;
for (i=0; i<n; i++)
 for (j=0; j<m; j++)
  if (a[i][j]>max) {
      max=a[i][j];
      im=i;
```

```
      jm=j;
    }
printf("Max element a[%d][%d]=%d\n", im, jm, max);
```

Задача 3. Переписать матрицу в одномерный массив.

```
int b[100];
for (i=k=0; i<n; i++)
  for (j=0; j<m; j++)
    b[k++]=a[i][j];
```

После выполнения этого участка программы k=n*m -количество элементов в полученном массиве.

Приложение

Основные функции стандартной библиотеки string.h

Функция	Описание
char *strcat(char *s1, char *s2)	присоединяет s2 к s1, возвращает s1
char *strncat(char *s1, char *s2, int n)	присоединяет не более n символов s2 к s1, завершает строку символом '\0', возвращает s1
char *strcpy(char *s1, char *s2)	копирует строку s2 в строку s1, включая '\0', возвращает s1
char *strncpy(char *s1, char *s2, int n)	копирует не более n символов строки s2 в строку s1, возвращает s1;
int strcmp(char *s1, char *s2)	сравнивает s1 и s2, возвращает значение 0, если строки эквивалентны
int strncmp(char *s1, char *s2, int n)	сравнивает не более n символов строк s1 и s2, возвращает значение 0, если начальные n символов строк эквивалентны
int strlen(char *s)	возвращает количество символов в строке s
char *strset(char *s, char c)	заполняет строку s символами, код которых равен значению c, возвращает указатель на строку s
char *strnset(char *s, char c, int n)	заменяет первые n символов строки s символами, код которых равен c, возвращает указатель на строку s

Основные функции стандартной библиотеки ctype.h

Функции проверки категории символа	
isalnum	Проверяет, является ли аргумент буквой или цифрой
isalpha	Проверяет, является ли аргумент буквой
iscntrl	Проверяет, является ли аргумент управляющим символом
isdigit	Проверяет, является ли аргумент цифрой
isgraph	Проверяет, является ли аргумент символом, имеющим графическое представление
islower	Проверяет, является ли аргумент буквой в нижнем регистре
isprint	Проверяет, является ли аргумент символом, который может быть

	напечатан
ispunct	Проверяет, является ли аргумент символом, имеющим графическое представление, но не являющимся при этом буквой или цифрой
isspace	Проверяет, является ли аргумент разделительным символом
isupper	Проверяет, является ли аргумент буквой в верхнем регистре
isxdigit	Проверяет, является ли аргумент цифрой шестнадцатеричной системы счисления
Функции изменения регистра	
tolower	Изменить прописную букву на строчную («большую» на «маленькую»)
toupper	Изменить строчную букву на прописную («маленькую» на «большую»)