Л Е К Ц И Я 7

МАССИВЫ

Если с группой величин одинакового типа требуется выполнять однообразные действия, им дают одно имя, а различают по порядковому номеру. Это позволяет компактно записывать множество операций с помощью циклов. *Конечная именованная последовательность однотипных величин называется массивом*.

Отдельная единица таких данных, входящих в массив, называется *элементом массива*. В качестве элементов массива могут выступать данные любого типа, а также указатели на однотипные данные. Массивы бывают *одномерными* и *многомерными*.

ОДНОМЕРНЫЕ МАССИВЫ

Объявление массива

Поскольку все элементы массива имеют один тип, они также обладают одинаковым размером. Использованию массива в программе предшествует его объявление, резервирующее под массив определенное количество памяти. При этом указывается *тип* элементов массива, *имя* массива и его *размер*. Размер сообщает компилятору, какое количество элементов будет размещено в массиве. Синтаксис объявления массива имеет вид:

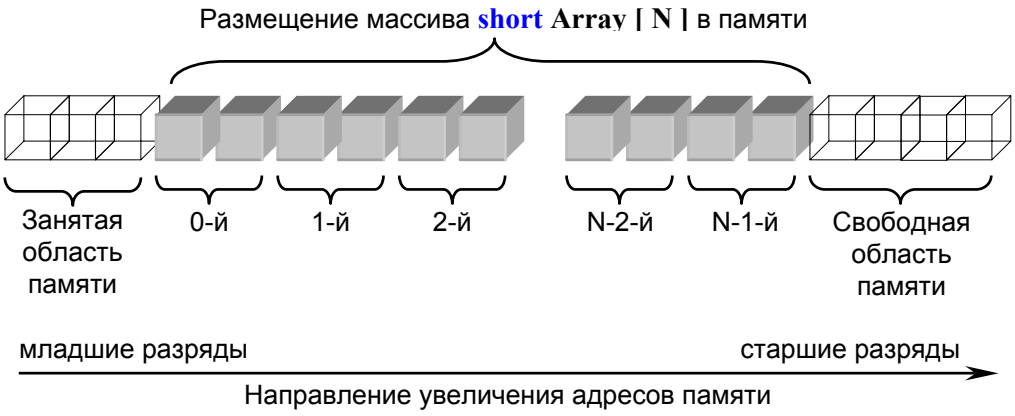
тип имя [[размер]] = [{ выражение0 [, выражение1 [, выражение2 …]] }] ;

Например, объявление

```
short Array [ 20 ] ;
```

зарезервирует в памяти место для размещения двадцати целочисленных элементов.

Элементы массива в памяти располагаются непосредственно один за другим. На рисунке показано расположение одномерного массива двухбайтных элементов (типа **short**) в памяти.



Инициализация массива

Инициализацию массивов, содержащих элементы базовых типов, можно производить при их объявлении. При этом непосредственно после объявления необходимо за знаком равенства (=) перечислить значения элементов в фигурных скобках через запятую (**,**) по порядку их следования в массиве.

Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются и имеет место частичная инициализация. В этом случае иногда после последнего значения в инициализирующем выражении для наглядности ставят запятую:

```
int b [ 5 ] = { 3 , 2 , 1 , } ;        // b [ 0 ] = 3 ,  b [ 1 ] = 2 ,  b [ 2 ] = 1 ,  b [ 3 ] = 0 ,  b [ 4 ] = 0
```

Размер массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется па этапе компиляции, поэтому размер может быть задан только целой положительной *константой или константным выражением*. Если при описании массива размер не указан, должен присутствовать инициализатор, в этом случае компилятор выделит память по количеству инициализирующих значений.

```
int E [ ] = { 0 , 2 , 4 , 6 } ;        // выделение в памяти места для хранения четырёх объектов типа int
```

Если далее в программе потребуется определить, сколько элементов имеется в массиве, можно воспользоваться следующим выражением:

```
int Size = sizeof ( E ) / sizeof ( E [ 0 ] ) ;
```

Здесь выражение **sizeof (E)** определяет общий размер, занимаемый массивом **E** в памяти в байтах, а выражение **sizeof (E [0])** возвращает размер в байтах одного элемента массива.

Обращение к элементам массива с помощью индекса

Обращение к элементам массива может осуществляться одним из двух способов:

- по номеру элемента в массиве (через его индекс);
- по указателю.

Для доступа к элементу массива после его имени указывается номер элемента (*индекс*) в квадратных скобках. Следует помнить, что в C++ *элементы массива нумеруются, начиная с 0*. Первый элемент массива имеет индекс 0, второй – индекс 1 и т.д. В следующем примере подсчитывается сумма элементов массива.

```
#include <iostream.h>
void main ( )
{
   const int n = 10 ;
   int m [ n ] = { 3 , 4 , 5 , 4 , 4 } ;
   for ( int i = 0 , sum = 0 ; i < n ; i++ ) sum += m [ i ] ;
   cout << ”Summa of elements:\t” << sum ;
}
```

Размер массивов предпочтительнее задавать с помощью именованных констант, как это сделано в примере, поскольку при таком стиле программирования для ее изменения достаточно скорректировать значение константы всего лишь в одном месте программы. Обратите внимание, что последний элемент массива имеет номер, на единицу меньший размера, заданного при его описании. При обращении к элементам массива автоматический *контроль выхода индекса за границу массива не производится*, что может привести к ошибкам.

Обращение к элементам массива с помощью указателя

Доступ к элементам массива через указатели заключается в следующем. Имя объявляемого массива ассоциируется компилятором с адресом его самого первого элемента с индексом 0. Таким образом можно присвоить указателю адрес нулевого элемента, используя имя массива:

```
char A [ ] = { ’w’ , ’o ’ , ’r ’ , ’l’ , ’d ’ } ;        // объявляет и инициализирует массив символов A
char* pA = A ;                                            // pA указывает на A [ 0 ]
```

Разыменовывая указатель **pA**, можно получить доступ к содержимому **A [0]**:

```
char Letter = *pA ;                                       // объявляет и инициализирует символьную переменную
cout << Letter << ’\n’ ;                                  // выводит на экран          w
```

Поскольку в C++ указатели и массивы тесно взаимосвязаны, увеличивая или уменьшая значение указателя на массив, программист получает возможность доступа ко всем элементам массива путем соответствующей модификации указателя:

```
pA += 2 ;                                                 // увеличивает адрес на 2 байта
```

```
cout << *pA << '\n' ;                    // выводит на экран        r
```

К этому же элементу можно обратиться иным способом:

```
Letter = *( A + 2 ) ;                    // эквивалент Letter = A [ 2 ] ;
```

Присвоение значений одного массива значениям другого массива вида **A [ ] = B [ ]** или **A = B** недопустимо, так как компилятор не может самостоятельно скопировать все значения одного массива в значения другого. Для этого программисту необходимо предпринимать определенные действия. Для этой цели чаще всего используется циклическое присвоение.

Объявление вида:

```
char ( *pA ) [ 10 ] ;
```

определяет указатель **pA** на массив из 10 символов. Если же опустить круглые скобки, то компилятор поймет запись

```
char *pA [ 10 ] ;
```

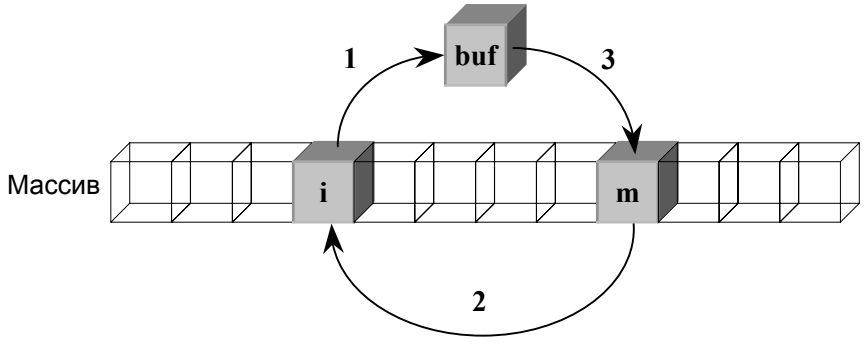
как объявление массива из 10 указателей на тип **char**.

Сортировка элементов одномерного массива

Сортировка целочисленного массива методом выбора. Алгоритм состоит в том, что выбирается наименьший элемент массива и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом, и так далее **n-1** раз. При последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива.

```
#include <iostream.h>          // подключает библиотеку функций ввода-вывода
#include <stdlib.h>            // подключает библиотеку, которая содержит функции rand ( ) и srand ( )
#include <time.h>              // подключает библиотеку, которая содержит функцию time ( )
void main ( )
{
   const int n = 10 ;                                  // количество элементов массива
   int B[ n ] ;                                        // объявляет массив целых чисел
   srand ( time ( NULL )) ;                            // обновляет базу случайных чисел
   for ( int i = 0 ; i < n ; i++ )                     // инициализирует все элементы массива
       B [ i ]=rand ( )%100 ;                          // случайными числами от 0 до 99
   for ( i = 0 ; i < n ; i++ ) cout << B[ i ] << '\t' ;   // выводит массив случайных чисел
   cout << "\n\n" ;
   for ( i = 0 ; i < n – 1 ; i++ )                     // n-1 раз ищем наименьший элемент
   {
      // принимаем за наименьший первый из рассматриваемых элементов
      int m = i ;
      // поиск номера минимального элемента из неупорядоченных
      for ( int j = i + 1 ; j < n ; j++ )
         // если нашли меньший элемент, запоминаем его номер
         if ( B [ j ] < B [ m ] ) m = j ;
      // обмен элементов с номерами i и m через буфер
      int buf = B [ i ] ;      B [ i ] = B [ m ] ;      B [ m ] = buf ;
   }
   for ( i = 0 ; i < n ; i++ ) cout << B[ i ] << '\t' ;   // выводит упорядоченный массив
   cout << "\n\n" ;
}
```

Процесс обмена элементов массива с номерами **i** и **m** через буферную переменную **buf** на **i**-м проходе цикла проиллюстрирован на рисунке. Цифры около стрелок обозначают порядок действий.



Поскольку для нашего алгоритма несущественно, какие именно числа сортировать, то элементы массива инициализируются случайными числами с помощью функции **rand ( )**, которая возвращает случайное число в диапазоне от **0** до **32767** и объявлена в заголовочном файле **<stdlib.h>**. Выражение **rand ( )%100**, которое использует операцию нахождения остатка от целочисленного деления, уменьшает диапазон случайных чисел (от 0 до 99). Функция **rand ( )** считывает значения из базы случайных чисел, которая поддерживается операционной системой.

В результате массив заполняется случайными числами, но при повторном запуске программы элементы массива будут иметь те же значения, что и при предыдущем. Для того, чтобы значения элементов массива числа были бы “более случайными” – отличались от предыдущего запуска программы, мы обновляем базу случайных чисел с помощью функции **srand ( )**, прототип которой содержится в заголовочном файле **<stdlib.h>** и имеет вид:

```
void srand ( unsigned int seed ) ;
```

где **seed** – параметр генератора случайных чисел.

В качестве параметра генератора случайных чисел удобно использовать системное время, которое возвращает функция **time ( )**. Прототип этой функции, объявленной в заголовочном файле **<stdlib.h>**, имеет вид:

```
time_t time ( time_t *timer ) ;
```

Если параметр функции **timer** равен **NULL**, то функция возвращает количество секунд прошедшее с полуночи 1 января 1970 года.

Поскольку времена различных запусков программы отличаются, то и значения элементов массива будут изменяться от запуска до запуска.

Динамические массивы

В программе каждая переменная может размещаться в одном из трех мест: в области данных программы, в стеке или в свободной памяти (так называемая куча).

Каждой переменной в программе память может отводиться либо статически, то есть в момент загрузки программы, либо динамически – в процессе выполнения программы. До сих пор определяемые массивы объявлялись статически, и, следовательно, хранили значения всех своих элементов в стековой памяти или области данных программы. Если количество элементов массива невелико, такое размещение оправдано. Однако довольно часто возникают случаи, когда в стековой памяти, содержащей локальные переменные и вспомогательную информацию (например, точки возврата из вложенных функций), недостаточно места для размещения всех элементов большого массива. Ситуация еще более усугубляется, если массивов большого размера должно быть много. Здесь на помощь приходит возможность использования для хранения данных динамической памяти.

Кроме того, во многих задачах разработчик заранее не знает размер массива, который должен быть определён в ходе выполнения программы. Поскольку выделять память “с запасом” не эффективно, то использование динамического массива является лучшим решением.

Динамические массивы создают с помощью операции **new**, при этом необходимо указать *тип* и *размер* в соответствии со следующим синтаксисом:

```
тип *имя = new тип [ размер ] ;
```

Пример

```
int n ;                                  // количество элементов массива
cin >> n ;                               // ввод количества элементов массива
// объявляет указатель на нулевой элемент массива и выделяет память для n чисел типа float
float *p = new float [ n ] ;
```

В последней строке создается переменная-указатель на **float**, в динамической памяти отводится непрерывная область, достаточная для размещения **n** элементов вещественного типа, и адрес ее начала записывается в указатель **p**. Динамические массивы нельзя при создании инициализировать, и они не обнуляются.

Преимущество динамических массивов состоит в том, что размерность может быть переменной, то есть объем памяти, выделяемой под массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведенного выше массива можно обратиться как **p [5]** или ***(p + 5)**.

Память, зарезервированная под динамический массив с помощью **new [ ]**, должна освобождаться оператором **delete [ ]**, например:

```
delete [ ] p ;
```

Для иллюстрации использования динамического массива и доступа к элементам массива с помощью указателей решим рассмотренную выше задачу сортировки массива:

```
#include <iostream.h>          // подключает библиотеку функций ввода-вывода
#include <stdlib.h>            // подключает библиотеку, которая содержит функции rand ( ) и srand ( )
#include <time.h>              // подключает библиотеку, которая содержит функцию time ( )
void main ( )
{
   int n = 10 ;                                          // количество элементов массива
   cout << "Input size\t" ;      cin >> n ;               // ввод размера массива
   int *pB = new int [ n ] ;                             // создаёт динамический массив
   int *p ;                                              // указатель на элемент массива
   srand ( time ( NULL )) ;                              // обновляет базу случайных чисел
   int i , j ;                                           // номера элементов массива
   for ( i = 0 , p = pB ; i < n ; i++ )                  // инициализирует все элементы массива
       *p++ = rand ( ) % 100 ;                           // случайными числами от 0 до 99
   for ( i = 0 , p = pB ; i < n ; i++ )
       cout << *p++ << '\t' ;                            // выводит массив случайных чисел
   cout << "\n\n" ;
   for ( i = 0 , p = pB ; i < n – 1 ; i++ , p++ )        // n-1 раз ищем наименьший элемент
   {
       // принимаем за наименьший первый из рассматриваемых элементов
       int m = i ;
       // поиск номера минимального элемента из неупорядоченных
       for ( j = i + 1 ; j < n ; j++ )
           // если нашли меньший элемент, запоминаем его номер
           if ( pB [ j ] < pB [ m ] ) m = j ;
       // обмен элементов с номерами i и m через буфер
       int buf = *p ;       *p = pB [ m ] ;          pB [ m ] = buf ;
   }
   for ( i = 0 , p = pB ; i < n ; i++ )
       cout << *p++ << '\t' ;                            // выводит упорядоченный массив
   cout << "\n\n" ;
   delete [ ] pB ;
}
```

МНОГОМЕРНЫЕ МАССИВЫ

Многомерный массив размерности **N** можно представить как одномерный массив из массивов размерности **(N – 1)**. Таким образом, например трехмерный массив – это массив, каждый элемент которого представляет двумерную матрицу.

Описание многомерного массива

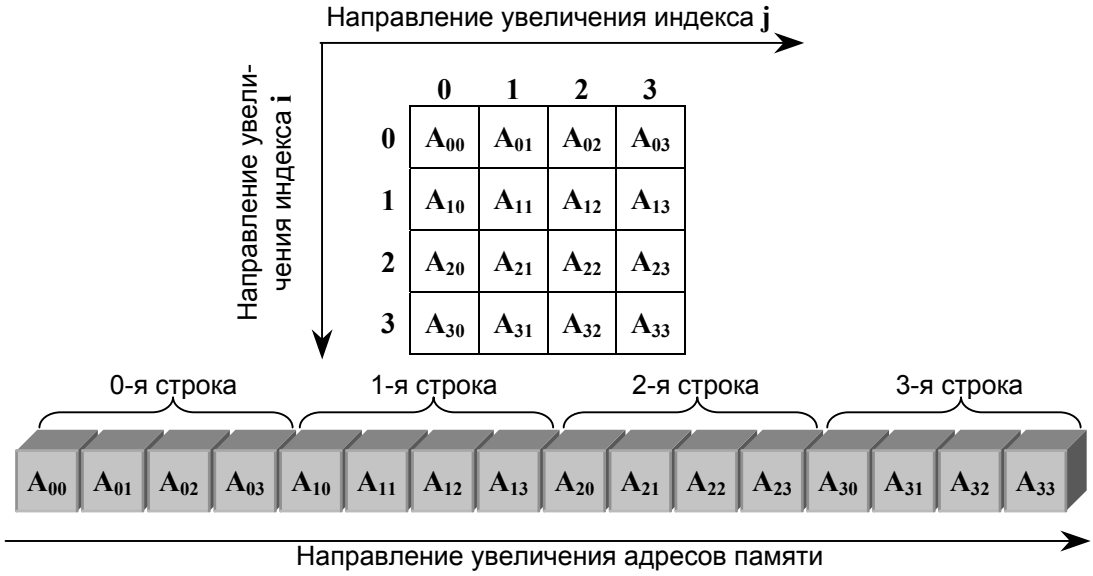
Многомерные массивы задаются указанием каждого измерения в квадратных скобках. Примеры объявления многомерных массивов:

```
char Mat [ 6 ][ 9 ] ;                  // двумерный массив 6×9 элементов
unsigned long Arr [ 4 ][ 2 ][ 8 ] ;    // трехмерный массив 4×2×8 элементов
```

Выражение **A [i][j]**, представляющее элемент двумерного массива, переводится компилятором в эквивалентное выражение:

```
* ( * ( A + i ) + j )
```

Элементы двумерного массива **A [i][j]** располагаются в оперативной памяти *построчно*, как показано на рисунке – сначала элементы нулевой строки (**i = 0**), затем первой (**i = 1**), второй (**i = 2**) и т.д.



Инициализация многомерного массива

Многомерные массивы инициализируются в порядке наискорейшего изменения самого правого индекса (задом наперед): сначала происходит присвоение начальных значений всем элементам последнего индекса, затем предыдущего и т.д. до самого начала:

```
int Mass [ 2 ][ 2 ][ 3 ] = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 } ;
```

В этом случае элементы массива будут получать следующие значения:

```
Mass [ 0 ][ 0 ][ 0 ] =  1 ;    Mass [ 0 ][ 0 ][ 1 ] =  2 ;    Mass [ 0 ][ 0 ][ 2 ] =  3 ;
Mass [ 0 ][ 1 ][ 0 ] =  4 ;    Mass [ 0 ][ 1 ][ 1 ] =  5 ;    Mass [ 0 ][ 1 ][ 2 ] =  6 ;
Mass [ 1 ][ 0 ][ 0 ] =  7 ;    Mass [ 1 ][ 0 ][ 1 ] =  8 ;    Mass [ 1 ][ 0 ][ 2 ] =  9 ;
Mass [ 1 ][ 1 ][ 0 ] = 10 ;    Mass [ 1 ][ 1 ][ 1 ] = 11 ;    Mass [ 1 ][ 1 ][ 2 ] = 12 ;
```

Чтобы не запутаться, для наглядности можно группировать данные с помощью промежуточных фигурных скобок:

```
int Mass [ 2 ][ 2 ][ 3 ] = {{{ 1 , 2 , 3 } , { 4 , 5 , 6 }} , {{ 7 , 8 , 9 } , { 10 ,11 , 12 }}} ;
```

Для многомерных массивов при инициализации разрешается опускать только величину первой размерности:

```
int Mat [ ][ 3 ] = {{ 9 , 8 , 7 } , { 6 , 5 , 4 } , { 3 , 2 , 1 }} ;
```

Доступ к элементам многомерного массива

Доступ к элементам многомерного массива через указатели осуществляется немного сложнее. Поскольку, например, двумерный массив **Mat [i][j]** может быть представлен как одномерный (**Mat[i]**), каждый элемент которого также является одномерным массивом (**Mat [j]**), указатель на двумерный массив **pMat**, ссылаясь на элемент массива **Mat [i][j]**, по сути, указывает на массив **Mat [j]** в массиве **Mat [i]**. Таким образом, для доступа к содержимому ячейки указатель **pMat** придется разыменовывать дважды.

```
char Arr [ 3 ][ 2 ] = { 'W' , '0' , 'R' , 'L' , 'D' ,' ! ' } ;
char* pArr = (char*) Arr ;
pArr += 3 ;
char Letter = *pArr ;
cout << Letter ;
```

В приведенном примере объявляется массив символов размерностью 3×2 и указатель **pArr** на него (фактически – указатель на **Arr [0][0]**). В строке

```
char* pArr = (char*) Arr ;
```

идентификатор **Arr** уже является указателем на элемент с индексом 0, однако, поскольку массив двумерный, требуется его повторное разыменование.

Увеличение **pArr** на 3 приводит к тому, что он указывает на элемент массива, значение которого – символ **'L'** (элемент **Arr [1][1]**). Далее осуществляется вывод содержимого ячейки массива, на которую указывает **pArr**.

Рассмотрим пример, в котором на экран выводится двумерный массив размером **n×m** в виде таблицы. Обращение к элементам массива осуществляется с помощью индексов – номера строки **i** и номера столбца **j**.

```
const n = 3 , m = 4 ;                    // размер массива
// объявление и инициализация двумерного массива
int Mat [ n ][ m ] = {{ 1 , 2 , 3 , 4 } , { 5 , 6 , 7 , 8 } , { 9 , 10 , 11 , 12 }} ;
for ( int i = 0 ; i < n ; i++)           // для всех номеров строк i от 0 до n
{
   cout << "i = " << i << '\t' ;          // вывод на экран номер строки
   for ( int j = 0 ; j < m ; j++ )       // для всех номеров столбцов j от 0 до m
   cout << Mat [ i ][ j ] << '\t' ;      // вывод на экран элемент массива из i строки и j столбца
   cout << '\n' ;                        // перевод каретки на начало следующей строки
}
```

На экран будет выведено:

```
i = 0    1    2    3    4
i = 1    5    6    7    8
i = 2    9    10   11   12
```

Рассмотрим пример, в котором демонстрируется, что идентификатор **Mat + j** является указателем на **j** строку и что двумерный массив является массивом одномерных массивов.

```
for ( int i =0 ; i < n ; i++) cout << "i =" << i << '\t' << Mat + i << '\n' ;
```

На экран будет выведено, например:

```
i = 0    0x0012FF48
i = 1    0x0012FF58
i = 2    0x0012FF68
```

Приведённый фрагмент кода выводит на экран значения указателей на строки. Разница между адресами составляет 4×4 байта =16 байт.

Разыменовывание указателей на строки позволяет получить указатели на элементы двумерного массива, что показано в следующем примере:

```
for ( int i = 0 ; i < n ; i++)           // для всех номеров строк i от 0 до n
{
   cout << "i = " << i << '\t' ;          // вывод на экран номер строки
   for ( int j = 0 ; j < m ; j++)        // для всех номеров столбцов j от 0 до m
   // вывод на экран значения указателя на элемент массива из i строки и j столбца
   cout << *( Mat + i ) + j << '\t' ;
   cout << '\n' ;                        // перевод каретки на начало следующей строки
}
```

На экран будет выведено, например:

```
i = 0    0x0012FF48    0x0012FF4C    0x0012FF50    0x0012FF54
i = 1    0x0012FF58    0x0012FF5C    0x0012FF60    0x0012FF64
i = 2    0x0012FF68    0x0012FF6C    0x0012FF70    0x0012FF74
```

Разыменовывание указателей на элементы двумерного массива позволяет получить значения самих элементов, что показано в следующем примере:

```
for ( int i = 0 ; i < n ; i++)           // для всех номеров строк i от 0 до n
{
   cout<<" i = "<< i << '\t' ;           // вывод на экран номер строки
   for ( int j = 0 ; j < m ; j++)        // для всех номеров столбцов j от 0 до m
   // вывод на экран элемент массива из i строки и j столбца
   cout<< *(*( Mat + i ) + j ) << '\t' ;
   cout << '\n' ;                        // перевод каретки на начало следующей строки
}
```

Поскольку элементы двумерного массива располагаются в памяти в том же порядке, что и выводятся на экран, воспользуемся этим обстоятельством в следующем примере для решения той же задачи:

```
int* pMat ;                              // объявление указателя на элемент массива
// преобразование указателя на строку в указатель на элемент массива
```

```
pMat = ( int* ) Mat ;                          // инициализация указателя на элемент массива
for ( int k=0 ; k < n*m ; k++)                 // для всех элементов массива
{
   if ( k%m == 0 ) cout << "i = " << k / m << '\t' ;  // вывод на экран номер строки
   cout << *pMat << '\t' ;                     // вывод на экран элемент массива
   if (( k + 1 ) % m == 0 ) cout << '\n' ;     // перевод каретки на начало следующей строки
   *pMat ++ ;                                  // увеличение значения указателя
}
```

Многомерные динамические массивы

Для создания динамического многомерного массива необходимо указать в операции **new** все его размеры, причём самый левый размер может быть переменным. Например:

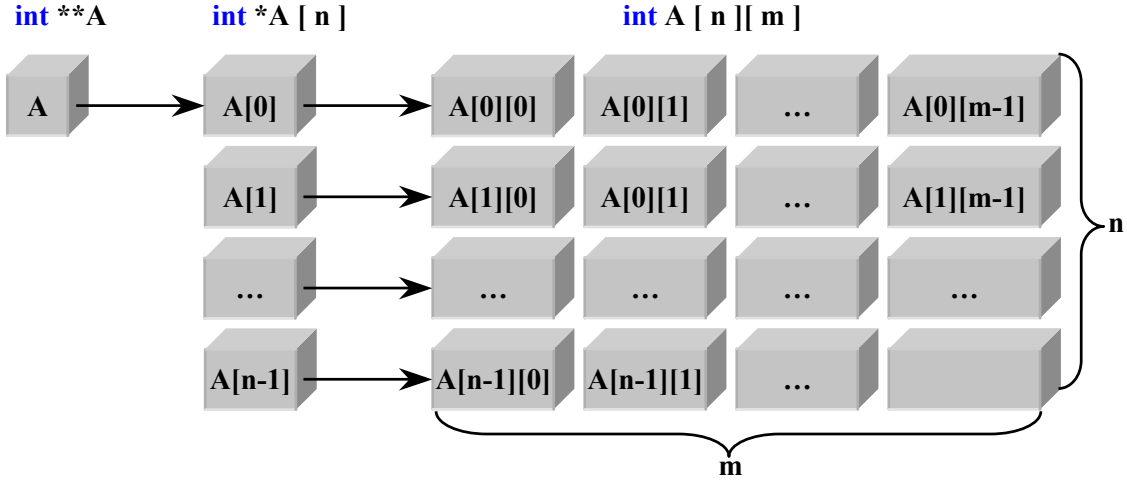
```
int n = 5 ;
int ** A = ( int ** ) new int [ n ][ 10 ] ;
```

Освобождение памяти из-под массива любой размерности выполняется с помощью операции **delete [ ]**.

Более *универсальный* и безопасный способ выделения памяти под двумерный массив, когда оба его размера задаются на этапе выполнения программы, приведен ниже:

```
int n = 5 , m = 6 ;                            // размер массива
// определяет указатель на указатель на int и выделяет память для массива указателей на строки
int ** A = new int *[ n ] ;
for ( int i = 0 ; i < n ; i++)                 // для всех строк
   A [ i ] = new int [ m ] ;                   // выделяет память для строки
```

Сначала определяется переменная типа "указатель на указатель на **int**" и выделяется память под массив указателей на строки массива. Массив указателей содержит **n** элементов типа **int***. Далее организуется цикл для выделения памяти под каждую строку матрицы. В последнем операторе каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из **m** элементов типа **int**.



Освобождать память, выделенную под динамический массив, необходимо также в два приёма – сначала освобождается память, выделенная под строки матрицы, а затем память, выделенная под указатели на строки:

```
for ( int i = 0 ; i < n ; i++)                 // для всех строк
   delete [ ] A [ i ] ;                        // освобождает память, выделенную для i-й строки
delete [ ] A ;                                 // освобождает память, выделенную для указателей на строки
```

МАССИВЫ В КАЧЕСТВЕ ПАРАМЕТРОВ ФУНКЦИЙ

В тело функций в качестве аргументов можно передавать значения, хранящиеся в массивах. При вызове функции, параметр типа массива преобразовывается компилятором в указатель на тип массива. Например, если аргумент-массив имеет тип **unsigned long**, при вызове он будет преобразован в **unsigned long***. Таким образом, изменение в функции значения любого элемента массива, являющегося аргументом, обязательно повлияет и на оригинал. Массивы отличаются от других типов тем, что их нельзя передавать по значению – внутрь тела функции попадает только адрес массива.

Синтаксис вызова функции при этом может быть следующим:

FunctionName (ArrayName) ;

Тогда прототип функции включает указание в качестве параметра типа передаваемого массива и следующих за ним прямоугольных скобок. Например:

FunctionName (char [ ]) ;

Другой вариант синтаксиса передачи массива в функцию – когда прототип функции содержит символ операции взятия адреса после указания типа аргумента:

char FunctionName (char&) ;

При этом синтаксис вызова функции принимает следующий вид:

FunctionName (*ArrayName) ;

Ниже приводится пример, иллюстрирующий оба варианта передачи массива в качестве параметра функции.

```
#include <iostream.h>                          // подключает библиотеку функций ввода-вывода
void Out1 ( int* , short ) ;                   // прототип первой функции
void Out2 ( int& , short ) ;                   // прототип второй функции
void main ()
{
   int Array [ ] = { 10 , 8 , 6 , 4 , 2 , 0 } ;          // объявление и инициализация массива
   short n = sizeof ( Array ) / sizeof ( Array [ 0 ] ) ;  // размер массива
   Out1 ( Array , n ) ;                        // вызов первой функции
   Out2 ( *Array , n ) ;                       // вызов второй функции
}

void Out1 ( int arr* , short n )               // реализация первой функции
{
   for ( int i = 0 ; i < n ; i++ ) cout << arr [ i ] << '\t' ;
   cout << '\n' ;
}
void Out2 ( int& arr , short n )               // реализация второй функции
{
   for ( int i = 0 ; i < n ; i++) cout << *( & ( arr ) + i ) << '\t' ;
   cout << '\n' ;
}
```

В рассмотренном примере объявляется массив **Array [ ]**, содержащий шесть целочисленных значений, и осуществляется его передача в функции **Out1** и **Out2**. Обе функции выполняют одно и то же действие – выводят содержимое массива-аргумента на печать. Функция **Out1** получает в качестве параметра указатель на массив, а функция **Out2** – ссылку на массив. Кроме того, объявляется, инициализируется и передаётся в качестве параметра переменная n, которая хранит количество элементов массива.

Использование в качестве параметра функции многомерного массива затруднено, поэтому на практике чаще всего осуществляется передача массива указателей, что значительно упрощает синтаксис.