Arrays

The variables which we have used before are single variables which represent single quantities. There are situations where we need to store/access more than one value that are related to each other (i.e. they belong to same data set).

Linear Array (1-D array)

A linear array (1-D array) is a list of a finite number *n* of homogeneous data elements (i.e. elements of same data type) such that:

a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers i.e. referred by the position of the element in the group.
b) The elements of the array are stored respectively in successive memory locations.

The number *n* of elements is called the *length* of the array. The elements of the array A are denoted by the parenthesis notation

A[0],A[1], A[2], A[3], A[4], A[5], A[6], A[7] etc.

Suppose we want to store the marks obtained by 50 students of a class. In such a case we have two options to store these marks in the memory:

1. Construct 50 variables to store the percentage marks obtained by each student i.e. each variable containing one student's marks.
2. Construct one variable (subscripted) that is capable of holding all the values.

The second approach seems to be feasible because its difficult to manage fifty names. It is very difficult to manage a program with such variable declarations

int a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28, a29, a30, a31, a32, a33, a34, a35, a36, a37, a38, a39, a40, a41, a42, a43, a44, a45, a46, a47, a48, a49, a50;

Now there are two ways to count fifty elements:

METHOD 1: 0 to 49 METHOD 2: 1 to 50

C FOLLOWS THE FIRST COUNTING METHOD. FIRST ELEMENT IS NUMBERED 0

If the array is linear its called 1-D array. If the array is in the form of a matrix its 2-D array.

1-D arrays are declared as

***DataType ArrayName*[*Totalelements*]**

e.g. int arr[50];
float b[30]

First element of an array arr is arr[0]

A sample program to read and write elements of 1-D array is given as

```
main() {
 int a[5]; /*first tell compiler what is the type and size of array*/
 clrscr();
 printf("\nEnter the elements of array ");
 for(i=0;i<5;i++)
  scanf("%d",&a[i] ); /*you are always going to read one element at a
time. Observe how it is read */
 printf("\nThe array elements are: ");
 for(i=0;i<5;i++)
  printf("\n%d",a[i] ); /*even at time of printing there is one
element written at a time.*/
 getch();
}
```

Before we get into details of higher dimension arrays here are a few points to remember about arrays:

1. Before using an array, its size must be declared
2. First element is numbered 0, so last element is one less than total elements.
   int a[5] will have elements a[0], a[1], a[2], a[3], a[4]
3. How big the array might be, its elements are always stored in the consecutive memory locations.
4. If desired the elements of the array can be initialized at the same point at the time of declaration

```
int num[5] = { 2,4,6,7,8};
int n[] = {4,5,6,7,8,2}; /*read the next point for its explanation*/
```

5. If array is initialized in its declaration as in above example-2; there is no need to write the number of elements in the declaration, the compiler can count them by itself.
6. TILL THE ARRAY ELEMENTS ARE NOT GIVEN ANY VALUE, THEY CONTAIN GARBAGE VALUE LIKE ANY OTHER ORDINARY VARIABLES.
7. IN C THERE IS NO CHECK TO SEE IF THE SUBSCRIPT SPECIFIED IN THE PROGRAM IS IN THE RANGE OR NOT. IF IT EXCEEDS THE VALID RANGE (like you write a[10] for an array who is declared as int a[5]) IT IS QUITE LIKELY THAT DATA WILL BE WRITTEN AHEAD OF THE ARRAY STORAGE SPACE (ON TOP OF SOME OTHER DATA). IT CAN EVEN CRASH YOUR PROGRAM/COMPUTER.

```
main() {
 int a[10];
 int i;
 for(i=0;i<50;i++)
   a[i] = i;
}
```

It is the responsibility of the programmer not to exceed the array limits (called as array bounds). The compiler doesn't flash any error upon compilation. You can see unpredictable results only at the execution of the program.

2-D Array

A two dimensional $m \times n$ array A is a collection of $m.n$ data elements such that each data element is specified by a pair of integers (i,j) called *subscripts* with the property that

$0 \leq i < m$ and $0 \leq j < n$

The element with first subscript i and second subscript j shall be written as $A[i][j]$

A 2-D array is also called as *matrix* and is a collection of *row* and *columns*.

Array is declared as

***DataType ArrayName*[TotalRowElements][*TotalColelements*]**

e.g. float arr[3][3]

Since memory is linear, the array must be stored linearly in the memory. For this Row Major order of storage of array is followed.

A[0][0]	A[0][1]	A[0][2]	A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]	A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]	A[2][0]	A[2][1]	A[2][2]

A[0][0] A[0][1] A[0][2] A[1][0] A[1][1] A[1][2] A[2][0] A[2][1] A[2][2]
Row Major Order

A[0][0] A[1][0] A[2][0] A[0][1] A[1][1] A[2][1] A[0][2] A[1][2] A[2][2]
Column Major Order

Initializing a 2-D array:

Just like 1-D array was initialized by giving values of its member elements, a 2-D array is also initialized using the same method. Consider a 2-D array storing student roll number and marks

```
int arr[4][2] =  { {111, 67}, {222, 57}, {333, 73}, {444, 64} };
```

Observe the usage of commas and parenthesis { }

Storage of array elements in case of 2-D array:

Asuming that array starts at memory location 5002. Observe that the 2-D array is always stored in one dimension.

arr[0][0]	arr[0][1]	arr[1][0]	arr[1][1]	arr[2][0]	arr[2][1]	arr[3][0]	arr[3][1]
111	67	222	57	333	73	444	64
5002	5004	5006	5008	5010	5012	5014	5016