MODULE III

ARRAY

An array is a collective name given to a group of ‘similar quantities’. These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees. These similar elements could be all ints, or all floats, or all chars, etc. Usually, the array of characters is called a ‘string’, whereas an array of ints or floats is called simply an array.
assume the following group of numbers, which represent percentage marks obtained by five students.
per = { 48, 88, 34, 23, 96 }
In C, the fourth number is referred as per[3]. This is because in C the counting of elements begins with 0 and not with 1. Thus, in this example per[3] refers to 23 and per[4] refers to 96. In general, the notation would be per[i], where, i can take a value 0, 1, 2, 3, or 4, depending on the position of the element being referred. Here per is the subscripted variable (array), whereas i is its subscript.

Declaration
Like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want.

```
int marks[30] ;
```

Here, int specifies the type of the variable and the word marks specifies the name of the variable.The number 30 tells how many elements of the type int will be in our array. This number is often called the ‘dimension’ of the array. The bracket ([ ]) tells the compiler that we are dealing with an array.

Accessing array elements in an array

Accessing array can be done with subscript, the number in the brackets following the array name. This number specifies the element’s position in the array. All the array elements are numbered, starting with 0. Thus, marks[2] is not the second element of the array, but the third. we are using the variable i as a subscript to refer to various elements of the array. This variable can take different values and hence can refer to the different elements in the array in turn.

ARRAY INITIALIZATION

Arrays can be initialised along with declaration.Consider the examples,

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;
int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;
```

BOUNDS CHECKING

In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself.
Thus, to see to it that we do not reach beyond the array size is entirely the programmer’s botheration and not the compiler’s.

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function. These two calls are illustrated below:

```
/* Demonstration of call by value */
main( )
{
        int i ;
        int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
        for ( i = 0 ; i <= 6 ; i++ )
                display ( marks[i] ) ;
}

display ( int m )
{
        printf ( "%d ", m ) ;
}
```

Here, we are passing an individual array element at a time to the function display( ) and getting it printed in the function display( ). Since at a time only one element is being passed, this element is collected in an ordinary integer variable m, in the function display( ).

```
/*Demonstration of call by reference */
main( )
{
        int i ;
        int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
        for ( i = 0 ; i <= 6 ; i++ )
                disp ( &marks[i] ) ;
}

disp ( int *n )
{
        printf ( "%d ", *n ) ;
}
```

Here, we are passing addresses of individual array elements to the function display( ). Hence, the variable in which this address is collected (n) is declared as a pointer variable. And since n contains the address of array element, to print out the array element we are using the ‘value at address’ operator (*).

POINTERS & ARRAYS

The method to accesses the array elements using pointers:

```
main( )
{
        int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
        int i, *j ;
        j = &num[0] ; /* assign address of zeroth element */
```

```
        for ( i = 0 ; i <= 5 ; i++ )
        {
                printf ( "\naddress = %u ", j ) ;
                printf ( "element = %d", *j ) ;
                j++ ; /* increment pointer to point to next location */
        }
}
```

Here, to begin with we have collected the base address of the array (address of the 0 th element) in the variable j using the statement,

```
j = &num[0] ; /* assigns address of first element to j */
```

When we are inside the loop for the first time, j contains the address of the first element of array , and the value at that address. These are printed using the statements,

```
printf ( "\naddress = %u ", j ) ;
printf ( "element = %d", *j ) ;
```

On incrementing j it points to the next memory location (i.e the address of second element) therefore when the printf( ) statements are executed for the second time they print out the second element of the array and its address and so on till the last element of the array has been printed.

Passing an Entire Array to a Function

Now we can see how to pass an entire array to a function rather than its individual elements. Consider the following example:

```
/* Demonstration of passing an entire array to a function */
main( )
{
        int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
        dislpay ( &num[0], 6 ) ;
}

display ( int *j, int n )
{
        int i ;
        for ( i = 0 ; i <= n - 1 ; i++ )
        {
                printf ( "\nelement = %d", *j ) ;
                j++ ; /* increment pointer to point to next element */
        }
}
```

Here, the display( ) function is used to print out the array elements. Note that the address of the zeroth element is being passed to the display( ) function. The for loop is same as the one used in the earlier program to access the array elements using pointers. Thus, just passing the address of the zeroth element of the array to a function is as good as passing the entire array to the function. It is also necessary to pass the total number of elements in the array, otherwise the display( ) function would not know when to terminate the for loop.

Address of the zeroth element (many a times called the base address) can also be passed by just passing the name of the array. Thus, the following two function calls are same:

```
display ( &num[0], 6 ) ;
display ( num, 6 ) ;
```

TWO DIMENSIONAL ARRAYS

We have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two- dimensional array is also called a matrix. Here is a sample program that stores roll number and marks obtained by a student side by side in a matrix.

```
main( )
{
        int x[4][2] ;
        int i, j ;
        for ( i = 0 ; i <= 3 ; i++ )
        {
                for(j=0;j<=1;j++)
                {
                        printf ( "\n Enter student marks" ) ;
                        scanf ( "%d", &x[i][j] ) ;
                }
        }
        for ( i = 0 ; i <= 3 ; i++ )
        {
                for(j=0;j<=1;j++)
                {
                        printf ( "\n%d %d", x[i][j]) ;
                }
        }
}
```

In stud[i][j], the first subscript of the variable stud,is row number which changes for every student. The second subscript tells which of the two columns are we talking about.The counting of rows and columns begin with zero. The complete array arrangement is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initialising a 2-Dimensional Array

A 2D array can be initialized in two ways:

```
int stud[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                } ;
```

or

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declaration, int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ; is perfectly acceptable
where as, int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ; would never work.

Memory Map of a 2-Dimensional Array

This is because memory doesn't contain rows and columns. In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

Pointers and 2-Dimensional Arrays

We can access array elements of a two-dimensional array using pointers.

Thus, the declaration,

```
int s[5][2] ;
```

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript. Similarly, if we can imagine s to be a one-dimensional array then we can refer to its zeroth element as s[0], the next element as s[1] and so on. More specifically, s[0] gives the address of the zeroth one-dimensional array, s[1] gives the address of the first one- dimensional array and so on.

```
/* Demo: 2-D array is an array of arrays */
main( )
{
        int s[4][2] = {
                          { 1234, 56 },
                          { 1212, 33 },
                          { 1434, 80 },
                          { 1312, 78 }
                        } ;
        int i ;
        for ( i = 0 ; i <= 3 ; i++ )
                printf ( "\nAddress of %d th 1-D array = %u", i, s[i] ) ;
}
```

And here is the output...

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65512
Address of 2 th 1-D array = 65516
Address of 3 th 1-D array = 65520
```

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

The expressions s[0] and s[1] would yield the addresses of the zeroth and first one-dimensional array respectively. From Figure these addresses turn out to be 65508 and 65512.
Suppose we want to refer to the element s[2][1] using pointers. We know (from the above program) that s[2] would give the address 65516, the address of the second one-dimensional array. Obviously (65516 + 1) would give the address 65518. Or (s[2] + 1) would give the address 65518. And the value at this address can be obtained by using the value at address operator, saying *(s[2] + 1).

But, we have already studied while learning one-dimensional arrays that num[i] is same as *(num + i). Similarly, *(s[2] + 1) is same as, *(*(s + 2) + 1). Thus, all the following expressions refer to the same element,

```
s[2][1]
* ( s[2] + 1 )
* ( * ( s + 2 ) + 1 )
```

Using these concepts the following program prints out each element of a two-dimensional array using pointer notation.

```
/* Pointer notation to access 2-D array elements */
main( )
{
        int s[4][2] = {
                          { 1234, 56 },
                          { 1212, 33 },
                          { 1434, 80 },
                          { 1312, 78 }
                        } ;
                          int i, j ;
        for ( i = 0 ; i <= 3 ; i++ )
        {
                printf ( "\n" ) ;
                for ( j = 0 ; j <= 1 ; j++ )
                        printf ( "%d ", *( *( s + i ) + j ) ) ;
        }
}
```

And here is the output...

```
1234 56
1212 33
1434 80
1312 78
```

Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointer to a char, then we cant have a pointer to an array.

```
/* Usage of pointer to an array */
main( )
{
                int s[5][2] = {
                                  { 1234, 56 },
                                  { 1212, 33 },
                                  { 1434, 80 },
                                  { 1312, 78 }
                                } ;
                int ( *p )[2] ;
                int i, j, *pint ;
                for ( i = 0 ; i <= 3 ; i++ )
                {
                        p = &s[i] ;
                        pint = p ;
                        printf ( "\n" ) ;
                        for ( j = 0 ; j <= 1 ; j++ )
                                printf ( "%d ", *( pint + j ) ) ;
                }
}
```

And here is the output...
1234 56
1212 33
1434 80
1312 78

Here p is a pointer to an array of two integers. Note that the parentheses in the declaration of p are necessary. Absence of them would make p an array of 2 integer pointers. In the outer for loop each time we store the address of a new one-dimensional array. Thus first time through this loop p would contain the address of the zeroth 1-D array. This address is then assigned to an integer pointer pint. Lastly, in the inner for loop using the pointer pint we have printed the individual elements of the 1-D array to which p is pointing.

Passing 2-D Array to a Function

There are three ways in which we can pass a 2-D array to a function. These are illustrated in the following program.

```
/* Three ways of accessing a 2-D array */
main( )
{
        int a[3][4] = {
                          1, 2, 3, 4,
                          5, 6, 7, 8,
                          9, 0, 1, 6
                       } ;
        clrscr( ) ;
        display ( a, 3, 4 ) ;
        show ( a, 3, 4 ) ;
        print ( a, 3, 4 ) ;
}

display ( int *q, int row, int col )
{
        int i,j;
        for(i= 0 ; i < row ; i++ )
        {
                for ( j = 0 ; j < col ; j++ )
                printf ( "%d ", * ( q + i * col + j ) ) ;
                printf ( "\n" ) ;
        }
        printf ("\n" ) ;
}

show ( int ( *q )[4], int row, int col )
{
        int i, j ;
        int *p ;
        for ( i = 0 ; i < row ; i++ )
        {
                p = q + i ;
```

```
                for ( j = 0 ; j < col ; j++ )
                        printf ( "%d ", * ( p + j ) ) ;
                printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
}

print ( int q[ ][4], int row, int col )
{
        int i, j ;
        for ( i = 0 ; i < row ; i++ )
        {
                for ( j = 0 ; j < col ; j++ )
                        printf ( "%d ", q[i][j] ) ;
                printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
}
```

And here is the output...

```
1 2 3 4
5 6 7 8
9 0 1 6

1 2 3 4
5 6 7 8
9 0 1 6

1 2 3 4
5 6 7 8
9 0 1 6
```

In the **display( ) function** we have collected the base address of the 2-D array being passed to it in an ordinary int pointer. Then through the two for loops using the expression * (q + i * col + j) we have reached the appropriate element in the array. Suppose i is equal to 2 and j is equal to 3, then we wish to reach the element a[2][3].

1	2	3	4	5	6	7	8	9	0	1	6
65502	65504	65506	65508	65510	65512	65514	65516	65518	65520	65522	65524

The expression * (q + i * col + j) becomes * (65502 + 2 * 4 + 3) . This turns out to be * (65502 + 11). Since 65502 is address of an integer, * (65502 + 11) turns out to be * (65524). Value at this address is 6. This is indeed same as a[2][3]. The general formula would be:
* (base address + row no. * no. of columns + column no.)

In the **show( ) function** we have defined q to be a pointer to an array of 4 integers through the declaration:
int (*q)[4] ;
To begin with, q holds the base address of the zeroth 1-D array,This address is then assigned to p, an int pointer, and then using this pointer all elements of the zeroth 1- D array are accessed. Next time through the loop when i takes a value 1, the expression q + i fetches the address of the first 1-D

array. This is because, q is a pointer to zeroth 1-D array and adding 1 to it would give us the address of the next 1-D array. This address is once again assigned to p, and using it all elements of the next 1-D array are accessed.

In the third function **print( )**, the declaration of q looks like this:
int q[ ][4] ;
This is same as int (*q)[4], where q is pointer to an array of 4 integers. The only advantage is that we can now use the more familiar expression q[i][j] to access array elements

Array of Pointers

The way there can be an array of ints or an array of floats, similarly there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

```
main( )
{
        int *arr[4] ; /* array of integer pointers */
        int i = 31, j = 5, k = 19, l = 71, m ;
        arr[0] = &i ;
        arr[1] = &j ;
        arr[2] = &k ;
        arr[3] = &l ;
        for ( m = 0 ; m <= 3 ; m++ )
                printf ( "%d ", * ( arr[m] ) ) ;
}
```

The below figure shows the contents and the arrangement of the array of pointers in memory. As you can observe, arr contains addresses of isolated int variables i, j, k and l. The for loop in the program picks up the addresses present in arr and prints the values present at these addresses.