Arrays

Definition- An array is a collection of similar data in contiguous memory location.

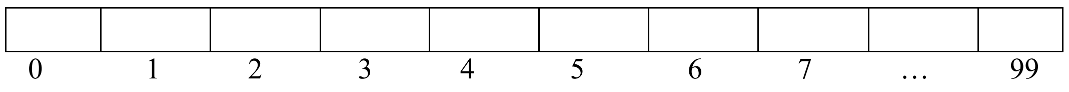
Importance- situations in which we would want to store more than one value at a time in a single variable, we have to use array.

Real thing: For example, suppose we wish to store the percentage marks obtained by 100 students. In such a case we have two options to store these marks in memory:

i) Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.

ii) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Now it is very difficult to keep track of 100 variables in a program. So the first option is not handy. Arrays offer a solution to this problem. An array is a multi-element box and uses an indexing system to find each variable stored within it. In C, indexing starts at zero.
We can create an array namely percentage, which has space for 100 integer variables.



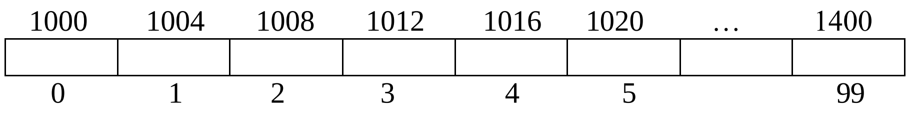
Thus in this example the percentage[0] stores the percentage obtained by the first student, percentage[1] stores the percentage obtained by the second student and so on.
In general the notation would be percentage[i] where **i** can take a value 0,1,2….99, depending on the position of the element being referred. Here percentage is the subscripted variable (array) and 'i' is its subscript.

Note: An array is a collection of similar elements. These similar elements could be all ints, floats, or chars. Usually the array of character is called a 'string' where as an array of ints or floats is called simply an array. Remember that all element of any given array must be of the same type. So we can't have an array of 10 numbers, of which 5 are ints and 5 are floats.

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.

The general form of declaring an array is –
Data type variable_name[size];
Example- float percent[100];

Note: when we make above declaration 400 bytes space get immediately reserved in memory, 4 bytes each for the 100 floats and since the array is not being initialized, all 100 values present in it would be garbage values. This is so happen because the storage class of this array is assumed to be auto. If the storage class is declared to be static then all the array elements would have a default initial value as zero. All the array elements are always present in contiguous memory location as follows-



Accessing Elements of an Array:

Once an array is declared, individual elements in the array can be referred by the 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. Consider the following code fragment-

```
for ( i = 0 ; i <= 99 ; i++ )
{
  printf ( "\nEnter percentage " ) ;
  scanf ( "%f", &percentage[i] ) ;
 }
```

Here the variable **i** is a subscript that refers 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.
The **for** loop causes the process of asking for and receiving a student's percentage from the user to be repeated 100 times. The first time through the loop, **i** has a value 0, so the **scanf( )** function will cause the value typed to be stored in the array element **percentage [0]**, the first element of the array. This process will be repeated until **i** becomes 99.

Array Initialisation:

If the array is initialized where it is declared, mentioning the dimension of the array is optional as in the 2nd example below-

```
int marks[5] = { 92, 49, 75, 97, 67 } ;
int num[ ] = { 1, 2 ,3, 4, 7 } ;
```

Note: Till the array elements are not given any specific values, they are supposed to contain garbage values.

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. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size. In some cases the computer may just hang. 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.
Thus, the following program may turn out to be suicidal-

```
void main()
 {
    int num[100], i ;
    for ( i = 0 ; i <= 500 ; i++ )
    printf("%d",i);
  }
```

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:

Call by value	**Call by reference**
void main()	void main()
{	{
int i ;	int i ;
int number[ ] = { 5, 7, 8, 9, 10 } ;	int number[ ] = { 5, 7, 8, 9, 10 } ;
for (i = 0 ; i <= 4 ; i++)	for (i = 0 ; i <= 4 ; i++)
display (number[i]);	display (&number[i]);
}	}
display (int m)	display (int *m)
{	{
printf ("%d ", m) ;	printf ("%d ", *m) ;
}	}
…………………………………………	…………………………………………..
Note: 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( )**.	**Note:** Here, we are passing addresses of individual array elements to the function **display( )**. Hence, the variable in which this address is collected (**m**) is declared as a pointer variable. And since **m** contains the address of array element, to print out the array element we are using the 'value at address' operator **(*)**.

Passing an Entire Array to a Function: Consider the following program-

```
void 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.
    }
   }
```

Note: Here, the **display( )** function is used to print out the array elements. The address of the zeroth element (often called the base address of the array) 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. 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 ) ;
```

Problem set-1

1. Twenty-five numbers are entered from the keyboard into an array. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array.

2. Twenty-five numbers are entered from the keyboard into an array. Write a program to find out how many of them are positive, how many are negative, how many are even and how many odd.

3. WAP to find out the maximum & the minimum element of an array.

4. WAP to arrange 100 numbers in ascending order using bubble sort technique.

5. WAP to copy the contents of one array into another in the reverse order.