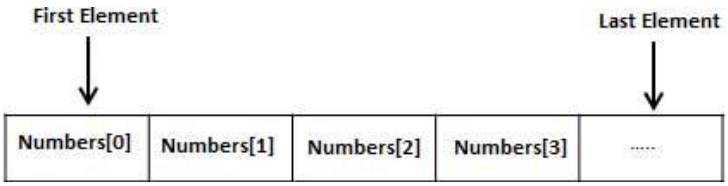
C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The array may be categorized into –

- One dimensional array
- Two dimensional array
- Multidimensional array

1.10.1 Representation of One-Dimensional Array

In Pascal language we can define array as

VAR X: array [1 … N] of integer {or any other type}

That's means the structure contains a set of data elements, numbered (N), for example called (X), its defined as type of element, the second type is the index type, is the type of values used to access individual element of the array, the value of index is

$$1 <= I =< N$$

By this definition the compiler limits the storage region to storing set of element, and the first location is individual element of array , and this called the Base Address, let's be as 500. Base Address (501) and like for the all elements and used the index I, by its value are range 1<= I => N according to Base Index (500), by using this relation:

$$\text{Location (X[I])} = \text{Base Address} + (I-1)$$

When the requirement to bounding the forth element (I=4):

$$\text{Location (X[4])} = 500 + (4-1)$$

= 500 +3

= 503

So the address of forth element is 503 because the first element in 500.

When the program indicate or dealing with element of array in any instruction like (write (X [I]), read (X [I])), the compiler depend on going relation to bounding the requirement address.

1.10.2 Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

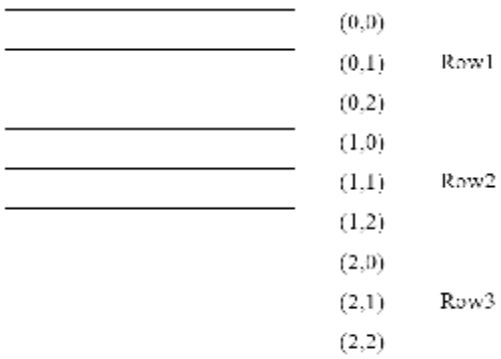
Where **type** can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

1.10.2.1 Representation of two dimensional arrays in memory
A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

- Row Major Order: First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.



To determine element address A[i,j]:

$$\text{Location}(A[i,j]) = \text{Base Address} + (N \times (I - 1)) + (j - 1)$$

For example:

Given an array [1…5,1…7] of integers. Calculate address of element T[4,6], where BA=900.

Sol) I = 4 , J = 6

M= 5 , N= 7

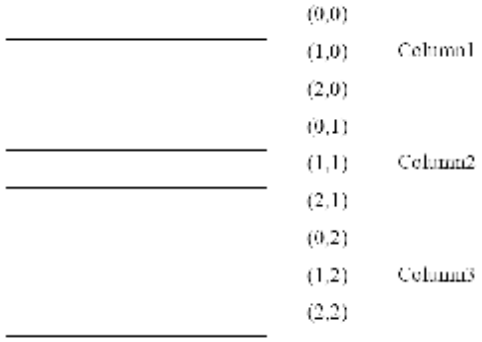
Location (T [4,6]) = BA + (7 x (4-1)) + (6-1)

= 900+ (7 x 3) +5

= 900+ 21+5

= 926

- Column Major Order**:** Order elements of first column stored linearly and then comes elements of next column.



To determine element address A[i,j]:

$$\text{Location}(A[i,j]) = \text{Base Address} + (M \times (j - 1)) + (i - 1)$$

For example:

Given an array [1…6,1…8] of integers. Calculate address element T[5,7], where BA=300

Sol) I = 5 , J = 7

M= 6 , N= 8

Location (T [4,6]) = BA + (6 x (7-1)) + (5-1)

= 300+ (6 x 6) +4

= 300+ 36+4

= 340

1.10.3 Representation of Three& Four Dimensional Array

By the same way we can determine address of element for three and four dimensional array:

Three Dimensional Array

To calculate address of element X[i,j,k] using row-major order :

$$\text{Location (X[i,j,k])} = BA + MN(k-1) + N(i-1) + (j-1)$$

using column-major order

$$\text{Location (X[i,j,k])} = BA + MN(k-1) + M(j-1) + (i-1)$$

Four Dimensional Array

To calculate address of element X[i,j,k] using row-major order :

$$\text{Location (Y[i,j,k,l])} = BA + MNR(l-1) + MN(k-1) + N(i-1) + (j-1)$$

using column-major order

$$\text{Location (Y[i,j,k,l])} = BA + MNR(l-1) + MN(k-1) + M(j-1) + (i-1)$$

For example:

Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6], by using rows &columns methods, if BA=900?

Sol) The dimensions of A are :

M=8 , N=5, R=7

i=5, j=3, k=6

Rows- wise

$$\text{Location (A[i,j,k])} = BA + MN(k-1) + N(i-1) + (j-1)$$

$$\text{Location (A[5,3,6])} = 900 + 8x5(6-1) + 5(5-1) + (3-1)$$

$$= 900 + 40 \times 5 + 5 \times 4 + 2$$

$$= 900 + 200 + 20 + 2$$

= 1122

Columns- wise

$$\text{Location } (A[i,j,k]) = BA + MN(k-1) + M(j-1) + (i-1)$$

$$\text{Location } (A[5,3,6]) = 900 + 8x5(6-1) + 8(3-1) + (5-1)$$

$$= 900 + 40 \times 5 + 8 \times 2 + 4$$

$$= 900 + 200 + 16 + 4$$

1.10.4 Operations on array

a) Traversing: means to visit all the elements of the array in an operation is called traversing.
b) Insertion: means to put values into an array
c) Deletion / Remove: to delete a value from an array.
d) Sorting: Re-arrangement of values in an array in a specific order (Ascending or Descending) is called sorting.
e) Searching: The process of finding the location of a particular element in an array is called searching.

a) Traversing in Linear Array:

It means processing or visiting each element in the array exactly once; Let **'A'** is an array stored in the computer's memory. If we want to display the contents of ***'A'***, it has to be traversed i.e. by accessing and processing each element of ***'A'*** exactly once.

Algorithm: (Traverse a Linear Array) Here **LA** is a Linear array with lower boundary **LB** and upper boundary **UB**. This algorithm traverses **LA** applying an operation Process to each element of **LA**.

1. [Initialize counter.] Set K=LB.
2. Repeat Steps 3 and 4 while K≤UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set k=K+1.
   [End of Step 2 loop.]
5. Exit.

The alternate algorithm for traversing (using for loop) is :

Algorithm: (Traverse a Linear Array) This algorithm traverse a linear array **LA** with lower bound **LB** and upper bound **UB**.

1. Repeat for K=LB to UB
   Apply PROCESS to LA[K].
   [End of loop].
2. Exit.

This program will traverse each element of the array to calculate the sum and then calculate & print the average of the following array of integers.
(4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

```
#include <iostream.h>
#define size 10 // another way int const size = 10
int main()
{ int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0,LB=0, UB=size;
float av;
for(i=LB,i<UB;i++) sum = sum + x[i];
av = (float)sum/size;
cout<< "The average of the numbers= "<<av<<endl;
return 0;
}
```

b) Sorting in Linear Array:

Sorting an array is the ordering the array elements in ***ascending*** (increasing from min to max) or ***descending*** (decreasing from max to min) order.

Bubble Sort:

The technique we use is called *"Bubble Sort"* because the bigger value gradually bubbles their way up to the top of array like air bubble rising in water, while the small values sink to the bottom of array. This technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

Pass = 1	Pass = 2	Pass = 3	Pass=4
2 1 5 7 4 3	**1 2** 5 4 3 7	**1 2** 4 3 5 7	**1 2** 3 4 5 7
1 **2 5** 7 4 3	1 **2 5** 4 3 7	1 **2 4** 3 5 7	1 **2 3** 4 5 7
1 2 **5 7** 4 3	1 2 **5 4** 3 7	1 2 **4 3** 5 7	1 2 3 4 5 7
1 2 5 **7 4** 3	1 2 4 **5 3** 7	1 2 3 4 5 7	
1 2 5 4 **7 3**	1 2 4 3 5 7		
1 2 5 4 3 7			

- Underlined pairs show the comparisons. For each pass there are size-1 comparisons.
- Total number of comparisons= $(size-1)^2$

Algorithm: (Bubble Sort) BUBBLE (DATA, N)
Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.

1. for pass=1 to N-1.
2. for (i=0; i<= N-Pass; i++)
3. If DATA[i]>DATA[i+1], then:
Interchange DATA[i] and DATA[i+1].
[End of If Structure.]
[End of inner loop.]
[End of Step 1 outer loop.]
4. Exit.

/* This program sorts the array elements in the ascending order using bubble sort method */

```
#include <iostream.h>
int const SIZE = 6
void BubbleSort(int [ ], int);
int main()
{
int a[SIZE]= {77,42,35,12,101,6};
int i;
cout<< "The elements of the array before sorting\n";
for (i=0; i<= SIZE-1; i++) cout<< a[i]<<", ";
BubbleSort(a, SIZE);
cout<< "\n\nThe elements of the array after sorting\n";
for (i=0; i<= SIZE-1; i++) cout<< a[i]<<", ";
return 0;
}
void BubbleSort(int A[ ], int N)
{
int i, pass, hold;
for (pass=1; pass<= N-1; pass++)
{
for (i=0; i<= SIZE-pass; i++)
{
if(A[i] >A[i+1])
{
hold =A[i];
A[i]=A[i+1];
A[i+1]=hold;
}
}
}
}
```

1.10.5 Applications of array

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.

Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists.

One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array.

1.10.6 Sparse matrix

Matrix with maximum zero entries is termed as sparse matrix. It can be represented as:

- Lower triangular matrix: It has non-zero entries on or below diagonal.
- Upper Triangular matrix: It has non-zero entries on or above diagonal.
- Tri-diagonal matrix: It has non-zero entries on diagonal and at the places immediately above or below diagonal.

1.11 STATIC AND DYNAMIC MEMORY ALLOCATION

In many programming environments memory allocation to variables can be of two types static memory allocation and dynamic memory allocation. Both differ on the basis of time when memory is allocated. In static memory allocation memory is allocated to variable at compile time whereas in dynamic memory allocation memory is allocated at the time of execution. Other