Arrays

Array is a structure which fulfils all these requirements:

It is a collection of homogeneous (of same data type) data items stored in consecutive memory locations and addressed by a common identifier.

Example - int marks [80].

Here - int is a data type

- Marks is an identifier name

- 80 is the size of the array (it must be integer constant).

1-Dimentional Array

Array is a structure which fulfils all these requirements:

It is a collection of homogeneous (of same data type) data items stored in consecutive memory locations and addressed by a common identifier.

• There are two variations in arrays:

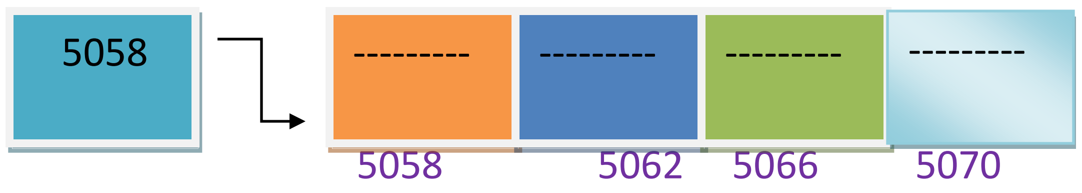
One dimensional arrays (1-D) and multidimensional arrays.

• 1-D arrays store one row of elements (store marks of a student in n subjects) and multidimensional arrays store two or more rows of elements.

1-D array Declaration:

// array of 4 uninitialized integers

- int myList[4];   // an array myLIST of size 4
- By the above declaration, myList has no values initialized for its elements.
- Such an array contains garbage values initially.
- It creates contiguous memory locations.



Memory Allocation of the array

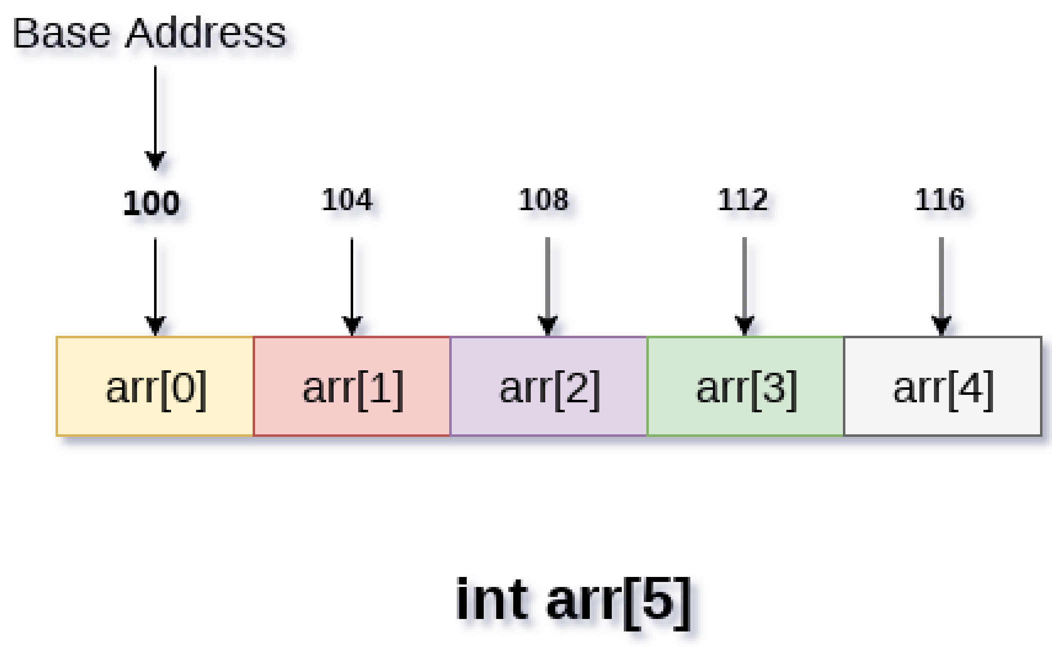
As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing): The first element of the array will be arr[0].

2. 1 (one - based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



Array Initialization:

• Initialization can be done in several ways.

Consider the following declaration int a[4]; // an integer array of size 4.

• Both declaration and initialization in single step.

Ex. int a[4] = {1, 2, 3, 4};

• Or by accessing a particular location

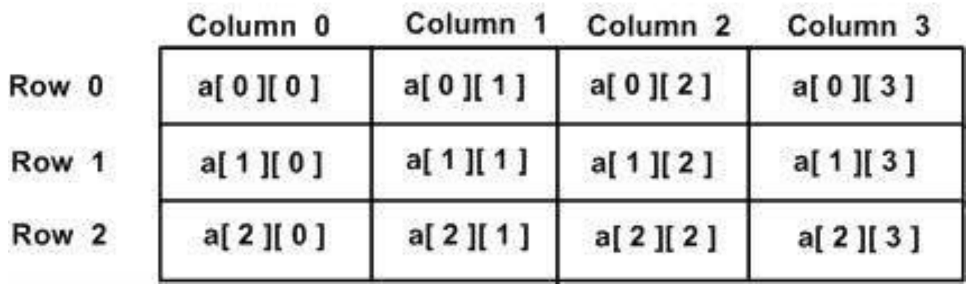
Ex. a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4;

• Or by the user

Ex. scanf("%d", &a[index]); ; // where index is a value in {0,1,2,3}.

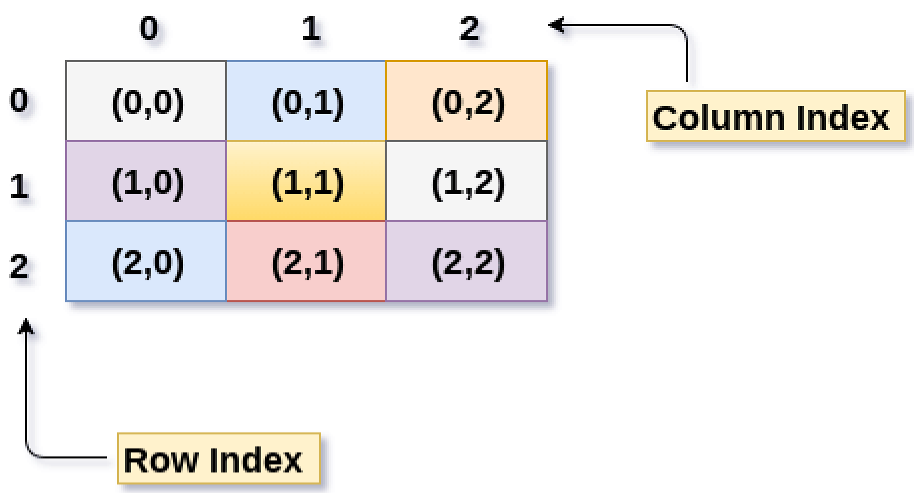
2-D ARRAY

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.



	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]

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.



How to initialize 2-D elements?

Initialization can be done in several ways:

Consider a declaration int a[2][2]; // an integer array of size 2 X 2.

- Both declaration and initialization in single step.

```
int a[2][2] = {1, 2, 3, 4};

int a[2][2] = {{1, 2}, { 3, 4}};

int a[][2] = {{1, 2}, { 3, 4}};

int a[][2] = {1, 2, 3, 4};
```

- Or by accessing a particular location

```
Ex. a[0][0] = 1;

      a[0][1] = 2;

     a[1][0] = 3;

    a[1][1] = 4;
```

- Or by the user

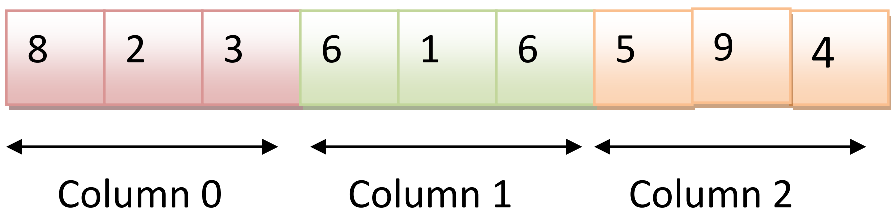
  Ex. scanf("%d", &a[i][j]); ; // where i is 0 or 1 and j is 0 or 1.

Column major order:

Column-major order is a similar method of flattening arrays onto linear memory, but the columns are listed in sequence.

For example, consider this 3×4 array:

- int A[3][3] = {{8, 6, 5}, {2, 1, 9,}, {3, 6, 4,}}



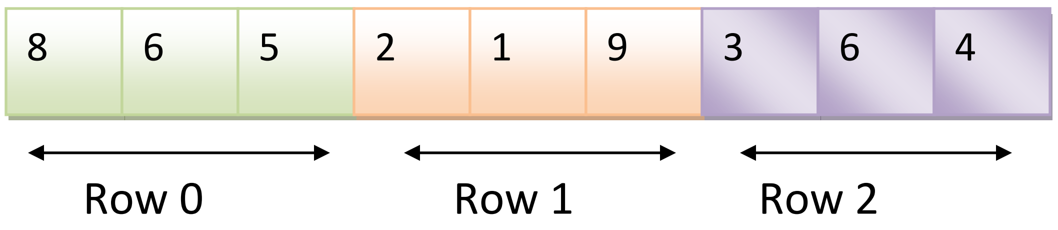
Row major order:

For example,

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

- We would find the array laid-out in linear memory as:

8 6 5 2 1 9 3 6 4



Address calculation in 1-D:

The address calculation of 1-D array is calculated in the following manner:

- Address of $A[K] = B + W * (K - LB)$ // LB = 0

Where,

B = Base address

W = Storage size of one element stored in the array (in byte)

K = Subscript of element whose address is to be found.

LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Address calculation in 2-D:

In row major skip the i-Lr rows and move to the j-Lc position

In row-major: $a[i][j] = (B + W * (n * (i - Lr) + j - Lc))$

In Column-major $a[i][j] = (B + W * (m * (j - Lc) + i - Lr))$

Where,

B = base address

W = storage size of one element stored in the array (in byte).

i and j are the row subscript and column subscript of the element whose address is to be found.

m is the number of rows and n is the number of columns.

Lr is the start row index of matrix

Lc is the start column index of matrix

Complexity of array operations:

Algorithm	Average case	Worst case
search	O(n)	O(n)
insert	O(n)	O(n)
delete	O(n)	O(n)

Linked lists:

- Limitations of arrays
  - Fixed size – The size of the array is static, so wastage of memory.
  - One block allocation – Not always possible.
  - Complex position based insertion/deletion – To insert an element we need to shift the existing elements.
- Solution - Linked List

What is a linked list?

- A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.
- Each node contains
  - **Data** will store the information part