UNIT 0- ARRAY

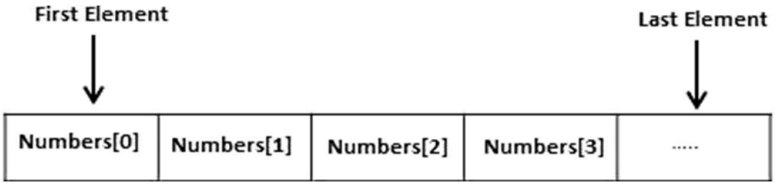
Definition of Array:

An array is a collection of homogeneous elements or n 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. What is important is that the quantities must be 'similar'. Each member in the group is referred to by its position in the group.

Similar elements of array could be all ints, or all float, 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.

Where an Ordinary variables are capable of holding only one value at a time.

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



Declaration of 1D Array:

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
datatype arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **marks** of type int, use this statement:

int marks[10];

Marks[0]	Marks[1	Marks[2]	Marks[3]	Marks[4]	Marks[5]	Marks[6]	Marks[7]	Marks[8]	Marks[9]

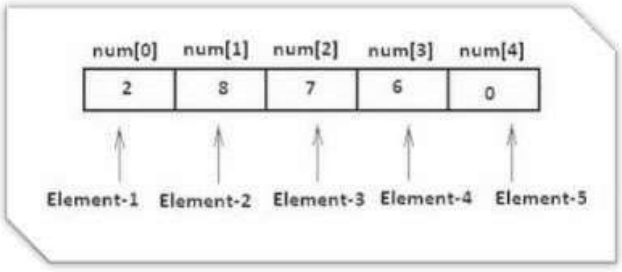
If an integer takes 4 bytes (32 bits) then size occupied by **marks** array will be 40 bytes.

Initialization of 1D array

- **Compile time initialization**

Fully initialization:

int num[5] = {2,8,7,6,0};



This is also correct:

int num[] = {2,8,7,6,0};

int num[5] = {0}; // all elements of array will initialize with zero

we can initialize an array Partially :

int num[5] = {2,8}; // remaining elements will be initialized with zero

- **Run time initialization**

Using scan function, we can store elements of array at run time.

Declaration of 2D Array:

int abc[5][4];

	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]

Initialization of 2D array:

int arr[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};//Storage take place in sequential order

```
int arr[2][4] = {
   {10, 11, 12, 13},
   {14, 15, 16, 17}
};
```

10	11	12	13
14	15	16	17

1	1	0
2	2	2
3	0	0

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

int abc[2][2] = {1, 2, 3 ,4 }  /* Valid */
int abc[ ][2] = {1, 2, 3 ,4 }   /* Valid , first dimension can be empty*/
int abc[ ][ ] = {1, 2, 3 ,4 }   /* Invalid declaration – you must specify second dimension*/
int abc[2][ ] = {1, 2, 3 ,4 } }   /* Invalid declaration – you must specify second dimension*/
```

What are Advantages and Disadvantages of arrays?

Advantages:

1. It is used to represent multiple data items of same type by using only single name.
2. It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
3. 2D arrays are used to represent matrices.

Disadvantages:

1. We must know in advance that how many elements are to be stored in array.
2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.
3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
4. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

Passing array to function

Just like variables, array can also be passed to a function as an argument. When we pass the address of an array while calling a function then this is called function call by reference

Passing 1D array to function

```
void modify(int a[3]);
int main()
{
        int arr[3] = {1,2,3};
        modify(arr);// passing address of array
        for(i=0;i<3;i++)
          printf("%d",arr[i]);
        getch();
        return 0;
}

void modify(int a[3])
{
int i;
for(i=0;i<3;i++)
  a[i] = a[i]*a[i];
}
```

Passing 2D array to function

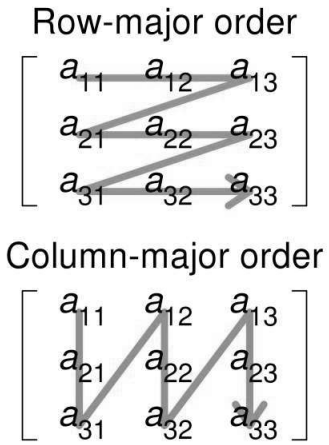
```
void print(int m, int n, int arr[3][3]);
int main()
 {
        int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
        int m = 3, n = 3;
        print(m, n, arr); // passing address of array
        return 0;
 }

void print(int m, int n, int arr[3][3])
{
   int i, j;
   for (i = 0; i < m; i++)
     for (j = 0; j < n; j++)
       printf("%d ", arr[i][j]);
}
```

Row-major vs. column-major

By far the two most common memory layouts for multi-dimensional array data are *row-major* and *column-major*.

When working with 2D arrays (matrices), row-major vs. column-major are easy to describe. The row-major layout of a matrix puts the first row in contiguous memory, then the second row right after it, then the third, and so on. Column-major layout puts the first column in contiguous memory, then the second, etc



Row-major order is used in C/C++/Objective-C (for C-style arrays), PL/I,Pascal, Speakeasy[citation needed], SAS, and Rasdaman.

Column-major order is used in Fortran, MATLAB, GNU Octave, S-Plus, R, Julia, and Scilab.

Formula of finding the location (address) of particular element in 2D array using example

1. In case of Column Major Order:

The formula is:

LOC (A [J, K]) = Base (A) + w [M (K-1) + (J-1)]

Here

LOC (A [J, K])	:	is the location of the element in the Jth row and Kth column.
Base (A)	:	is the base address of the array A.
w	:	is the number of bytes required to store single element of the array A.
M	:	is the total number of rows in the array.
J	:	is the row number of the element.
K	:	is the column number of the element.

E.g.

A 3 x 4 integer array A is as below:

	Subscript	Elements	Address
10	(1,1)	1000	
20	(2,1)	1002	
50	(3,1)	1004	
60	(1,2)	1006	

90	(2,2)	1008
40	(3,2)	1010
30	(1,3)	1012
80	(2,3)	1014
75	(3,3)	1016
55	(1,4)	1018
65	(2,4)	1020

Suppose we have to find the location of A [3, 2]. The required values are:

Base (A) : 1000

w : 2 (because an integer takes 2 bytes in memory)

M : 3

J : 3

K : 2

Now put these values in the given formula as below:

LOC (A [3, 2]) = 1000 + 2 [3 (2-1) + (3-1)]

= 1000 + 2 [3 (1) + 2]

79	(3,4)	1022

= 1000 + 2 [3 + 2]

= 1000 + 2 [5]

= 1000 + 10

= 1010

2. In case of Row Major Order:

The formula is:

LOC (A [J, K]) = Base (A) + w [N (J-1) + (K-1)]

Here

LOC (A [J, K]) : is the location of the element in the Jth row and Kth column.

Base (A) : is the base address of the array A.

w : is the number of bytes required to store single element of the array A.

N : is the total number of columns in the array.

J : is the row number of the element.

K : is the column number of the element.

E.g.

A 3 x 4 integer array A is as below:

	Subscript	Elements	Address
10	(1,1)	1000	
60	(1,2)	1002	
30	(1,3)	1004	
55	(1,4)	1006	

20	(2,1)	1008
90	(2,2)	1010
80	(2,3)	1012
65	(2,4)	1014
50	(3,1)	1016
40	(3,2)	1018
75	(3,3)	1020
79	(3,4)	1022

Suppose we have to find the location of A [3, 2]. The required values are:

Base (A) : 1000

w : 2 (because an integer takes 2 bytes in memory)

N : 4

J : 3

K : 2

Now put these values in the given formula as below:

LOC (A [3, 2]) = 1000 + 2 [4 (3-1) + (2-1)]

= 1000 + 2 [4 (2) + 1]

= 1000 + 2 [8 + 1]

= 1000 + 2 [9]

= 1000 + 18

= 1018

Array Bounds Checking in C

Most languages (Java, C#, Python, Javascript) prevent the programmer from going past the end of an array. This process, performed at runtime by the language implementation, is frequently called *Bounds Checking.*

What if programmer accidentally accesses any index of array which is out of bound?

Unfortunately in C there is no way for the programmer to determine the size of an array at runtime and so it becomes necessary for the programmer to keep track of the length of the array.

Array Bounds checking in not available in C language. C don't provide any specification which deal with problem of accessing invalid index.

As per ISO C standard it is called **Undefined Behavior**.
An undefined behavior (UB) is a result of executing computer code whose behavior is not prescribed by the language specification to which the code can adhere to, for the current state of the program (e.g. memory). This generally happens when the translator of the source code makes certain assumptions, but these assumptions are not satisfied during execution.

Examples of Undefined Behavior while accessing array out of bounds

1. **Access non allocated location of memory:** The program can access some piece of memory which is owned by it.

```
// Program to demonstrate // accessing array out of bounds
#include <stdio.h>
int main()
```