DATA STRUCTURE AND ALGORITHM

Unit I:

Introduction: Data structures – Types of Data structures –Data structure operations – Abstract data type- Analysis of algorithms – Amortized Analysis.

Arrays: Introduction – Characteristics of Arrays – One-dimensional Arrays – Operation with Arrays – Two-dimensional Arrays – Multi-dimensional Arrays.

INTRODUCTION

Data Structure

- A data structure is a named location that can be used to store and organize data
- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.
- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
- A data structure has also defined an instance of ADT.ADT means ABSTRACT DATA TYPE. It is formally defined as a triplet.[D,F,A]
- D: Set of the domain.
- F: the set of operations.
- A: the set of axioms

Definition:

- Data Structures is the concept of set of algorithms used to structure the information.
- These algorithms are implemented using C, C++, Java, etc
- Structure the information means store and process data in an efficient manner.

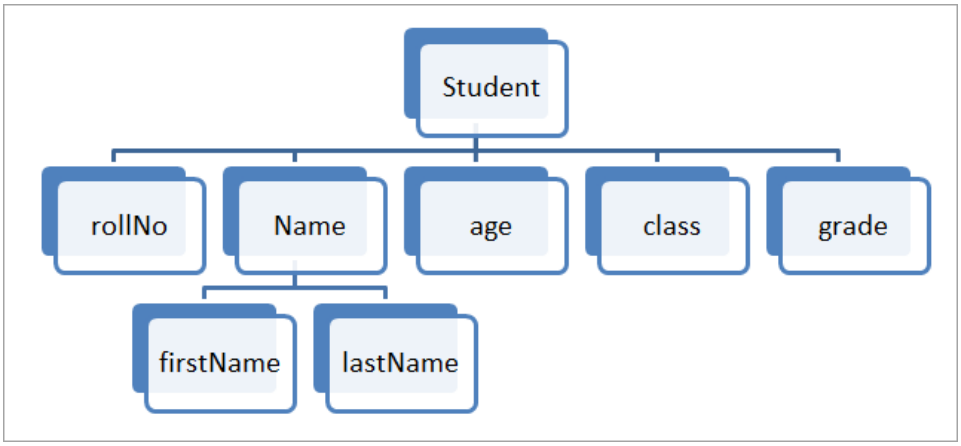
To store and process data we may use the following operations

1. create() .
2. sorting()
3. insert()
4. merging()

5. delete()
6. splitting()
7. display()
8. traversal()
9. searching()

Basic Terminology

Data structures are the building blocks of any program or the software.



- **Data:** It is the elementary value. In the above figure, student Roll No. can be data.
- **Group item:** This is the data item that has more than one sub-items. In the above figure, Student_name has First Name and Last Name.
- **Record:** It is a collection of data items. In the above example, data items like student Roll No., Name, Class, Age, Grade, etc. form a record together.
- **Entity:** It is a class of records. In the above diagram, the student is an entity.
- **Attribute or field:** Properties of an entity are called attributes and each field represents an attribute.
- **File:** A file is a collection of records. In the above example, a student entity can have thousands of records. Thus a file will contain all these records.

Need of Data Structures

Data structures provide an easy way of organizing, retrieving, managing, and storing data.

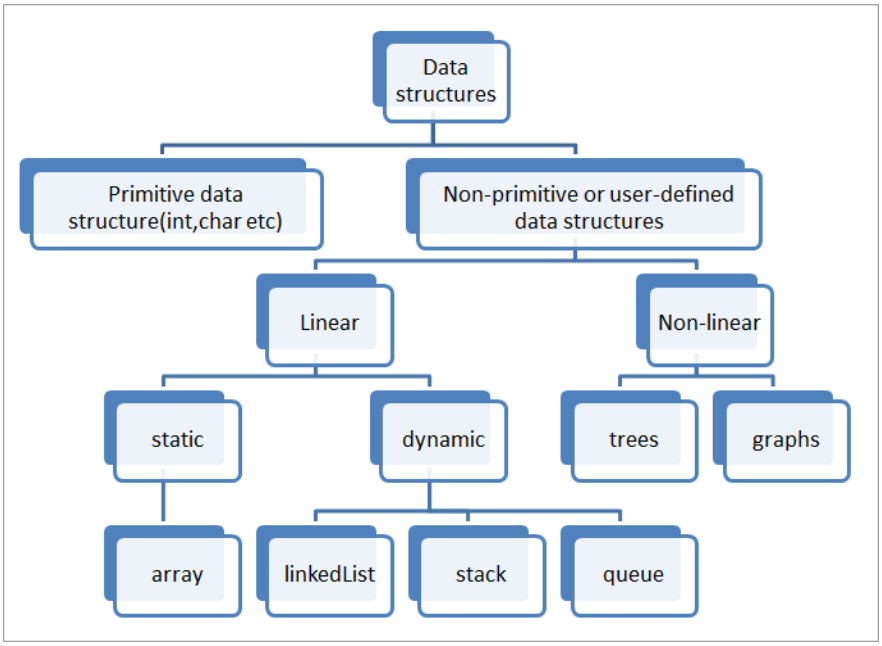
Here is a list of the needs for data.

1. Data structure modification is easy.
2. It requires less time.
3. Save storage memory space.
4. Data representation is easy.
5. Easy access to the large database.

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures.
Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place.
Abstraction: Data structure is specified by the ADT which provides a level of abstraction.

TYPES OF DATA STRUCTURE



Type of data structure :

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure:

- Elements are arranged in one dimension ,also known as linear dimension.

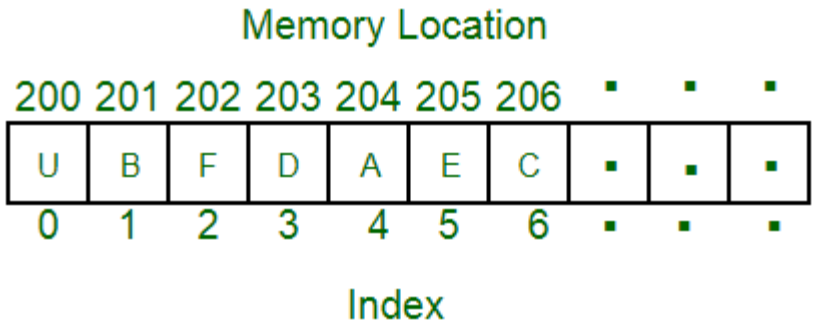
- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
  Example : Array
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

- Example: lists, stack, queue, etc.

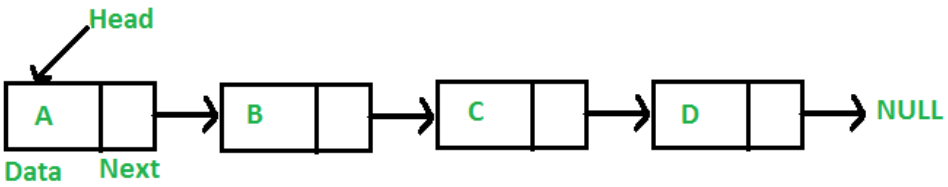
Non-Linear Data Structure

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.

1. Array: An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

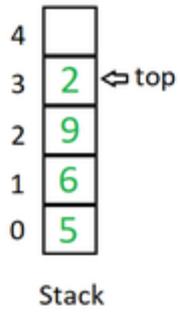


2. Linked Lists: Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



3.<u>Stack</u>: Stack is a linear data structure which follows a particular order in which the operations are performed.

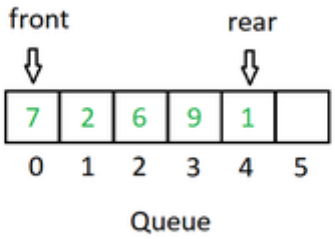
The order may be LIFO(Last In First Out) or FILO(First In Last Out). In stack, all insertion and deletion are permitted at only one end of the list.



Mainly the following three basic operations are performed in the stack:

- **Initialize**: Make a stack empty.
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.

4. <u>Queue</u>: Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



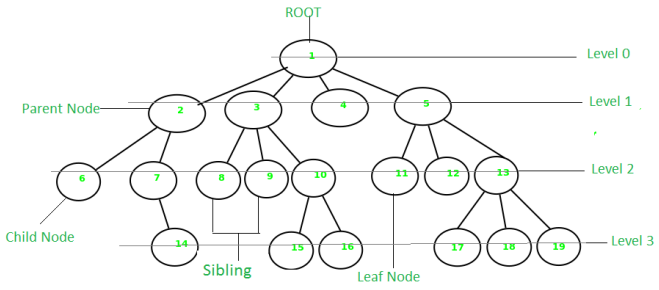
Mainly the following four basic operations are performed on queue:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.

Tree:
A tree is a non-linear and hierarchal data structure where the elements are arranged in a tree-like structure. In a tree, the topmost node is called the root node. Each node contains some data, and data can be of any type. It consists of a central node, structural nodes, and sub-nodes which are connected via edges. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure. A tree has various terminologies like Node, Root, Edge, Height of a tree, Degree of a tree, etc.

There are different types of Tree like

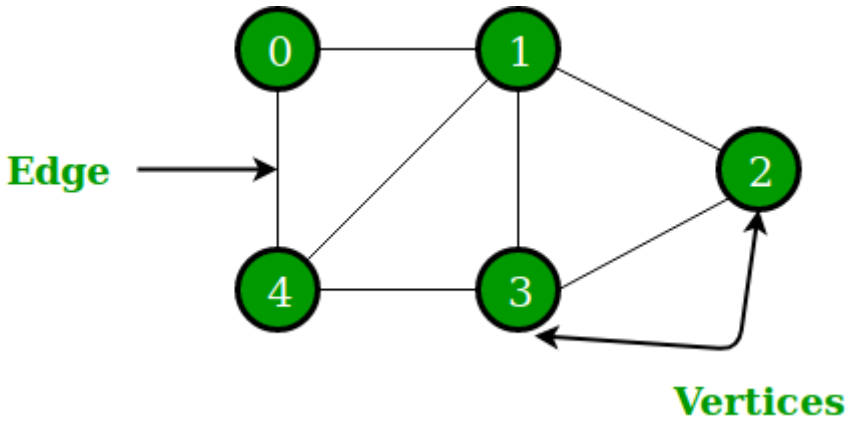
- Binary Tree,
- Binary Search Tree,
- AVL Tree,
- B-Tree, etc.



Tree

Graph:

A graph is a non-linear data structure that consists of vertices (or nodes) and edges. It consists of a finite set of vertices and set of edges that connect a pair of nodes. Graph is used to solve the most challenging and complex programming problems. It has different terminologies which are Path, Degree, Adjacent vertices, Connected components, etc.



OPERATIONS ON DATA STRUCTURE

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

DS Algorithm

What is an Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- Input:
- Unambiguity:
- Finiteness:
- Effectiveness:
- Language independent:

Dataflow of an Algorithm

- Problem:
- Algorithm:
- Input:
- Processing unit:
- Output:

Why do we need Algorithms?

We need algorithms because of the following reasons:

- Scalability
- Performance:

Factors of an Algorithm

The following are the factors that we need to consider for designing an algorithm:

- Modularity:
- Correctness:
- Maintainability:
- Functionality:
- Robustness:
- User-friendly:
- Simplicity:

- Extensibility:

Importance of Algorithms

1. Theoretical importance:
2. Practical importance:

Issues of Algorithms

The following are the issues that come while designing an algorithm:

- How to design algorithm
- How to analyze algorithm efficiency

Approaches of Algorithm

- Brute force algorithm
    1. Optimizing
    2. Sacrificing
- Divide and conquer
- Greedy algorithm
- Dynamic programming
- Branch and Bound Algorithm
- Randomized Algorithm
- Backtracking
    - Sort
    - Search
    - Delete
    - Insert
    - Update

ANALYSIS OF ALGORITHM

it generally focused on CPU (time) usage, Memory usage, Disk usage, and Network usage. All are important, but the most concern is about the CPU time. Be careful to differentiate between:

- **Performance:** How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
- **Complexity:** How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Algorithm Analysis:

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

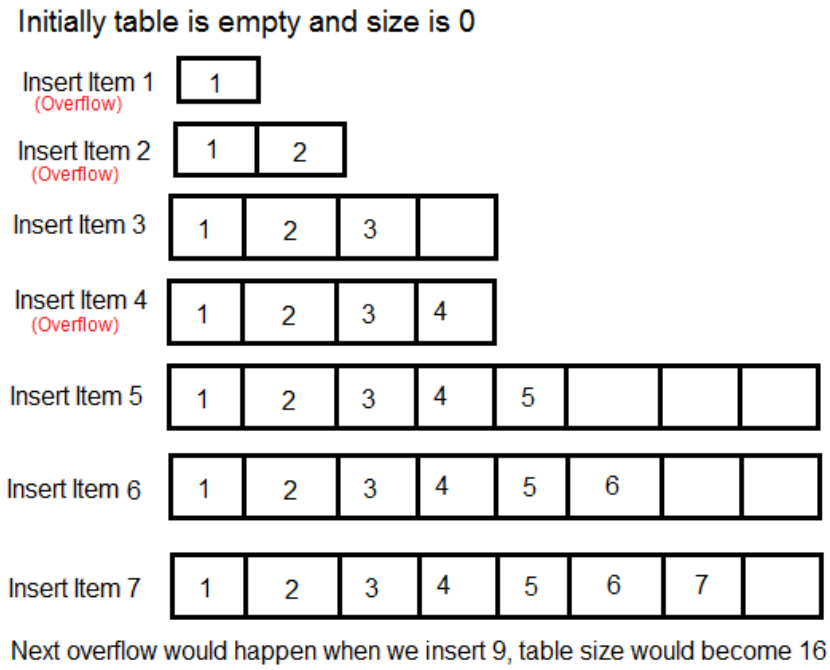
- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.And then we divide it by the total number of inputs.

Average case = all random case time / total no of case

AMORTIZED ANALYSIS

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.

Let us consider an example of simple hash table insertions. How do we decide on table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but the space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase the size of the table whenever it becomes full. Following are the steps to follow when the table becomes full.

1) Allocate memory for larger table size, typically twice the old table.
2) Copy the contents of the old table to a new table.
3) Free the old table.

If the table has space available, we simply insert a new item in the available space.

What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst-case cost of insertion is O(n). Therefore, the worst-case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $\Theta(n)$ time.

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + ...)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

ARRAY

INTRODUCTION

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

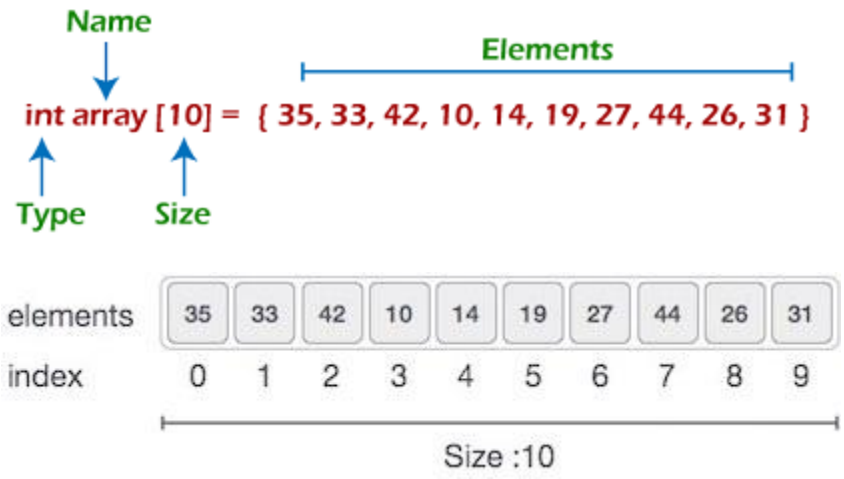
CHARACTERISTICS

The characteristics of arrays are as follows –

- An array is always stored in consecutive memory location.
- It can store multiple value of similar type, which can be referred with single name.
- The pointer points to the first location of memory block, which is allocated to the array name.
- An array can either be an integer, character, or float data type that can be initialised only during the declaration.
- The particular element of an array can be modified separately without changing the other elements.
- All elements of an array can be distinguishing with the help of index number.

REPRESENTATION OF AN ARRAY

ONE dimensional array



As per the above illustration, there are some of the following important points -

- Index starts with 0.

- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

OPERATIONS WITH AN ARRAY

The operations of an array include –

- **Searching** – It is used to find whether particular element is present or not.
- **Sorting** – Helps in arranging the elements in an array either in an ascending or descending order.
- **Traversing** – Processing every element in an array, sequentially.
- **Inserting** – Helps in inserting elements in an array.
- **Deleting** – helps in deleting the element in an array.

Example Program

Following is the C program for **searching an element in an array**

```
#include<iostream.h>
#include <stdio.h>
void main()
{
  int LA[ ] = {1,3,5,7,8};
  int item = 5, n = 5;
  int i = 0, j = 0;
  Cout("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    Cout("LA[%d] = %d \n", i, LA[i]);
  }

  while( j < n)
{
    if( LA[j] == item )
{
      break;
    }

    j = j + 1;
  }
```

```
  cout("Found element %d at position %d\n", item, j+1);
}
```

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

MULTI DIMENSIONAL ARRAY

The ***general form of declaring N-dimensional arrays*** is:

data_type array_name[size1][size2]....[sizeN];

- **data_type**: Type of data to be stored in the array.
- **array_name**: Name of the array
- **size1, size2,... ,sizeN**: Sizes of the dimension

Examples:

Two dimensional array: int two_d[10][20];

Three dimensional array: int three_d[10][20][30];

Size of Multidimensional Arrays:

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

- The array **int x[10][20]** can store total (10*20) = 200 elements.
- Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one-dimensional array for easier understanding.

The basic form of declaring a two-dimensional array of size x, y:

Syntax:

data_type array_name[x][y];

Here, **data_type** is the type of data to be stored.

We can declare a two-dimensional integer array say 'x' of size 10,20 as:

int x[10][20];

Elements in two-dimensional arrays are commonly referred to by x[i][j] where i is the row number and 'j' is the column number.

A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and the column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns 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]

Initializing Two – Dimensional Arrays: There are various ways in which a Two-Dimensional array can be initialized.

First Method:

int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.

Accessing Elements of Two-Dimensional Arrays: Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

int x[2][1];

The above example represents the element present in the third row and second column.

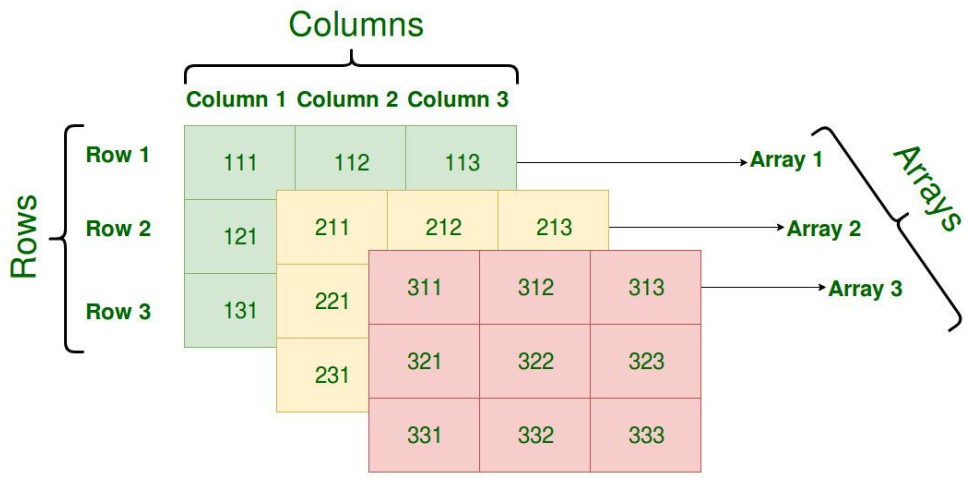
Example:

```
#include<iostream>
using namespace std;
 int main()
{
   // an array with 3 rows and 2 columns.
   int x[3][2] = {{0,1}, {2,3}, {4,5}};

   // output each array element's value
   for (int i = 0; i < 3; i++)
   {
      for (int j = 0; j < 2; j++)
      {
         cout << "Element at x[" << i
             << "][" << j << "]: ";
         cout << x[i][j]<<endl;
      }
   }

   return 0;
}
```

Output:

```
Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5
```

Three-Dimensional Array



Initializing Three-Dimensional Array:

Initialization in a Three-Dimensional array is the same as that of Two-dimensional arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18, 19,
                  20, 21, 22, 23};
```

Accessing elements in Three-Dimensional Arrays: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

```
// C++ program to print elements of Three-Dimensional
// Array
#include <iostream>
using namespace std;

int main()
{
   // initializing the 3-dimensional array
   int x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },
                      { { 6, 7 }, { 8, 9 }, { 10, 11 } } };
```

```
    // output each element's value
   for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 3; ++j) {
         for (int k = 0; k < 2; ++k) {
            cout << "Element at x[" << i << "][" << j
               << "][" << k << "] = " << x[i][j][k]
               << endl;
         }
      }
   }
   return 0;
}
```

Output:

```
Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7
Element at x[1][1][0] = 8
Element at x[1][1][1] = 9
Element at x[1][2][0] = 10
Element at x[1][2][1] = 11
```