

Programming with Data Structure



Dr. Dharm Raj Singh

Assistant Professor, (HOD)

Department of Computer Application

Jagatpur P. G. College, Varanasi

Mobile No. 9452070368, 7275887513

Email- dharmrajsingh67@yahoo.com

Outline

1. MODULE 1

- Unit 1 : Array
- Unit 2 : pointer
- Unit 3 : Stacks and Queues

2. MODULE 2

- unit 1 : linked lists
- unit 2 : Trees
- unit 3 : B-Trees

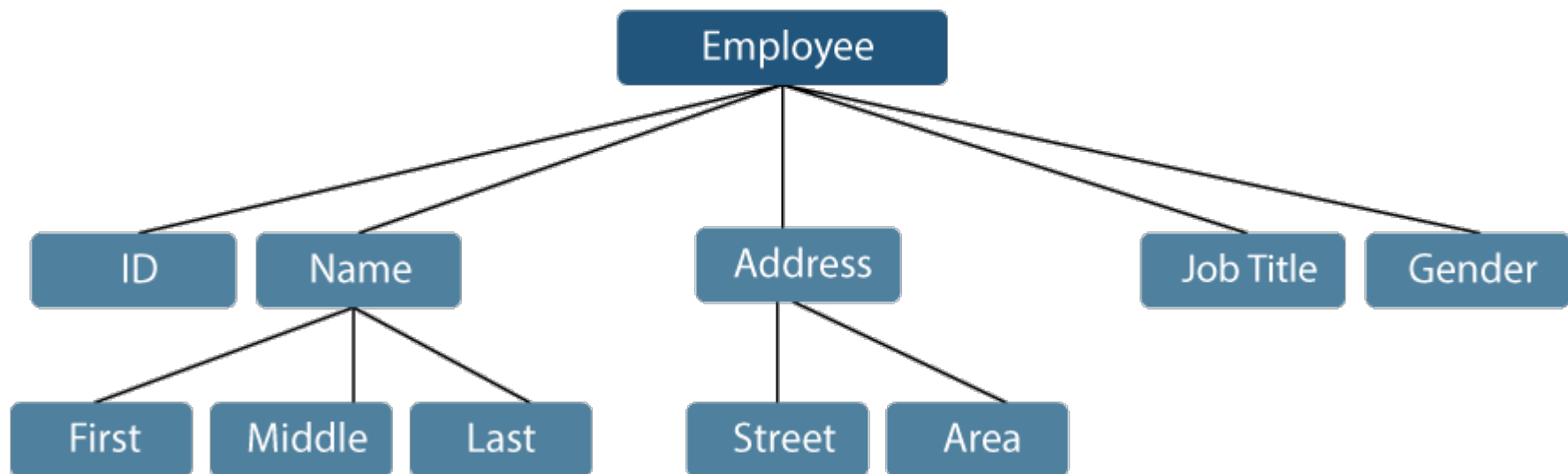
3. MODULE 3

- Unit 1 : Sorting Techniques
- Unit 2 : Searching Techniques

Basic terminology

Data: data are simply **values** or **sets of values**. A data item refers to a single unit of values. **Data** is a collection of **facts** and **figures**.

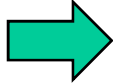
- The data items are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.
- Let us consider an example where an employee name can be broken down into three sub-items: First, Middle, and Last. However, an ID assigned to an employee will generally be considered a single item.
- In the example data items are ID, Age, Gender, First, Middle, Last, Street, Locality, etc., are elementary data items. In contrast, the Name and the Address are **group data items**.



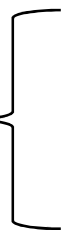
Basic terminology

Information: information is the manipulation of data which have meaningful or **processed data**.

Entity : An entity is something that has certain attributes or properties which may be assigned values. For example employee of a given organization, students.

Attributes or Field: 

Name	Age	Sex	Emp_ID
Rohit	35	M	101
Mohit	32	M	102
Ramesh	40	M	103

Values or Records: 

Entity Set: Entities with similar attributes form an entity set. e.g. all the employees in an organization.

Field: A single elementary unit of information symbolizing the Attribute of an Entity is known as Field.

Record: record is a collection of related information. A record is the collection of field values of a given entity.

File: a file is the collection of related records of the entities in a given entity set.

Database: database is a collection of related file.

Introduction to Data Structures

- A data *type* is a well-defined collection of data with a well-defined set of operations on it.
- A data *structure* is an actual implementation of a particular abstract data type.
- Data structure is the structural representation of **logical relationships** between elements of data.
- Data structure is a representation of the **logical relationship** existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a **mathematical or logical model** of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure**.
- The storage structure representation in auxiliary memory is called as **file structure**.

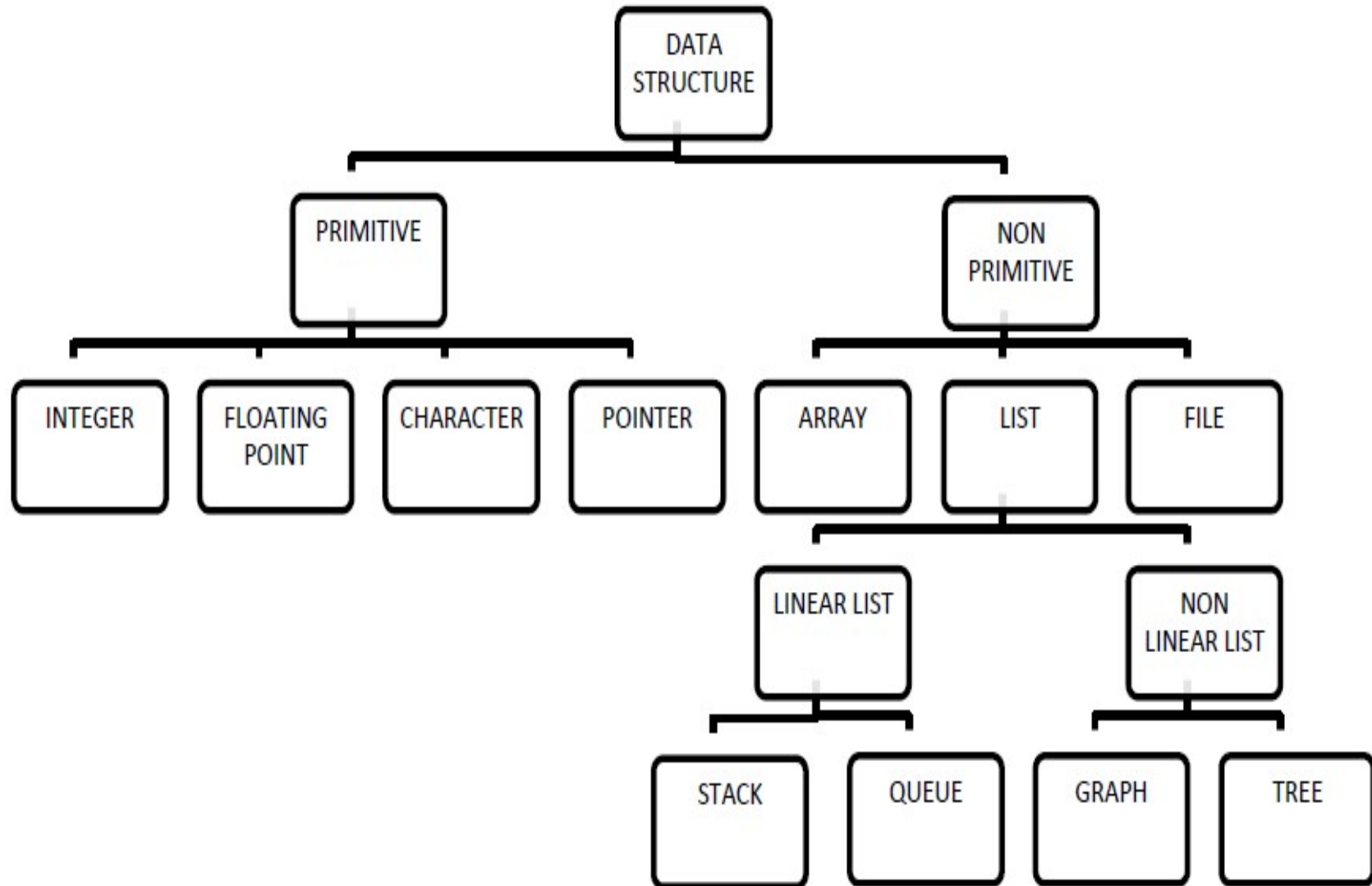
Data Structure mainly specifies the following four things

- ❑ Organization of Data
- ❑ Accessing methods
- ❑ Degree of associativity
- ❑ Processing alternatives for information
- ❖ Algorithm + Data Structure = Program
- ❖ Data structure study covers the following points
 - Amount of memory require to store.
 - Amount of time require to process.
 - Representation of data in memory.
 - Operations performed on that data.

Data organization

- Organization of data refers to **classifying and organizing** data to make it more meaningful and usable.
- The collection of data you work with in a program have some kind of structure or organization of data in Data Structures. No matter how complex your data structures are they can be broken down into two fundamental types.
 1. **Contiguous**
 2. **Non-Contiguous**
- In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An **array** is an example of a contiguous structure.
- In contrast, items in a non-contiguous structure are scattered in memory, but are linked to each other in some way. A **linked list** is an example of a non-contiguous data structure.

Classification of Data Structure



Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- The storage structure of these data structures may vary from one machine to another.
- Integers, floats, character and pointers are examples of primitive data structures.

Non primitive Data Type

1. Non-Primitive Data Structures are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either homogeneous or heterogeneous data types. Examples of Array, File, strings, Unions, linked lists, stacks and queues etc. we can divide these data structures into two sub-categories -
 - **Linear Data Structures**
 - **Non-Linear Data Structures**

Linear data structures

- A data structure is said to be linear if and only if there is a adjacency relationship between the elements. or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
 - **Static memory allocation**
 - **Dynamic memory allocation**
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear data structure are arrays, linked list, Stack and Queue etc.

Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- A data structure in which insertion and deletion is not possible in a linear fashion and Elements are stored based on the hierarchical relationship among the data.
- Examples of Non-linear Data Structure are Tree and Graph.

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.

- Trees represent the hierarchical relationship between various elements.
- Tree consist of nodes connected by edge, the node represented by circle and edge represented by line.

Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

Operation on Data Structures

- Some of the common operations on Non-Primitive Data Structure are:
 1. **Create:** The create operation results in *reserving memory for program elements*. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time.
 2. **Destroy / delete:** Destroy operation destroys memory space allocated for specified data structure. `free()` function of C language is used to destroy data structure.
 - It is the process of removing an item from the structure.
 3. **Selection:** Selection operation deals with accessing a particular data within a data structure.
 4. **Updation:** It updates or ***modifies the data in the data structure***.
 5. **Traversal:** Traversal is a process of **visiting each and every node of a list** in systematic manner.

6. **Searching:** It is the process of finding the location of the element with a given key value in a particular data structure or **finding the location of an element**, which satisfies the given condition.
7. **Sorting:** It is the process of arranging the elements of a particular data structure in some form of logical order. The order may be *either ascending or descending or alphabetic order depending on the data items present*.
8. **Merging:** Merging is a process of *combining the data items of two different sorted list into a single sorted list*.
9. **Insertion:** It is *the process of adding a new element to the structure*. Most of the times this operation is performed by identifying the position where the new element is to be inserted.
10. **Splitting:** Splitting is a *process of partitioning single list to multiple list*.

ALGORITHM

- A set of ordered steps or procedures necessary to solve a problem.
- An algorithm is a sequence of unambiguous instructions for solving a problem.
- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:
 1. **Input:** Zero or more quantities are externally supplied.
 2. **Output:** At least one quantity is produced.
 3. **Definiteness:** Each instruction is clear and unambiguous.
 4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
 5. **Effectiveness:** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Basic Statements Used and Examples

- Algorithm always begins with the word '**Start**' and ends with the word '**Stop**'.
- Step wise solution is written in distinguished steps. This is as shown in

example

Start

Step 1:

Step 2:

▪

▪

▪

Step n:

End

➤ **Input Statement:** Algorithm takes one or more inputs to process.
The statements used to indicate the input is Read a or Input b. **for example**

Let a , b be the names of the Input

Input a or Read a

Input b or Read b

Where a and b are variable names.

➤ **Output Statements:** Algorithm produces one or more outputs.
The statement used to show the output is output b or print b.

Syntax: Output variable name

Print variable name

For example output a or print a

output b or print b

where a and b are variable names.

➤ **Assignment Statements:**

Processing can be done using the assignment statement.

i.e. L.H.S = R.H.S

On the L.H.S is a variable.

While on the R.H.S is a variable or a constant or an expression.

- The value of the variable, constant or the expression on the R.H.S is assigned in L.H.S.

➤ **The L.H.S and R.H.S should be of the same type.** Here ' = ' is called assignment operator.

For example Let the variables be x, y. The product be z this can be represented by as

Read x, y

Z = x * y

➤ Order in which the steps of an algorithm are executed is divided into 3 types namely

- i) Sequential Order
- ii) Conditional Order
- iii) Iterative Order

➤ Sequential Order

Each step is performed in serial fashion i.e. in a step by step procedure for example

Write an algorithm to add two numbers.

Step 1 : Start

Step 2 : Read a

Step 3 : Read b

Step 4 : Add a , b

Step 5 : Store in d

Step 6 : Print d

Step 7 : End

Conditional Order

Based on fact that the given condition is met or not the algorithm selects the next step to do. If statements are used when decision has to be made. Different format of if statements are available they are

a) Syntax : **if** (condition)

Then {set of statements S1}

- Here condition means Boolean expressions which is **TRUE** or **FALSE**.
- If condition is **TRUE** then the statements S1 is evaluated.
- If **FALSE** S1 is not evaluated Programme skips that section.
- For example

Write an algorithm to check equality of numbers.

Step 1 : Start

Step 2 : Read a, b

Step 3 : if $a == b$, print numbers are equal to each other

Step 4 : End

b) if else (condition) statement:

if (condition)

Then {set of statements S1}

else

Then {set of statements S2}

Here if condition evaluates to **true** then S1 is executed otherwise else statements are executed. For example Write an algorithm to print the grade.

Step 1 : Start

Step 2 : Read marks

Step 3 : if marks greater than 60 is TRUE

print 'GRADE A'

Step 4 : Other wise

print 'GRADE B'

Step 5 : End

iii) Iterative Order

Here algorithm repeats the finite number of steps over and over till the condition is not meet. Iterative operation is also called as looping operation. For example

Add 'n' natural numbers till the sum is 5.

Step 1 : Start

Step 2 : set count to 0 and $i=1$

Step 3 : add i to count

Step 4 : $i=i+1$

Step 5 : if count is less than 5, then repeat steps 3 & 4

Step 6 : otherwise print count

Step 7 : End

- The range of inputs for which an algorithm works has to be specified carefully.
 - The same algorithm can be represented in several different ways.
 - Several algorithms for solving the same problem may exist.
 - Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.
- Every problem as we understand can be solved using different methods or techniques.
- Thus each method may be represented using an algorithm.
- The important question to answer is How to choose the best algorithm?

- Once we develop an algorithm, it is always better to check whether the algorithm is efficient or not. The efficiency of an algorithm depends on the following factors:
 1. **Accuracy of the output**
 2. **Robustness of the algorithm**
 3. **User friendliness of the algorithm**
 4. **Time required to run the algorithm**
 5. **Space required to run the algorithm**
- The main aim of using a computer is to transform data from one form to another. The *algorithm describes the process of transforming data*
- *Efficiency of algorithms depends upon the data structures that are selected for data representation.*
- The data structure has to be finally represented in the memory. This is called as memory representation of data structures. While selecting the memory representation of data structures it should worth memory space and it should also be easy to access.

COMPLEXITY OF ALGORITHM

- Every algorithm we write should be analyzed before it is implemented as a program.
- There are two main criteria's or reasons upon which we can judge an algorithm. They are:
 - 1. *The correctness of the algorithm and***
 - 2. *The simplicity of the algorithm***
- The correctness of an algorithm can be analyzed by tracing the algorithm with certain sample data and by trying to answer certain questions such as.
 1. Does the algorithm do what we want it to do?
 2. Does it work correctly according to the original specifications of the task?
 3. Does the algorithm work when the data structure used is full?
 4. is there documentation that describes how to use it and how it works?

COMPLEXITY OF ALGORITHM

- In order to analyze this we will have to consider the time requirements and the space requirements of the algorithm.
- These are the two parameters on which the efficiency of the algorithms is measured.
- Space requirements are not a major problem today because memory is very cheap. So time is the only criteria for *measuring efficiency of the algorithm as we have to maximize the utilization of the CPU and response time to be minimized to be faster.*

Space complexity: The Space complexity of an algorithm is *the amount of main memory needed to run the program till completion*. Consider the following algorithms for exchange two numbers:

Algo1_exchange (a, b)

Step 1: tmp = a;

Step 2: a = b;

Step 3: b = tmp;

Algo2_exchange (a, b)

Step 1: a = a + b;

Step 2: b = a - b;

Step 3: a = a - b;

- The first algorithm uses three variables a, b and tmp and the second one take only two variables, so if we look from the space complexity perspective the second algorithm is better than the first one.

TIME COMPLEXITY

- The Time complexity of an algorithm is *the amount of computer time it needs to run the program till completion*.
- To measure the time complexity in absolute time unit has the following problems.
 1. The time required for an algorithm depends on number of instructions executed by the algorithm.
 2. The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.
 3. Different types of instructions take different amount of time on same computer.

- **Consider another algorithm for add n even number**

Statement	Per execution Cost	Frequency	Total cost
Step 1. $i = 2;$	1	1	1
Step 2. $sum = 0;$	1	1	1
Step 3. while $i \leq 2*n$	1	$n+1$	$n+1$
Step 4. $sum = sum + i$	1	n	n
Step 5. $i = i + 2;$	1	n	n
Step 6. end while;	0	1	0
Step 7. return sum;	1	1	1
Total cost			$3n+4$

- Consider another algorithm:

Statement	Per execution Cost	Frequency	Total cost
Step 1. sum(a, n)	0	-	0
Step 2. {	0	-	0
Step 3. s=0.0;	1	1	1
Step 4. for i=1 to n do	1	n+1	n+1
Step 5. s=s+a[i];	1	n	n
Step 6. return s;	1	1	1
Step 7. }	0	-	0
Total cost			2n+3

How to calculate $f(n)$?

- Calculating the value of $f(n)$ for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their $f(n)$ values. We will find the growth rate of $f(n)$ because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes.
- Now, how to find $f(n)$.
- Let's look at a simple example.
- $f(n) = 5n^2 + 6n + 12$
- where n is the number of instructions executed, and it depends on the size of the input.
- When $n=1$
- % of running time due to $5n^2 = \frac{5}{12} * 100 = 41.67\%$
- % of running time due to $6n = \frac{6}{12} * 100 = 50.00\%$
- % of running time due to $12 = \frac{12}{12} * 100 = 100.00\%$

From the above calculation, it is observed that most of the time is taken by 12. But, we have to find the growth rate of $f(n)$, we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of $f(n)$.

n	$5n^2$	$6n$	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

- As we can observe in the above table that with the increase in the value of n , the running time of $5n^2$ increases while the running time of $6n$ and 12 also decreases. Therefore, it is observed that for larger values of n , the squared term consumes almost 99% of the time. As the n^2 term is contributing most of the time, so we can eliminate the rest two terms.
- Therefore,**
- $f(n) = 5n^2$

- Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.
- In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

- **Order of growth:** Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array.
- Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

Worst Case Analysis

- In the worst case analysis, we calculate *upper bound on running time* of an algorithm.
- In that causes maximum number of operations to be executed.
- It defines the input for which the algorithm takes a huge time.
- For Linear Search, the *worst case happens when the element to be searched is not present in the array*. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.
- It takes average time for the program execution.

Best Case Analysis

- In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes ***minimum number of operations to be executed.***
- It defines the input for which the *algorithm takes the lowest time*
- In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

Asymptotic Analysis(O , Ω , Θ) :

- When we calculate the complexity of an algorithm we often get a complex polynomial. For simplify this complex polynomial we use some notation to represent the complexity of an algorithm call Asymptotic Notation. The function f and g are non negative functions.
 - **Big oh Notation (O)**
 - **Omega Notation (Ω)**
 - **Theta Notation (Θ)**

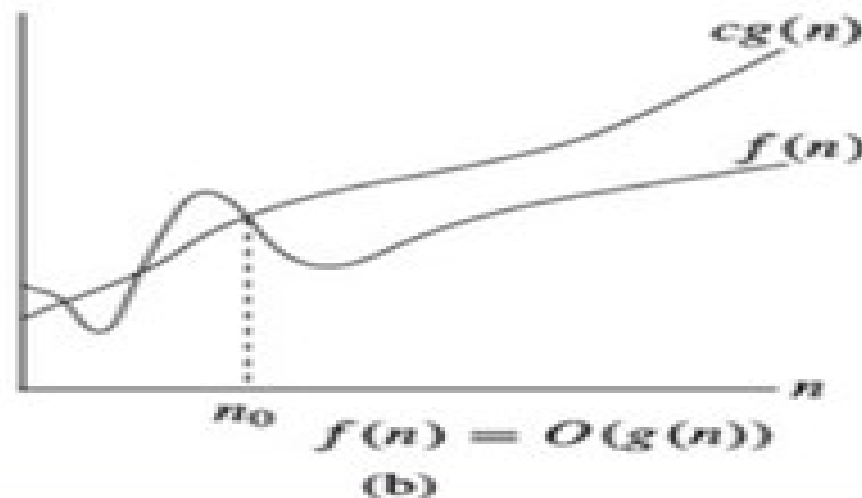
Big oh Notation (O):

This notation provides an **upper bound on a function which ensures that the function never grows faster than the upper bound**. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

If $f(n)$ and $g(n)$ are the two functions defined for positive integers, then $f(n) = O(g(n))$ as $f(n)$ is big oh of $g(n)$ or $f(n)$ is on the order of $g(n)$ if there exists constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

This implies that $f(n)$ does not grow faster than $g(n)$, or $g(n)$ is an upper bound on the function $f(n)$. In this case, we are calculating the **growth rate of the function which eventually calculates the worst time complexity** of a function, i.e., how worst an algorithm can perform.



Omega Notation (Ω)

1.It basically describes the ***best-case scenario which is opposite to the big o notation.***

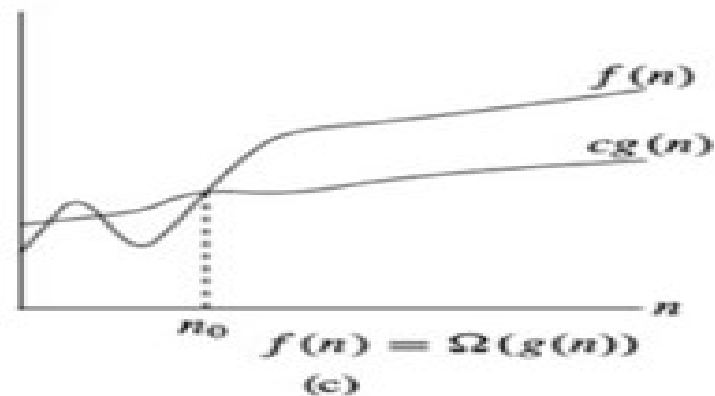
2.It is the formal way to represent the lower bound of an algorithm's running time. ***It measures the best amount of time an algorithm*** can possibly take to complete or the best-case time complexity.

3.It determines what is the fastest time that an algorithm can run.

If $f(n)$ and $g(n)$ are the two functions defined for positive integers,

then $f(n) = \Omega(g(n))$ as $f(n)$ is Omega of $g(n)$ or $f(n)$ is on the order of $g(n)$ if there exists constants c and n_0 such that:

$f(n) \geq c \cdot g(n)$ for all $n \geq n_0$ and $c > 0$



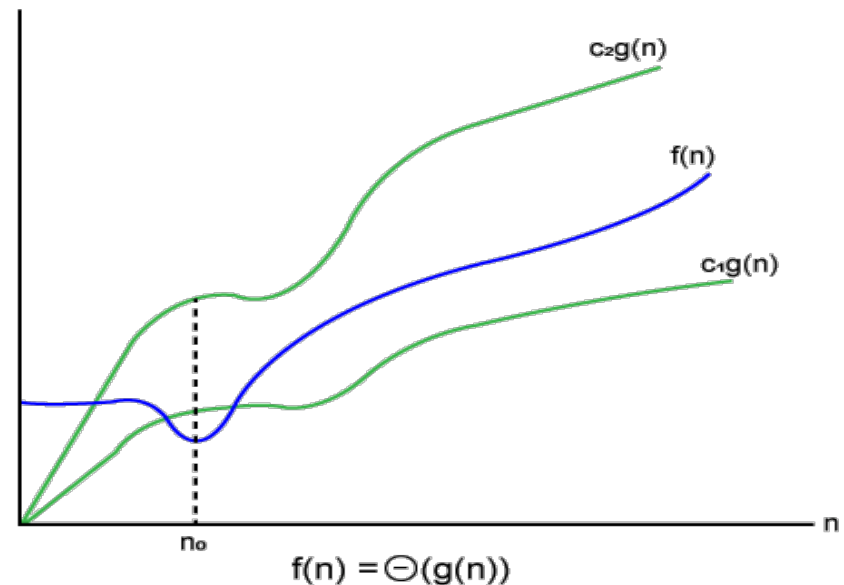
Theta Notation (θ)

- The theta notation mainly describes the *average case scenarios*.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, *algorithms mainly fluctuate between the worst-case and best-case*, and this gives us the average case of the algorithm.
- Let $f(n)$ and $g(n)$ be the functions of n where n is the steps required to execute the program then:

$$f(n) = \theta g(n)$$

The above condition is satisfied only if when

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



➤ **The common name of few order notations is listed below:**

Name	Notation	Example Algorithms
Logarithmic	$O(\log n)$	Binary Search
Linear	$O(n)$	Linear Search
Superlinear	$O(n \log n)$	Heap Sort, Merge Sort
Polynomial	$O(n^c)$	Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort
Exponential	$O(c^n)$	Tower of Hanoi
Factorial	$O(n!)$	Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem

- A Comparison of typical running time of different order notations for different input size listed below:

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1014	32768	4294967296

- The growth patterns of order notations have been listed below:
 $O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \dots$

Example 1 Show that $4n^2 = O(n^3)$.

- Solution By definition, we have

$$0 \leq f(n) \leq cg(n)$$

- Substituting $4n^2$ as $f(n)$ and n^3 as $g(n)$, we get

$$0 \leq 4n^2 \leq cn^3$$

Dividing by n^3

$$0/n^3 \leq 4n^2/n^3 \leq cn^3/n^3$$

$$0 \leq 4/n \leq c$$

Now to determine the value of c , we see that $4/n$ is maximum when $n=1$. Therefore, $c=4$.

To determine the value of n_0 ,

$$0 \leq 4/n_0 \leq 4$$

$$0 \leq 4/4 \leq n_0$$

$$0 \leq 1 \leq n_0$$

This means $n_0=1$. Therefore, $0 \leq 4n^2 \leq 4n^3, \forall n \geq n_0=1$.

Example 2 Show that $400n^3 + 20n^2 = O(n^3)$.

- Solution By definition, we have

$$0 \leq f(n) \leq cg(n)$$

Substituting $400n^3 + 20n^2$ as $f(n)$ and n^3 as $g(n)$, we get

$$0 \leq 400n^3 + 20n^2 \leq cn^3 \text{ Dividing by } n^3$$

$$0/n^3 \leq 400n^3/n^3 + 20n^2/n^3 \leq cn^3/n^3$$

$$0 \leq 400 + 20/n \leq c$$

Note that $20/n \rightarrow 0$ as $n \rightarrow \infty$, and $20/n$ is maximum when $n = 1$.

Therefore, $0 \leq 400 + 20/1 \leq c$

This means, $c = 420$

To determine the value of n_0 ,

$$0 \leq 400 + 20/n_0 \leq 420$$

$$-400 \leq 400 + 20/n_0 - 400 \leq 420 - 400$$

$$-400 \leq 20/n_0 \leq 20$$

$$-20 \leq 1/n_0 \leq 1$$

$$-20 n_0 \leq 1 \leq n_0. \text{ This implies } n_0 = 1.$$

Hence, $0 \leq 400n^3 + 20n^2 \leq 420n^3 \forall n \geq n_0 = 1$.

- **Example 3 Show that $n = O(n \log n)$.**
- Solution By definition, we have

$$0 \leq f(n) \leq c g(n)$$

- Substituting n as $f(n)$ and $n \log n$ as $g(n)$, we get

$$0 \leq n \leq c n \log n$$

Dividing by $n \log n$, we get

$$0/n \log n \leq n/n \log n \leq c n \log n / n \log n$$

$$0 \leq 1/\log n \leq c$$

We know that $1/\log n \rightarrow 0$ as $n \rightarrow \infty$

To determine the value of c , it is clearly evident that $1/\log n$ is greatest when $n=2$. Therefore,

$$0 \leq 1/\log 2 \leq c = 1. \text{ Hence } c = 1.$$

To determine the value of n_0 , we can write

$$0 \leq 1/\log n_0 \leq 1$$

$$0 \leq 1 \leq \log n_0$$

Now, $\log n_0 = 1$, when $n_0 = 2$.

Hence, $0 \leq n \leq c n \log n$ when $c = 1$ and $\forall n \geq n_0 = 2$.

- **Example 4 Show that $10n^3 + 20n \neq O(n^2)$.**

- Solution By definition, we have

$$0 \leq f(n) \leq cg(n)$$

Substituting $10n^3 + 20n$ as $f(n)$ and n^2 as $g(n)$, we get

$$0 \leq 10n^3 + 20n \leq cn^2$$

Dividing by n^2

$$0/n^2 \leq 10n^3/n^2 + 20n/n^2 \leq cn^2/n^2$$

$$0 \leq 10n + 20/n \leq c$$

$$0 \leq (10n^2 + 20)/n \leq c$$

Hence, $10n^3 + 20n \neq O(n^2)$

Example 5 Show that $5n^2 + 10n = \Omega(n^2)$.

- Solution By the definition, we can write
- $0 \leq cg(n) \leq f(n)$
- $0 \leq cn^2 \leq 5n^2 + 10n$
- Dividing by n^2
- $0/n^2 \leq cn^2/n^2 \leq 5n^2/n^2 + 10n/n^2$
- $0 \leq c \leq 5 + 10/n$
- Now, $\lim_{n \rightarrow \infty} 5 + 10/n = 5$.
- Therefore, $0 \leq c \leq 5$. Hence, $c = 5$
- Now to determine the value of n_0
- $0 \leq 5 \leq 5 + 10/n_0$
- $-5 \leq 5 - 5 \leq 5 + 10/n_0 - 5$
- $-5 \leq 0 \leq 10/n_0$
- So $n_0 = 1$ as $\lim_{n \rightarrow \infty} 1/n = 0$
- Hence, $5n^2 + 10n = \Omega(n^2)$ for $c=5$ and $\forall n \geq n_0 = 1$.

- **Example 6 Show that $n^2/2 - 2n = \Theta(n^2)$.**
- Solution By the definition, we can write
- $c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $c_1 n^2 \leq n^2/2 - 2n \leq c_2 n^2$
- Dividing by n^2 , we get
- $c_1 n^2/n^2 \leq n^2/2n^2 - 2n/n^2 \leq c_2 n^2/n^2$
- $c_1 \leq 1/2 - 2/n \leq c_2$
- This means $c_2 = 1/2$ because $\lim_{n \rightarrow \infty} 1/2 - 2/n = 1/2$ (Big O notation)
- To determine c_1 using Ω notation, we can write
- $0 < c_1 \leq 1/2 - 2/n$
- We see that $0 < c_1$ is minimum when $n = 5$. Therefore,
- $0 < c_1 \leq 1/2 - 2/5$
- Hence, $c_1 = 1/10$
- Now let us determine the value of n_0
- $1/10 \leq 1/2 - 2/n_0 \leq 1/2$
- $2/n_0 \leq 1/2 - 1/10 \leq 1/2$
- $2/n_0 \leq 2/5 \leq 1/2$
- $n_0 \geq 5$
- You may verify this by substituting the values as shown below.
- $c_1 n^2 \leq n^2/2 - 2n \leq c_2 n^2$
- $c_1 = 1/10, c_2 = 1/2$ and $n_0 = 5$ $1/10(25) \leq 25/2 - 20/2 \leq 25/2$
- $5/2 \leq 5/2 \leq 25/2$
- Thus, in general, we can write, $1/10n^2 \leq n^2/2 - 2n \leq 1/2n^2$ for $n \geq 5$.

Pointer

- A pointer is a derived data type in 'C'. It is built from any one of the primary data type available in 'C' programming language.
- Pointer is a variable that holds the Address of another variable.
- Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer.

Declaring Pointer Variable:- Pointer declaration is similar to other type of variable except asterisk (*) character before pointer variable name. it is also called dereference operator.

“Value at Address”(*) Operator

The * Operator is also known as Value at address operator.

syntax to declare a pointer:- *data_type *variable_name;*

int *p;

char *ch;

float *pf;

Example

```
#include<stdio.h>
int main()
{
int*p;
int v=10;
p=&v; /* Assigning the address of variable v to the pointer p*/
printf("Value of variable v is: %d",v);
printf("\n Value of variable v is: %d",*p);
printf("\n Address of variable v is:%p",&v);
printf("\n Address of variable v is: %p", p);
printf("\nAddress of pointer p is: %p",&p);
}
```

Note: *Pointers can be outputted using %p, since, most of the computers store the address value in hexadecimal form using %p gives the value in that form. But for simplicity and understanding we can also use %u to get the value in Unsigned int form.*

BENIFITS OF POINTER:-

1. Pointers reduce the length and complexity of a program.
2. Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location.
3. A pointer enables us to access a variable that is defined outside the function.
4. Pointers are more efficient in handling the data tables.
5. The use of a pointer array of character strings results in saving of data storage space in memory.
6. We can **return multiple values from a function** using the pointer.
7. Pointers save memory space.
8. Pointers are used to allocate memory dynamically.

%x -> Hexadecimal value represented with lowercase characters (unsigned integer type)
%X -> Hexadecimal value represented with uppercase characters (unsigned integer type)
%p -> Displays a memory address (pointer type) hexadecimal system are depending by our system bit.
%u which is format specifier for printing an unsigned integer.

```
#include <stdio.h>
int main( )
{
int a = 5;
int *b;
b = &a;
printf ("value of a = %d\n", a);
printf ("value of a = %d\n", *(&a));
printf ("value of a = %d\n", *b);
printf ("address of a = %u\n", &a);
printf ("address of a = %p\n", b);
printf ("value of b = address of a = %x\n",
    b);
printf ("address of b = %X", &b);
return 0;
}
```

OUTPUT:

value of a = 5
value of a = 5
value of a = 5
address of a = 6487580
address of a = 000000000062FE1C
value of b = address of a = 62fe1c
address of b = 62FE10

INITIALIZATION OF POINTER VARIABLE:-

- The process of assigning the address of a variable to a pointer variable is known as **initialization**. Once a pointer variable has been declared we can use the assignment operator to initialize the variable. EXAMPLE:

```
#include<stdio.h>
```

```
int main()
```

```
{ int a;
```

```
  int *ptr;
```

```
  a = 10;
```

```
  ptr = &a;
```

```
  printf("Value of ptr:%u", ptr);
```

```
  return (0);
```

```
}
```

```
int quantity;
```

```
int *p;
```

```
p= &quantity;
```

Dereferencing Pointer:

- Dereferencing is an operation performed to access and manipulates data contained in the memory location pointed to by a pointer.
- The operator `*` is used to dereference pointers. A pointer variable is dereference when the unary operator `*`, in this case called the indirection operator, is used as a prefix to the pointer variable.

```
#include<stdio.h>
int main()
{
int *p, v1, v2;
p=&v1;
*p=25;
*p+=10;
printf(" value of v1= %d\n", v1);
v2=*p;
printf("value of v2= %d\n", v2);
p=&v2;
*p+=20;
printf(" now the value of v2=%d \n", v2);
}
```

output
value of v1= 35
value of v2= 35
now the value of v2=55

Accessing the Address of a Variable through its Pointer:-

- Once a pointer has been assigned the address of variable, the question is how to access a value of a variable through its pointer? This is done by using another unary operator *(asterisk), Consider the following statements.

```
int ds, *p, n;  
ds = 133;  
p = &ds;  
n = *p;
```

- Here we can see that **ds** and **n** declare as integer and **p** pointer variable that point to an integer. When the operator ***** is placed before the pointer variable in an expression, the pointer returns the values of variable of which the pointer value is the address. In this case, ***p** returns the value of variable **ds**. Thus the value of **n** would be **133**. The two statements

```
p = &ds;  
n = *p; are equivalent to  
n = *&ds; which is turn is equivalent to n = ds;
```

```
int main()
{ int *p, q;
  q = 19;
  p = &q;    /* assign p the address of q */
  printf(" Vlaue of q=%d\n",q);
  printf("Contents of p=%d\n ", *p);
  printf("Address of q stored in p=%d ",p);
  return 0;
}
```

OUTPUT:

Vlaue of q=19

Contents of p=19

Address of q stored in p=6487572

Pointer Arithmetic

- The size of data type which the pointer variable points to is the number of bytes accessed in memory.
- The size of the pointer variable is dependent on the data type of the variable pointed by the pointer.
- Some arithmetic operations can be performed with pointers.
- C language supports the following arithmetic operators which can be performed with pointers .They are
 1. **Pointer increment(++)**
 2. **Pointer decrement(- -)**
 3. **addition (+)**
 4. **subtraction (-)**
 5. **Subtracting two pointers of the same type**
 6. **Comparison of pointers**

Pointer increment and decrement:-

- Integer, float, char, double data type pointers can be incremented and decremented. For all these data types both prefix and post fix increment or decrement is allowed.
- Integer pointers are incremented or decremented in the multiples of two. Similarly character by one, float by four and double pointers by eight etc For 32-bit machine.
- **Note** that pointer arithmetic cannot be performed on void pointers, since they have no data type associated with them.
- Let `int *x;`
`x++ /*valid*/`
`++x /*valid*/`
`x-- /*valid*/`
`--x /*valid*/`

- The Rule to increment the pointer is given as

$$\text{new_address} = \text{current_address} + i * \text{size_of}(\text{data type})$$

- Where i is the number by which the pointer get increased.
- For 32-bit int variable, it will be incremented by 2 bytes.
- For 64-bit int variable, it will be incremented by 4 bytes.
- The Rule to decrement the pointer is given as

$$\text{new_address} = \text{current_address} - i * \text{size_of}(\text{data type})$$

Note: If we have more than one pointer pointing to the same location, then the change made by one pointer will be the same as another pointer.

```
void main()
{
int *p1,x;
float *f1,f;
char *c1,c;
p1=&x;
f1=&f;
c1=&c;
printf("Memory address before increment:\n int=%p\n,float=%p\n, char=%p\n",p1,f1,
    ,c1);
p1++;
f1++;
c1++;
printf("Memory address after increment:\n int=%p\n, float=%p\n,char=%p\n",p1,f1,
    c1);
}
```

C Pointer Addition

- We can add a value to the pointer variable. The formula of adding value to pointer is given

new_address = current_address + (number * size_of(data type))

```
#include<stdio.h>
int main(){
int number=50;
int *p;          //pointer to int
p=&number; //stores the address of number variable
printf("Address of p variable is %u \n", p);
p=p+3;           //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n", p);
return 0;
}
```

Output

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

C Pointer Subtraction

- Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given as:

new_address = current_address - (number * size_of(data type))

```
#include<stdio.h>
int main()
{
    int number=50;
    int *p; //pointer to int
    p=&number; //stores the address of number variable
    printf("Address of p variable is %u \n", p);
    p=p-3; //subtracting 3 from pointer variable
    printf("After subtracting 3: Address of p variable is %u \n", p);
    return 0;
}
```

Output

- Address of p variable is 3214864300**
- After subtracting 3: Address of p variable is 3214864288**

Subtraction of one pointer from another:-

- The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bytes of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers. **This operation is useful for calculating the distance or offset between two pointers.**

Example: Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **$(4/4) = 1$** .

```
#include <stdio.h>
main()
{ int x[] = {10, 20, 30, 45, 67, 56, 74} ;
  int *i, *j ;
  i = &x[1] ;
  j = &x[5] ;
  printf("i=%d and j=%d\n",i,j);
  printf ( "%d %d", j - i, *j - *i ) ;
}
```

Output:

i=6487540 and j=6487556

4 36

```
#include <stdio.h>
main()
{
int x = 5, y = 7;
int *p = &y;
int *q = &x;
printf("p is %d\n q is %d\np - q is %d", p, q, (p - q));
}
```

Output

p is 6487560

q is 6487564

p - q is -1

Comparison of Pointers

We can compare pointers using operators like >, >=, <, <=, ==, and !=. These operators return true for valid conditions and false for unsatisfied conditions.

```
#include <stdio.h>
int main()
{
    int arr[5];
    int *ptr2 = &arr[0];
    int *ptr1 =arr;
    if (ptr1 == ptr2)
    {
        printf("Pointer to Array Name and First Element are Equal.");
    }
    else
    {
        printf("Pointer to Array Name and First Element are not Equal.");
    }
    return 0;
}
```

Null Pointers

- A pointer variable is a pointer to a variable of some data type. However, in some cases, we may prefer to have a null pointer which is a special pointer value and does not point to any value.
- This means that a null pointer does not point to any valid memory address.
- This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a null pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries.

```
#include<stdio.h>
int main ()
{
int *ptr=NULL;
printf("The value of ptr is : %x\n",&ptr);
printf("The value of ptr is :%d\n",ptr);
return 0;
}
```

Output

The value of ptr is : 62fe18

The value of ptr is :0

Pointers does not allow the flowing operation

- (a) Addition of two pointer
- (b) Multiplication of a pointer with a constant
- (c) Division of a pointer with a constant

There are various operations which can not be performed on pointers.

Address + Address = illegal

Address * Address = illegal

Address % Address = illegal

Address / Address = illegal

Address & Address = illegal

Address ^ Address = illegal

Address | Address = illegal

~Address = illegal

void Pointer

void Pointer: A generic pointer is a pointer variable that has **void** as its data type.

- The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type.
- For example if we have a pointer to int, then it would be incorrect to assign the address of a float variable to it. But an exception to this rule is a pointer to void.
- Syntax of declaration of a void type:
<void> *<variable name>;
- A void pointer cannot be dereferenced simply by using indirection operator. Before dereferencing, it should be type cast to appropriate pointer data type.
- Pointer arithmetic cannot be performed on void pointers without typecasting.

```
int main ()
{
    int a=3;
    float b=3.4, *fp=&b;
    void *vp;
    vp=&a;
    printf ("value of a= %d\n", *(int *)vp);
    *(int *)vp=12;
    printf ("value of a= %d\n", *(int *)vp);
    vp = fp;
    printf ("value of b= %f\n", *(float *)vp);
}
```

Output:

value of a= 3

value of a= 12

value of b=3.400000

Precedence of dereferencing Operator and Increment /decrement Operators

- The precedence level of * operator and increment/ decrement operators is same and their associativity is from right to left.
- Suppose ptr is an integer and x in an integer variable.
 - **X=*ptr++;**
 - The expression *ptr++ is equivalent to *(ptr++), since these operators associate from right to left. The increment operator is postfix, so first the value of ptr will be used in the expression and then it will be incremented.
 - **X=*++ptr;**
 - The expression *++ptr is equivalent to *(++ptr), since these operators associate from right to left. The increment operator is preefix, so first ptr will be incremented
 - **X=++*ptr;**
 - The expression ++*ptr is equivalent to ++(*ptr), since these operators associate from right to left.
 - **X=(*ptr)++;**
 - Here increment operator is applied over (*ptr), since it is postfix increment hence first the value of *ptr will be assigned to x and then it will be incremented.

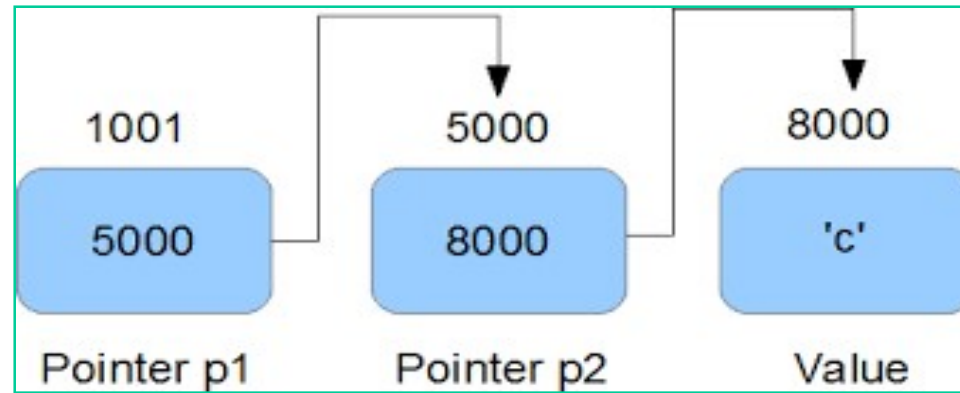
```
#include <stdio.h>
int main()
{ int nums[] = {10, 20, 30, 40, 50};
  int *p = nums;    //pointer storing the address of nums array
  int var;
  var = ++*p;        //value at index 0 incremented by 1
  printf(" *p = %d, var = %d \n", *p, var);
  var = *p++;        //value of index 0 stored in var and then pointer get inc.
  printf(" *p = %d, var = %d \n", *p, var);
  var = *++p;        //pointer address get inc. and then value is stored
  printf(" *p = %d, var = %d \n", *p, var);
  return 0;
}
```

Output

```
*p = 11, var = 11
*p = 20, var = 11
*p = 30, var = 30
```

Pointer to pointer

- It is possible to make a pointer to point to another pointer, (this is also known as **Pointer to pointer** and **Double pointer**), thus creating a chain of pointer as shown.



- Here, the pointer variable **p1** contains the address of the pointer variable **p2**. This is known as *multiple indirections*.
- A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.
- The general syntax for declaring a pointer to pointer is:

<data type> **< pointer to pointer variable name>.

Example: **int **p1;**

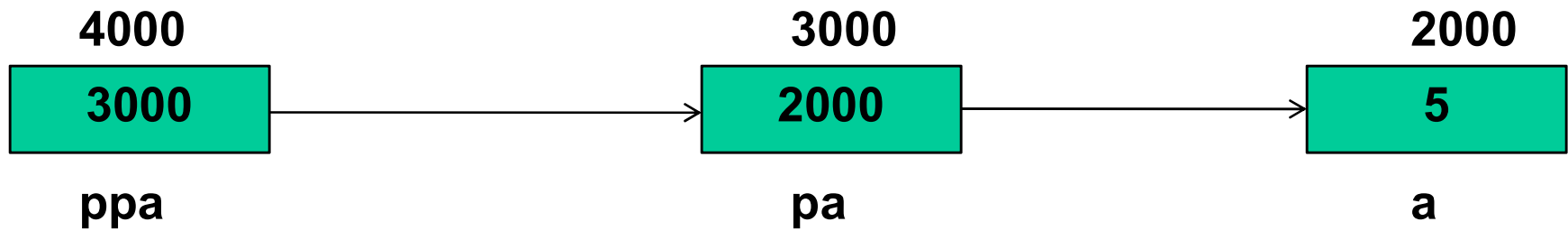
```
int main ()
{
intvar;
int *ptr;
int **pptr;
var = 3000;
ptr = &var; /* take the address of var */
pptr = &ptr; /* take the address of ptr using address of operator & */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr); /* take the value using pptr */
}
```

OUTPUT:-

Value of var = 3000

Value available at *ptr = 3000

Value available at **pptr = 3000



Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

Use of pointers to pointers

- Double pointers are often used in situations where we need to allocate memory for a two-dimensional array dynamically or when we need to pass a pointer to a pointer to a function to modify the original pointer value.
- Note that if we used a single pointer to hold the address of the row pointer array, we would not be able to allocate memory for the columns of the array dynamically, since the single pointer would only hold the address of the first row pointer, and we would not be able to access the other rows of the array.
- If we used a single pointer in the modify Pointer function, we would only be able to modify the copy of the original pointer value that was passed to the function, not the original pointer value itself. Therefore , using a double pointer is necessary to modify the original pointer value directly.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int m,n;
    printf("input number of row and column\n");
    scanf("%d \n %d",&m, &n);
    int** arr; // Allocate memory for rows
    arr = (int**)malloc(m * sizeof(int*));
    for (int i = 0; i < m; i++)
        arr[i] = (int*)malloc(n * sizeof(int));
    printf("input element in matrix \n");
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            scanf("%d",&arr[i][j]);
        }
    }
}
```

```
// Print the array
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
    {
        printf("%d ", arr[i][j]);
    }
    printf("\n");
}

// Free allocated memory
for (int i = 0; i < m; i++)
{
    free(arr[i]);
}
free(arr);
return 0;
}
```

```
#include <stdio.h>

// Function that takes array of strings as argument
void print(char** arr, int n) {
    for (int i = 0; i < n; i++)
        printf("%s\n", *(arr + i));
}

int main()
{
    char* arr[10] = {"Geek", "Geeks ", "Geekfor"};
    print(arr, 3);
    return 0;
}
```

C Array

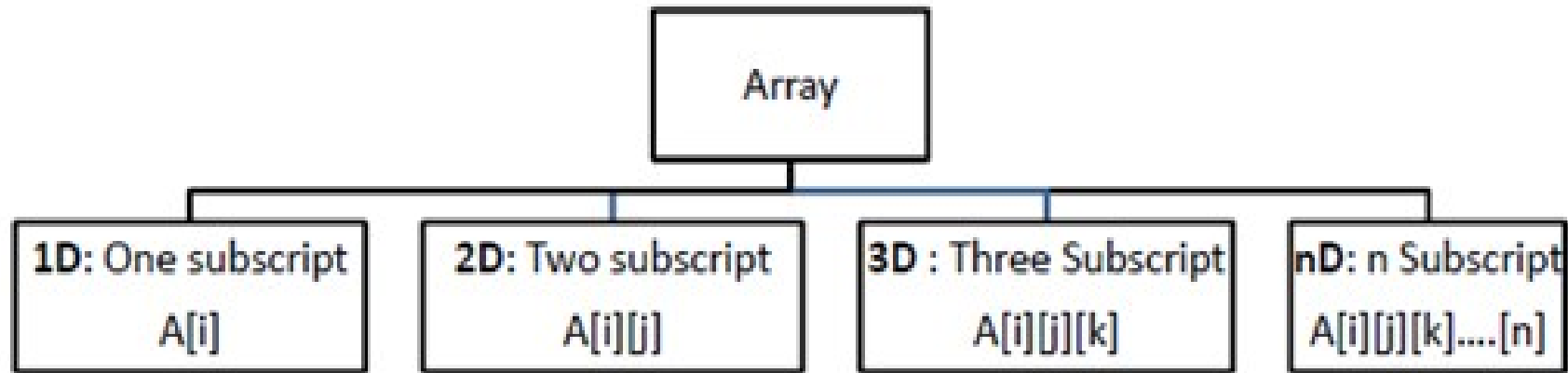
- An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- It also has the capability to store the collection of derived data types, such as pointers, structure, etc.
- The array is the simplest data structure where each data element can be randomly accessed by using its index number. example

Price of n items: Shares a common attribute that they are numeric value.

	item1p	item2p	item3p	item4p	item5p	item6p	item7p	item8p	-----	itemnp
Index	0	1	2	3	4	5	6	7	-----	n

This can be represented using a common name (say price) and index / subscript, as different terms represented in a sequence.

e.g. price[5] gives the price of the fifth item in the list.



Declaration One Dimensional Array: arrays must be declared explicitly before they are used. The general form of declaration is:

Data_type Array_Name[Array_Size];

int marks[5];

- Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of One Dimensional Array

- Once an array is declared, it must be initialized. Otherwise array will contain the garbage values. There are two different ways in which we can initialize the static array

1. Compile time

2. Run time

- **Compile Time Initialization**

1. **Initializing Arrays during Declaration**

data-type array-name[size] = { list of values separated by comma };

For example:

```
int marks[5]={90, 82, 78, 95, 88};
```

marks[0]	90
marks[1]	82
marks[2]	78
marks[3]	95
marks[4]	88

- **Note** that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros.

<code>int marks [5] = {90, 45, 67, 85, 78};</code>	<table><tr><td>90</td><td>45</td><td>67</td><td>85</td><td>78</td></tr><tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr></table>	90	45	67	85	78	[0]	[1]	[2]	[3]	[4]		
90	45	67	85	78									
[0]	[1]	[2]	[3]	[4]									
<code>int marks [5] = {90, 45};</code>	<table><tr><td>90</td><td>45</td><td>0</td><td>0</td><td>0</td></tr><tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr></table> <div>Rest of the elements are filled with 0's</div>	90	45	0	0	0	[0]	[1]	[2]	[3]	[4]		
90	45	0	0	0									
[0]	[1]	[2]	[3]	[4]									
<code>int marks [] = {90, 45, 72, 81, 63, 54};</code>	<table><tr><td>90</td><td>45</td><td>72</td><td>81</td><td>63</td><td>54</td></tr><tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td><td>[5]</td></tr></table>	90	45	72	81	63	54	[0]	[1]	[2]	[3]	[4]	[5]
90	45	72	81	63	54								
[0]	[1]	[2]	[3]	[4]	[5]								
<code>int marks [5] = {0};</code>	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr></table>	0	0	0	0	0	[0]	[1]	[2]	[3]	[4]		
0	0	0	0	0									
[0]	[1]	[2]	[3]	[4]									

initialize an array is by using the index of each element: We can initialize each element of the array by using the index.

```
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```

Run Time Initialization

- An array can be initialized at run time by the program or by taking the input from the keyboard. Generally, the large arrays are declared at run time in the program itself. Such as –

```
int i, marks[10];  
printf("Enter any ten marks: ");  
for(i=0;i<10;i++)  
scanf("%d", &marks[i]);
```

Accessing array elements: An element is accessed by indexing the array name.

```
Array_name[index];  
for example marks[0];  
marks[1];
```



```
main()  
{  
    int i;  
    int x[10];  
    for (i=0; i<10; i++)  
    {  
        x[i] = i + 1;  
        printf( "x=%d.\n", x[i]);  
    }  
}
```

```
main( )  
{  
    int val[10], i, total=0;  
    printf("Enter any ten numbers: ");  
    for(i=0;i<10;i++)  
        scanf("%d", &val[i]);  
    for(i=0;i<10;i++)  
        total = total + val[i];  
    printf("\nTotal is: %d", total);  
}
```

Write a program to find the mean of n numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i, n, arr[20], sum =0;
float mean = 0.0;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
for(i=0;i<n;i++)
sum += arr[i];
mean = (float)sum/n;
printf("\n The sum of the array elements = %d", sum);
printf("\n The mean of the array elements = %.2f", mean);
return 0;
}
```

Calculating the Address of Array Elements

- When we use the array name, we are actually referring to the first byte of the array. Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient.
- The address of other data elements can simply be calculated using the base address.

Address of data element, $A[k] = BA(A) + w(k - \text{lower_bound})$

- For Example Given an array `int marks[] = {99, 67, 78, 56, 88, 90, 34, 85}`, calculate the address of `marks[4]` if the base address = 1000.

99	67	78	56	88	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	1008	1010	1012	1014

- We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes. $\text{marks}[4] = 1000 + 2(4 - 0)$
 $= 1000 + 2(4) = 1008$

Calculating the Length of an Array

- The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is

$$\text{Length} = \text{upper_bound} - \text{lower_bound} + 1$$

- where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.
- Example Let `Age[5]` be an array of integers such that `Age[0] = 2`, `Age[1] = 5`, `Age[2] = 3`, `Age[3] = 1`, `Age[4] = 7` Show the memory representation of the array and calculate its length.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

$$\text{Length} = \text{upper_bound} - \text{lower_bound} + 1$$

- Here, `lower_bound = 0`, `upper_bound = 4`
- Therefore, `length = 4 - 0 + 1 = 5`

Pointers and one dimensional Arrays:-

- The elements of an array are store in contiguous memory location. When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array x,

```
int x[5] = { 1, 2, 3, 4, 5 };
```

Assuming that the base address of x is 1000 and each integer requires two bytes, the five elements will be stored as follows:

Element	x[0]	x[1]	x[2]	x[3]	x[4]
Address	1000	1002	1004	1006	1008

- Here variable **x** will give the base address, which is a constant pointer pointing to the first element of the array, **x[0]**. Hence **x** contains the address of **x [0]** i.e. **1000**.

We can also declare a pointer of type int to point to the array **x**.

```
int *p; p = x; // or,
```

```
p = &x[0]; //both the statements are equivalent.
```

- Now we can access every element of the array **x** using **p++** to move from one element to another.
- When using pointers, an expression like **x[i]** is equivalent to writing ***(x+i)**.

NOTE: You cannot decrement a pointer once incremented. **p--**
Won't work.

- Many beginners get confused by thinking of array name as a pointer.
- For example, while we can write **ptr = x; // ptr = &x[0]**
- **we cannot write x = ptr;**
- This is because while **x** is a variable, **x** is a constant. the location at which the first element of **x** will be stored cannot be changed once **x[]** has been declared. Therefore, an array name is often known to be a constant pointer.
- **Note:** **x[i]**, **i[x]**, ***(x+i)**, ***(i+x)** gives the same value of **ith** element of array **x**.

```
#include<stdio.h>
int main()
{
    int a[]={1,2,3,4,5,6};
    int *b=a;
    b=a++; //error
    printf("%d\n",*b);
}
```

```
#include<stdio.h>
int main()
{
    int a[]={1,2,3,4,5,6};
    int *b=a;
    b=a+1;
    printf("%d\n",*b);
}
```

Arrays Of Pointers

- An array of pointers can be declared as `int *ptr[10];`
- The above statement declares an array of 10 pointers where each of the pointer points to an integer variable.

```
#include <stdio.h>
int main()
{ int arr1[]={1,2,3,4,5};
  int arr2[]={0,2,4,6,8};
  int arr3[]={1,3,5,7,9};
  int *parr[3] = {arr1, arr2, arr3};
  int i;
  for(i = 0;i<3;i++)
    printf(" %d", *parr[i]);
  return 0;
}
```

Output 1 0 1

- `parr[0]` stores the base address of `arr1` (or, `&arr1[0]`). So writing `*parr[0]` will print the value stored at `&arr1[0]`. Same is the case with `*parr[1]` and `*parr[2]`

Pointer to an Array:

- We have seen that a pointer that pointed to the first element of array. We can also declare a pointer that can point to the whole array.
- For example `int (*ptr)[10]`; here `ptr` is point to an array of 10 integer.
- **Note** that it is necessary to enclose the pointer name inside parentheses.
- We know that the pointer arithmetic is performed relative to the base size, so if we write `ptr++`, then the pointer `ptr` will be shifted forward by 20 bytes.

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int arr[] = { 3, 5, 6, 7, 9 };
```

```
    int *p = arr;
```

```
    int (*ptr)[5] = &arr;
```

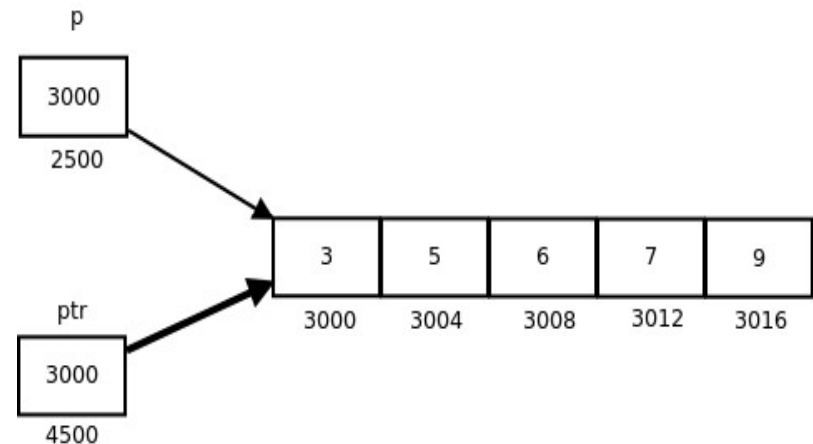
```
    printf("p = %p, ptr = %p\n", p, ptr);
```

```
    printf("*p = %d, *ptr = %p *ptr[0]=%d\n", *p, *ptr, *ptr[0]);
```

```
    printf("sizeof(p) = %lu, sizeof(*p) = %lu\n", sizeof(p), sizeof(*p));
```

```
    return 0;
```

```
}
```



Output: `p = 000000000062FDF0, ptr = 000000000062FDF0`
`*p = 3, *ptr = 000000000062FDF0 *ptr[0]=3`
`sizeof(p) = 8, sizeof(*p) = 4`

```
#include <conio.h>
int main()
{ int *p;
  int x[5];
  int (*ptr)[5];
  p=x;      // Points to 0th element of the x.
  ptr=&x;    // Points to the whole array of x.
  printf(" p=%u,  ptr=%d\n", p, ptr);
  p++;
  ptr++;
  printf(" p=%u,  ptr=%d\n", p, ptr);
}
```

Output:

P=3000, pa=3000

P=3002, pa=3010

```
int main()
{
    int(*a)[5];    // Pointer to an array of five numbers
    int b[5] = { 1, 2, 3, 4, 5 };
    int i = 0;
    a = &b;        // Points to the whole array b
    for (i = 0; i < 5; i++)
        printf("%d\n", *(*a + i));
    return 0;
}
```

Output

1
2
3
4
5

Multi Dimensional Array: It can be defined as ‘an Array which holds more than one subscript is known as Multi Dimensional Array. Generally 2-D array is called as Matrix. It is more suitable for the processing of table and matrix manipulations. C language allow to programmers to use arrays with more than two dimensions. On behalf of this it can be divided into two sub section:

Two Dimensional Array

N-Dimensional Array

Data_Type Array_Name[row size][column size];

For Example: int x[3][4];

Where int is the type of the array, x is the array name and row size is the number of rows and column size the number of column. To find the total number of element of an array, multiply the total number of row with the total number of column.

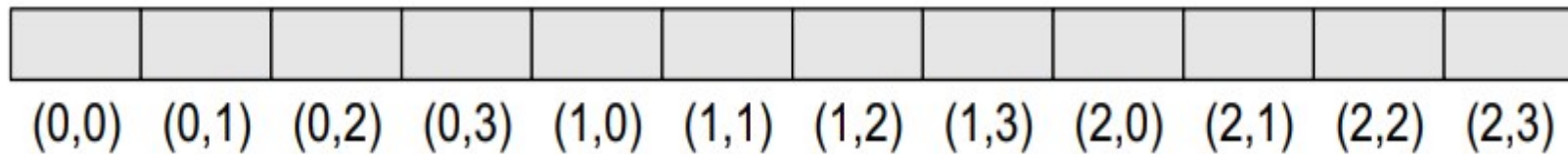
For Example: int a[3][4];

	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]

- Thus, every element in array **a** is identified by an element name of the form **a[i][j]**, where **a** of the array, and i and j are the subscripts (index) that uniquely identify each element in **a**.
- It can be seen that a rectangular picture of a two-dimensional array, in the memory, these elements actually will be stored sequentially.
- There are two ways of storing a two-dimensional array in the memory.
- The First way is the **row major order** and the second is the **column major order**.

row major order

- In a row major order the elements of the first row are stored before the elements of the second and third rows. That is, the elements of the array are stored row by row where n elements of the first row will occupy the first n locations.



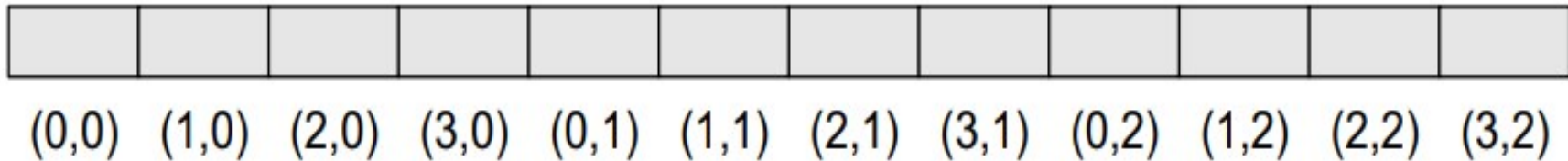
- The computer stores the base address, and the address of the other elements is calculated using the following formula.
- if the array elements are stored in row major order,

$$\text{Address}(A[I][J]) = \text{Base_Address} + w\{N (I - 1) + (J - 1)\}$$

- where w is the number of bytes required to store one element, N is the number of columns, and I and J are the subscripts of the array element.

column major order

- In a column major order, the elements of the first column are stored before the elements of the second and third column. That is, the elements of the array are stored column by column where m elements of the first column will occupy the first m locations.



- The computer stores the base address, and the address of the other elements is calculated using the following formula.
- If the array elements are stored in column major order,
$$\text{Address}(A[I][J]) = \text{Base_Address} + w\{(I - 1) + M (J - 1)\}$$
- where w is the number of bytes required to store one element, M is the number of rows, and I and J are the subscripts of the array element.

- Example Consider a $20 * 5$ two-dimensional array marks which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, marks[18][4] assuming that the elements are stored in row major order.
- Solution

$$\text{Address}(A[I][J]) = \text{Base_Address} + w\{N(I - 1) + (J - 1)\}$$

$$\begin{aligned}\text{Address}(\text{marks}[18][4]) &= 1000 + 2 \{5(18 - 1) + (4 - 1)\} \\ &= 1000 + 2 \{5(17) + 3\} \\ &= 1000 + 2 (88) \\ &= 1000 + 176 \\ &= 1176\end{aligned}$$

Initializing Two-Dimensional Arrays:

- Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} ,    /* initializers for row indexed by 0 */  
    {4, 5, 6, 7} ,    /* initializers for row indexed by 1 */  
    {8, 9, 10, 11}    /* initializers for row indexed by 2 */  
};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, by row index and column index of the array i.e.

Array_name[row index][column index];

For example: **int val = a[2][3];**

The above statement will take 4th element from the 3rd row of the array.

Example

```
#include <stdio.h>
int main ()
{
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}}; /* an array with 5 rows and 2 columns*/
int i, j;

/* output each array element's value */
for ( i = 0; i < 5; i++ )
{
for ( j = 0; j < 2; j++ )
{
printf("a[%d][%d] = %d\n", i, j, a[i][j] );
}
}
return 0;
}
```

Write a program to print a matrix of order $m * n$ where

```
void main( )
{
    clrscr( );
    int mat[50][50] i, j, m, n;
    printf("Enter the number of Rows");
    scanf("%d",&m);
    printf("Enter the number of Columns");
    scanf("%d",&n);
    printf("Enter the element for the Matrix");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d", &mat[i][j]);
        }
    }
    printf("The Matrix is:");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("\t%d",mat[i][j]);
        }
        printf("\n");
    }
    getch( );
}
```

Array with more than two dimensions

- Three-D array as 2-D arrays. For example

```
int x[2][3][4];
```

This array consist of two 2-D arrays and each of those 2-D array has 3 rows and 4 columns.

		Column 0	Column 1	Column 2	Column 3
[0]	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]

		Column 0	Column 1	Column 2	Column 3
[1]	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]

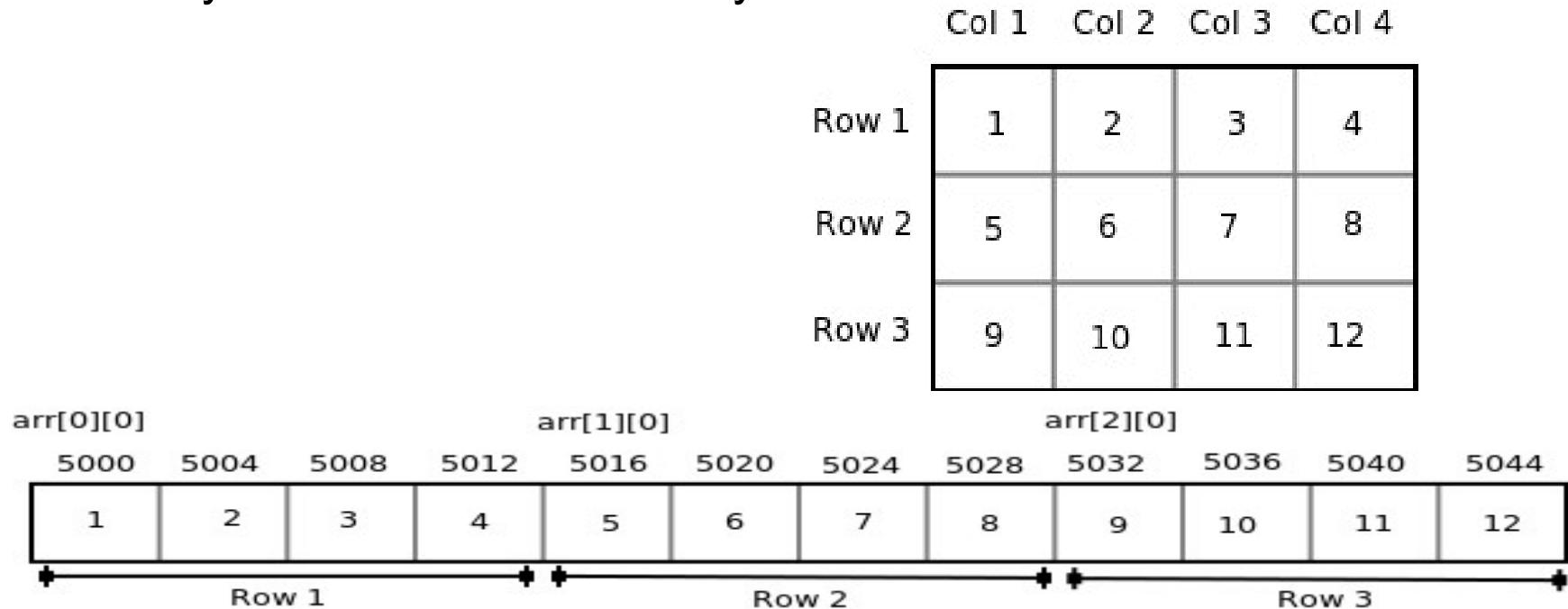
```
#include<stdio.h>
int i,j,k; //variables for nested for loops
int main()
{
    int arr[2][3][4]; //array declaration
    printf("enter the values in the array: \n");
    for(i=1;i<=2;i++) //represents block
    {
        for(j=1;j<=3;j++) //represents rows
        {
            for(k=1;k<=4;k++) //represents columns
            {
                printf("the value at arr[%d][%d][%d]: ", i, j, k
                    );
                scanf("%d", &arr[i][j][k]);
            }
        }
    }
}
```

```
printf("print the values in array: \n");
for(i=1;i<=2;i++)
{
    for(j=1;j<=3;j++)
    {
        for(k=1;k<=4;k++)
        {
            printf("%d ",arr[i][j][k]);
        }
        printf("\n");
    }
    printf("\n");
}
return 0;
}
```

Pointer to Two dimensional Array:

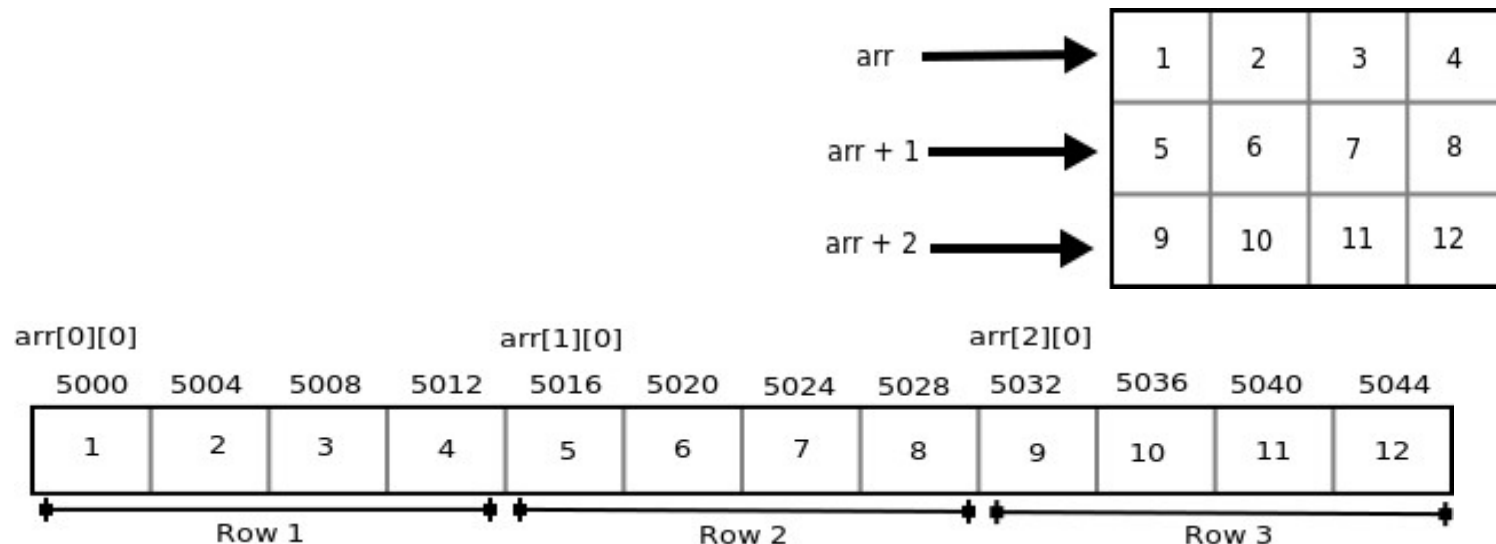
- In two dimensional arrays, we can access each element by using two subscripts, where first subscript represents row number and second subscript represents the column number.
- A two dimensional array is of form, $x[i][j]$. Lets see how we can make a pointer point to such an array. As we know, name of the array gives its base address.
- In $x[i][j]$, x will give the base address of this array, that is the address of $x[0][0]$ element.
- In the case of 2-D arrays, first element is a 1-D array, so the name of 2-D array represents a pointer to 1-D array. **$x[i]$ or $(x+i)$ or $*(x+i)$ point to i^{th} row of 1-D array of the base address where x is two dimension array.**
- Individual elements of the array mat can be accessed using either:
- **$x[i][j]$ or $*(*(x + i) + j)$ or $*(x[i]+j)$;**

- Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



- Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, **we can say that 2-D dimensional arrays that are placed one after another.** So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

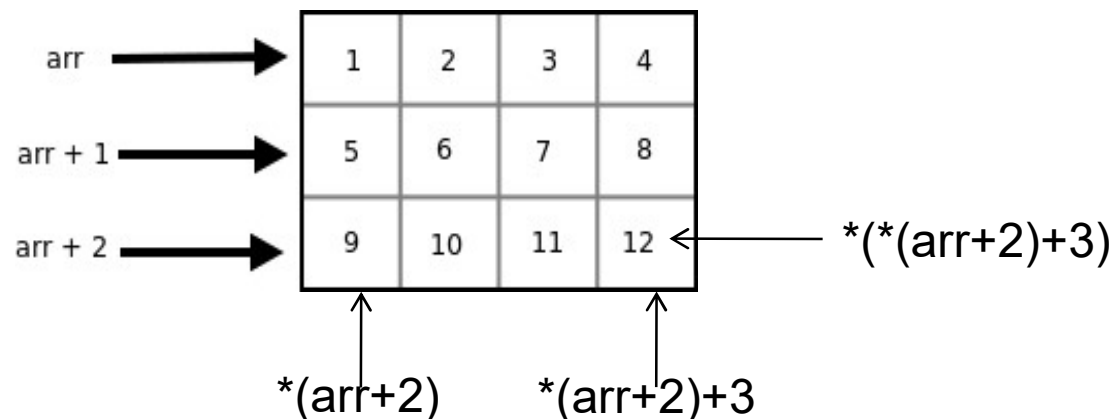
- We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression ***arr* + 1** will represent the address **5016** and expression ***arr* + 2** will represent address **5032**.
- So we can say that *arr* points to the 0th 1-D array, *arr* + 1 points to the 1st 1-D array and *arr* + 2 points to the 2nd 1-D array.



<i>arr</i>	-	Points to 0th element of <i>arr</i>	-	Points to 0th 1-D array	-	5000
<i>arr</i> + 1	-	Points to 1th element of <i>arr</i>	-	Points to 1st 1-D array	-	5016
<i>arr</i> + 2	-	Points to 2th element of <i>arr</i>	-	Points to 2nd 1-D array	-	5032

- arr* + *i*** Points to *i*th element of *arr* - Points to *i*th 1-D array

- Thus the expression $*(arr + i)$ gives us the base address of i^{th} 1-D array.
- The pointer expression $*(arr + i)$ is equivalent to the subscript expression $arr[i]$. So $*(arr + i)$ which is same as $arr[i]$ gives us the base address of i^{th} 1-D array.
- To access an individual element of our 2-D array, we should be able to access any j^{th} element of i^{th} 1-D array.
- Since the base type of $*(arr + i)$ is *int* and it contains the address of 0^{th} element of i^{th} 1-D array, we can get the addresses of subsequent elements in the i^{th} 1-D array by adding integer values to $*(arr + i)$.
- For example $*(arr + i) + 1$ will represent the address of 2^{nd} element of i^{th} 1-D array and $*(arr+i)+2$ will represent the address of 3^{rd} element of i^{th} 1-D array.
- Similarly $*(arr + i) + j$ will represent the address of j^{th} element of i^{th} 1-D array.



```
#include<stdio.h>
int main()
{
    int arr[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33} };
    int (*ptr)[4];
    ptr = arr;
    printf("%p %p %p\n", ptr, ptr + 1, ptr + 2);
    printf("%p %p %p\n", *ptr, *(ptr + 1), *(ptr + 2));
    printf("%d %d %d\n", **ptr, *(*ptr + 1) + 2, *(*ptr + 2) + 3);
    printf("%d %d %d\n", ptr[0][0], ptr[1][2], ptr[2][3]);
    return 0;
}
```

Output:

```
0x7ffead967560 0x7ffead967570 0x7ffead967580
0x7ffead967560 0x7ffead967570 0x7ffead967580
10 22 33
10 22 33
```

```

main( )
{
    int s[4][2] = { { 12, 56 }, { 12, 33 }, { 14, 80 }, { 13, 78 } } ;
    int i, j ;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        printf("\n Address of %d th row one D array=%u  %u\n", i, s[i], *(s+i));
        printf( "\n" );
        for ( j = 0 ; j <= 1 ; j++ )
            printf( "%d ", *( *( s + i ) + j ) ) ;
    }
}

```

Output:

```

Address of 0th row one D array=2293400  229340012 56
Address of 1th row one D array=2293408  229340812 33
Address of 2th row one D array=2293416  229341614 80
Address of 3th row one D array=2293424  229342413 78

```

- The golden rule to access an element of a two-dimensional array can be given as $\text{arr}[i][j] = (*(arr+i))[j] = *((*arr+i)+j) = *(arr[i]+j)$
- Therefore, $\text{arr}[0][0] = *(arr)[0] = *((*arr)+0) = *(arr[0]+0)$

If we declare an array of pointers using,

```
data_type *array_name[SIZE];
```

Here SIZE represents the number of rows and the space for columns that can be dynamically allocated

If we declare a pointer to an array using,

```
data_type (*array_name)[SIZE];
```

Here SIZE represents the number of columns and the space for rows that may be dynamically allocated

Pointers and Three Dimensional Arrays

- In a three dimensional array we can access each element by using three subscripts.
`int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };`
- We can consider a three dimensional array to be an array of 2-D array i.e. each element of a 3-D array is considered to be a 2-D array. The 3-D array *arr* can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array *arr* is a pointer to the 0th 2-D array.

arr	Points to 0th 2-D array.
arr + i	Points to ith 2-D array.
*(arr + i)	Gives base address of ith 2-D array, so points to 0th element of ith 2-D array, each element of 2-D array is a 1-D array, so it points to 0th 1-D array of ith 2-D array.
*(arr + i) + j	Points to jth 1-D array of ith 2-D array.
((arr + i) + j)	Gives base address of jth 1-D array of ith 2-D array so it points to 0th element of jth 1-D array of ith 2-D array.
((arr + i) + j) + k	Represents the value of jth element of ith 1-D array.
((arr + i) + j) + k)	Gives the value of kth element of jth 1-D array of ith 2-D array.

- Thus the pointer expression `*(*(arr + i) + j) + k)` is equivalent to the subscript expression `arr[i][j][k]`.

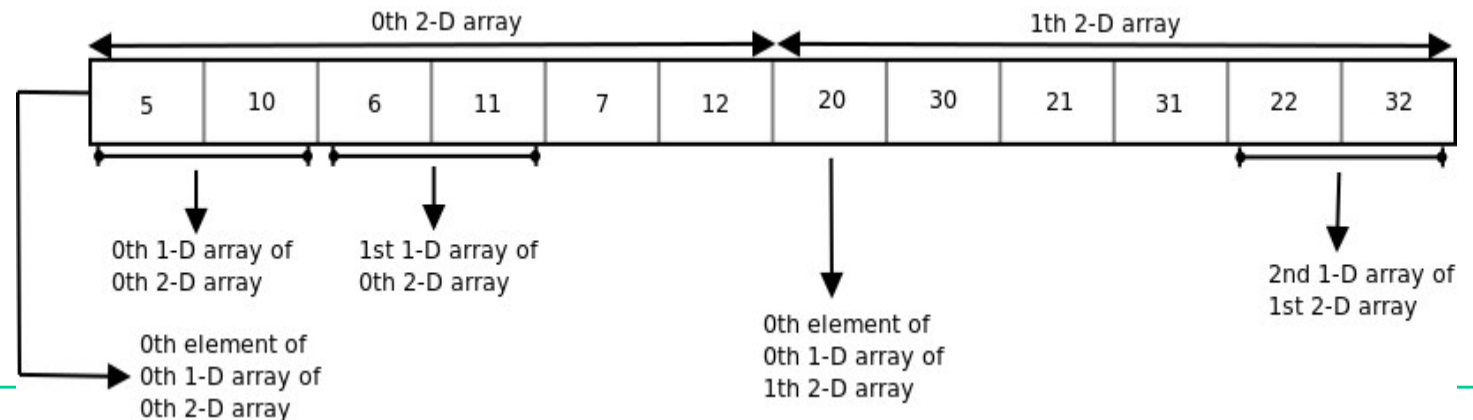
```

#include<stdio.h>
int main()
{ int arr[2][3][2] = { { {5, 10}, {6, 11}, {7, 12}, },
                        { {20, 30}, {21, 31}, {22, 32}, } };

  int i, j, k;
  for (i = 0; i < 2; i++)
  {
    for (j = 0; j < 3; j++)
    {
      for (k = 0; k < 2; k++)
        printf("%d\t", (*(arr + i) + j) + k));
      printf("\n");
    }
  }
  return 0;
}

```

How the 3-D array used in the above program is stored in memory



OPERATIONS ON ARRAYS

- There are a number of operations that can be preformed on arrays. These operations include:
 - **Traversing an array**
 - **Inserting an element in an array**
 - **Searching an element in an array**
 - **Deleting an element from an array**
 - **Merging two arrays**
 - **Sorting an array in ascending or descending order**

Traversing an Array

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements.

Step 1 SET $I = \text{lower_bound}$

Step 2: Repeat Steps 3 to 4 while $I \leq \text{upper_bound}$

Step 3: Apply Process to $A[I]$

Step 4: SET $I = I + 1$

[END OF LOOP]

Step 5: EXIT

Write a program to read and display n numbers using an array.

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int i, n, A[20];
```

```
    printf("\n Enter the number of elements in the array : ");
```

```
    scanf("%d", &n);
```

```
    for(i=0;i<n;i++)
```

```
        scanf("%d", &A[i]);
```

```
    printf("the Array elements are=");
```

```
    for(i=0;i<n; i++)
```

```
        printf("%d \t", A[i]);
```

```
}
```

Inserting an Element in an Array

➤ Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. Here, we assume that the memory space allocated for the array is still available.

Step 1: SET $I=N$

Step 2: Repeat Steps 3 and 4 while $I \geq POS$

Step 3: SET $A[I + 1] = A[I]$

Step 4: SET $I=I-1$

[END OF LOOP]

Step 5: SET $A[POS] = NUM$

Step 6: SET $N=N+1$

Step 7: EXIT

Inserting an Element in an Array

```
#include<stdio.h>
int main()
{ int i, n, num, pos, arr[10];
  printf("\n Enter the number of elements in the array : ");
  scanf("%d", &n);
  for(i=0;i<n;i++)
  scanf("%d", &arr[i]);
  printf("\n Enter the number to be inserted : ");
  scanf("%d", &num);
  printf("\n Enter the position at which the number has to be added :");
  scanf("%d", &pos);
  for(i=n-1;i>=pos;i--)
    arr[i+1] = arr[i];
  arr[pos] = num;
  n = n+1;
  printf("\n The array after insertion of %d is : ", num);
  for(i=0;i<n;i++)
  printf("%d \t", arr[i]);
}
```

Deleting an Element in an Array

➤Deleting an element from an array means removing a data element from an already existing array and re-organizing all elements of an array.

- Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that $K \leq N$.
 1. Start $NUM = LA[K]$
 2. Set $J = K$
 3. Repeat steps 4 and 5 while $J < N$
 4. Set $LA[J] = LA[J + 1]$
 5. Set $J = J + 1$
 6. Set $N = N - 1$
 7. Stop

Write a program to delete a number from a given location in an array

```
#include<stdio.h>

int main()
{ int i, n, pos, arr[10],num;
  printf("\n Enter the number of elements in the array : ");
  scanf("%d", &n);
  printf("\n Enter the elements in the array : ");
  for(i=0;i<n;i++)
  scanf("%d", &arr[i]);
  printf("\nEnter the position from which the number has to be deleted : ");
  scanf("%d", &pos);
  num=arr[pos];
  for(i=pos; i<n-1;i++)
  arr[i] = arr[i+1];
  n=n-1;
  printf("\n The array after deletion is : ");
  for(i=0;i<n; i++)
  printf("\n arr[%d] = %d", i, arr[i]);
}
```

Search Operation

➤ It is the process of finding the location of the element with a given key value in a particular data structure or finding the location of an element, which satisfies the given condition. There are mainly two techniques available to search the data in an array:

1. Linear search

2. Binary search

➤ Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value.

➤ It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. If the match is found, then it returns the index of the element else it returns element not found. This algorithm can be implemented on the unsorted list.

Linear Search Operation

- Consider **A** is a linear array with **N** elements and **K** is a positive integer such that $K \leq N$. Following is the algorithm to find an element with a value of **ITEM** using sequential search.
 1. Start
 2. Set $J = 0$
 3. Repeat steps 4 and 5 while $J < N$
 4. IF $A[J] == \text{ITEM}$
THEN GOTO STEP 6
 5. Set $J = J + 1$
 6. PRINT J, ITEM
 7. Stop

Time Complexity

Operation	Average Case	Worst Case
Access	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$
Insertion	$O(n)$	$O(n)$
Deletion	$O(n)$	$O(n)$

Example of linear search

```
#include<stdio.h>
int main()
{ int i,j, n, item, A[10];
  printf("\n Enter the number of elements in the array : ");
  scanf("%d", &n);
  printf("\n Enter the elements in the array : ");
  for(i=0;i<n;i++)
  scanf("%d", &A[i]);
  printf("\n Enter the item in the array whose search the position : ");
  scanf("%d",&item);
  j=0;
  while( j < n)
  {
    if( A[j] == item )
    { break;
      }
    j = j + 1;
  }
  printf("Found element %d at position %d\n", item, j+1);
}
```

Binary Search

- A Binary algorithm is the simplest algorithm that searches the element very quickly.
- It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm.
- Binary search cannot be implemented if the elements are stored in a random manner

BINARY_SEARCH(A, lower_bound, upper_bound, ITEM)

Step 1: SET BEG = lower_bound ; END = upper_bound, POS=-1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] == ITEM

 SET POS = MID

 PRINT POS Go to Step 6

ELSE IF A[MID] > ITEM

 SET END = MID-1

ELSE

 SET BEG = MID+1

[END OF IF]

[END OF LOOP]

Step 5: IF POS=-1

PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

Step 6: end

- Let the element to search is, $k = 56$ i.e. $ITEM=k$
- We have to use the below formula to calculate the **mid** of the array

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

- beg** = 0
- end** = 8
- mid** = $(0 + 8)/2 = 4$. So, 4 is the mid of the array.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[mid] = 39$
 $A[mid] < K$ (or, $39 < 56$)
 So, $beg = mid + 1 = 5$, $end = 8$
 Now, $mid = (beg + end)/2 = 13/2 = 6$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[mid] = 56$
 $A[mid] = K$ (or, $56 = 56$)
 So, location = mid
 Element found at 7th location of the array

Binary Search complexity

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.
- If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another. But merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted.
- If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult.

Merging Two Arrays

```
void mergeArrays(int arr1[], int arr2[], int n1, int n2, int arr3[])
{
    int i = 0, j = 0, k = 0;
    while (i < n1 && j < n2)
    {
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }
    while (i < n1)
        arr3[k++] = arr1[i++];
    while (j < n2)
        arr3[k++] = arr2[j++];
}
```

Write a program to merge two unsorted arrays.

```
int main()
{int arr1[10], arr2[10], arr3[20];
  int i, n1, n2, m, index=0;
printf("\n Enter the number of elements in array1 : ");
  scanf("%d", &n1);
printf("\n\n Enter the elements of the first array");
  for(i=0;i<n1;i++)
  scanf("%d", &arr1[i]);
printf("\n Enter the number of elements in array2 : ");
  scanf("%d", &n2);
printf("\n\n Enter the elements of the second array");
  for(i=0;i<n2;i++)
  scanf("%d", &arr2[i]);
  m = n1+n2;
  for(i=0;i<n1;i++)
  { arr3[index] = arr1[i];
  index++;}
  for(i=0;i<n2;i++)
  {arr3[index] = arr2[i];
  index++;}
printf("\n\n The merged array is");
  for(i=0; i<m; i++)
  printf("\n arr[%d] = %d", i, arr3[i]);
}
```

Write a program to merge two sorted arrays.

```
#include<stdio.h>
int main()
{ int arr1[10], arr2[10], arr3[20];
  int i, n1, n2, m, index=0; int index_first = 0,
  index_second = 0;
  printf("\n Enter the number of elements in array1 : ");
  scanf("%d", &n1);
  printf("\n\n Enter the elements of the first array");
  for(i=0;i<n1;i++)
  scanf("%d", &arr1[i]);
  printf("\n Enter the number of elements in array2 : ");
  scanf("%d", &n2);
  printf("\n\n Enter the elements of the second array");
  for(i=0;i<n2;i++)
  scanf("%d", &arr2[i]);
  m = n1+n2;
  while(index_first < n1 && index_second < n2)
  {
  if(arr1[index_first]< arr2[index_second])
  {arr3[index] = arr1[index_first];
   index_first++;
  }
  else
  { arr3[index] = arr2[index_second];
   index_second++;
  }
}
```

```
index++;
}
if(index_first == n1)
{
while(index_second<n2)
{
arr3[index] = arr2[index_second];
index_second++;
index++;
} }
else if(index_second == n2)
{
while(index_first < n1)
{ arr3[index] = arr1[index_first];
  index_first++;
  index++;
} }
printf("\n\n The merged array is");
for(i=0;i<m;i++)
printf("\n arr[%d] = %d", i, arr3[i]);
}
```


Exercise

1. Write a program to read an array of n numbers and then find the smallest number.
2. Write a program to interchange the largest and the smallest number in an array.
3. Write a program to transpose a 3×3 matrix.
4. Write a program to input two $m \times n$ matrices and then calculate the sum of their corresponding elements and store it in a third $m \times n$ matrix.
5. Write a program to multiply two $m \times n$ matrices.
6. Write a program to read a 2D array marks which stores the marks of five students in three subjects. Write a program to display the highest marks in each subject.