Module1

Data types to Data structures: Data structures, Classifications: Primitive & Non-Primitive (**Stack, Queue, Linked List, Graph, Trees, and Hash Table**). Data structure operations. Review of pointers, Dynamic Memory Allocation, Functions with Programming Examples **(Dynamic 1D array, 2D array)**, review of structures, polynomial, sparse matrices, String pattern matching (nfind, KMP)

1.1. Introduction to Data Structures:

A data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as: **Algorithm + Data structure = Program**

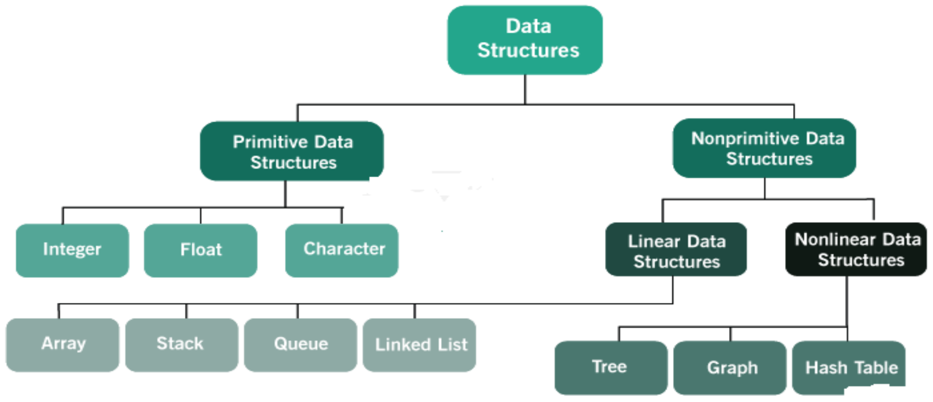
A data structure is said to be linear if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order.

A data structure is said to be non-linear if its elements form a hierarchical classification where, data items appear at various levels. Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements.

1.2 Classification of Data structures

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.



Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

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** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
    a. Linear Data Structures
    b. Non-Linear Data Structures

Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered. The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code.

   Moreover, the user can change the size as well as the data elements stored in these data structures at the run-time of the code. **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

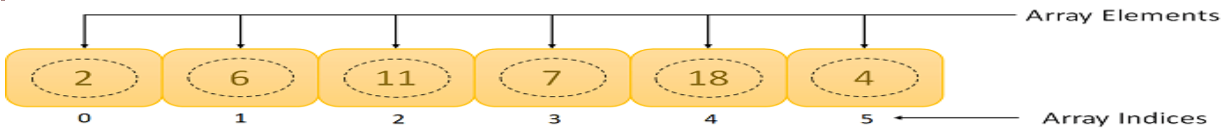
Types of Linear Data Structures

The following is the list of Linear Data Structures that we generally use:

1. Arrays

An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable.

The data elements of the array share the same variable name; however, each carries a different index number called a subscript. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.



An Array

Arrays can be classified into different types:

a) **One-Dimensional Array:** An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.

b) **Two-Dimensional Array:** An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.

c) **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

Some Applications of Array:

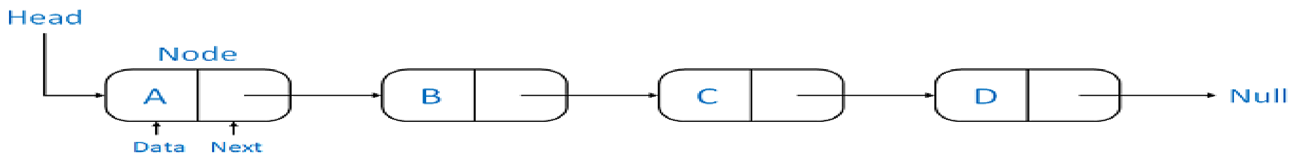
a) We can store a list of data elements belonging to the same data type.

b) Array acts as an auxiliary storage for other data structures.

c) Array also acts as a storage of matrices.

2. Linked Lists

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.



A Linked List

Linked Lists can be classified into different types:

a) **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

b) **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).

c) **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

Some Applications of Linked Lists:

a) The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.

b) We can also implement Operating System's function for dynamic memory management. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.

c) Linked Lists also allow polynomial implementation for mathematical operations.

d) Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.

e) Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

3. Stacks

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

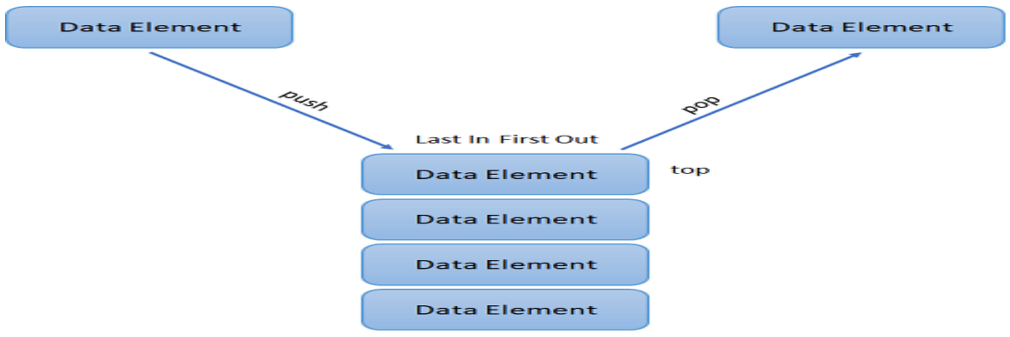
A Real-life Example of Stack

The above figure represents the real-life example of a Stack where the operations are performed from one end only, like the insertion and removal of new books from the top of the Stack. It implies that the insertion and deletion in the Stack can be done only from the top of the Stack. We can access only the Stack's tops at any given time.

The primary operations in the Stack are as follows:

a) **Push:** Operation to insert a new element in the Stack is termed as Push Operation.

b) **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.



A Stack

Some Applications of Stacks:

a) The Stack is used as a Temporary Storage Structure for recursive operations.

b) Stack is also utilized as Storage Structure for function calls, nested operations, and deferred/postponed functions.

c) Stacks are also utilized to evaluate the arithmetic expressions in different programming languages and are also helpful in converting infix expressions to postfix expressions.

d) Stacks allow us to check the expression's syntax in the programming environment. We can match parenthesis using Stacks.

e) Stacks can be used to reverse a String, helpful in solving problems based on backtracking. Stacks are also used in UNDO and REDO functions in an edit.

4. Queues

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays or Linked Lists. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

TICKET
Person who enters first receives the movie ticket first
Open at both ends!
Person who enters last will receives the ticket last

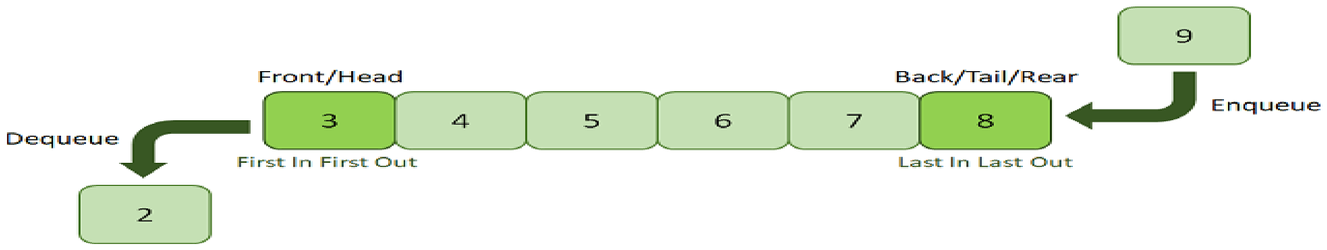
A Real-life Example of Queue

The above image is a real-life illustration of a movie ticket counter that can help us understand the Queue where the customer who comes first is always served first. The customer arriving last will undoubtedly be served last. Both ends of the Queue are open and can execute different operations. Another example is a food court line where the customer is inserted from the rear end while the customer is removed at the front end after providing the service they asked for.

The following are the primary operations of the Queue:

a) **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.

b) **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.



A Queue

Some Applications of Queues:

a) Queues are generally used in the breadth search operation in Graphs.

b) Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.

c) Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.

d) Priority Queues are utilized in file-downloading operations in a browser.

e) Queues are also used to transfer data between peripheral devices and the CPU.

f) Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

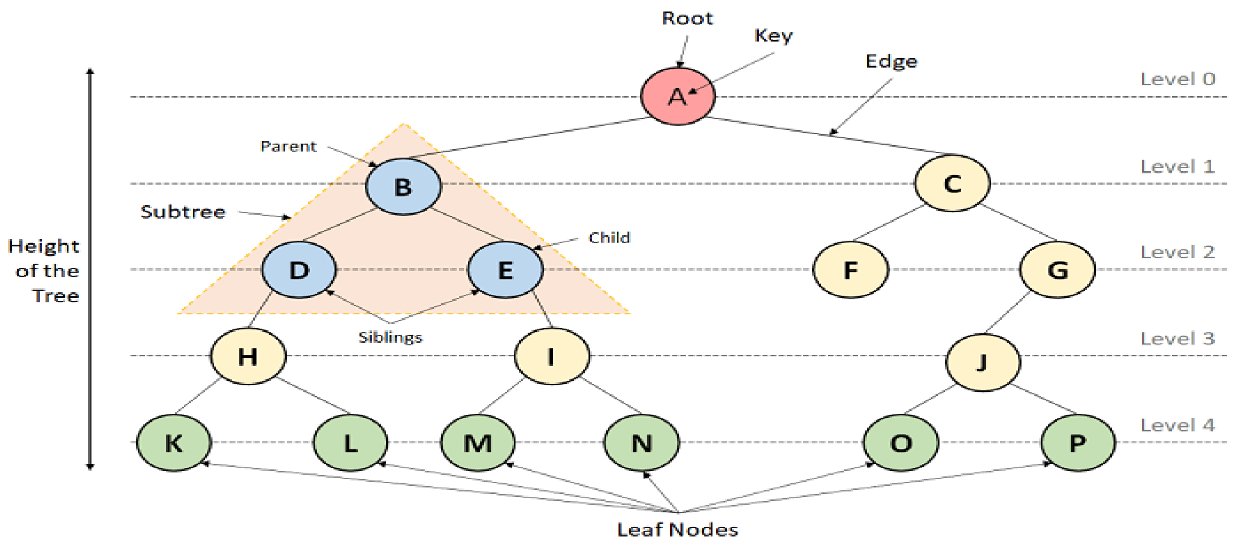
Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

1. Trees

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



A Tree

Trees can be classified into different types:

a) **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.

b) **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

c) **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.

d) **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

Some Applications of Trees:

a) Trees implement hierarchical structures in computer systems like directories and file systems.

b) Trees are also used to implement the navigation structure of a website.

c) Trees are also helpful in decision-making in Gaming applications.

d) Trees are responsible for implementing priority queues for priority-based OS scheduling functions.

e) Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.

f) We can use Trees to store data keys for indexing for Database Management System (DBMS).

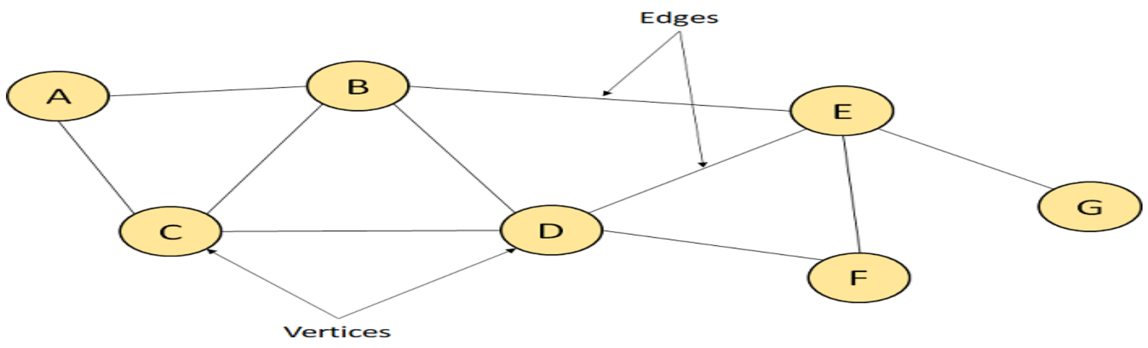
g) Spanning Trees allows us to route decisions in Computer and Communications Networks.

h) Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

2. Graphs

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below: G = (V,E)



A Graph

The above figure represents a Graph having seven vertices A, B, C, D, E, F, G, and ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

Some Applications of Graphs:

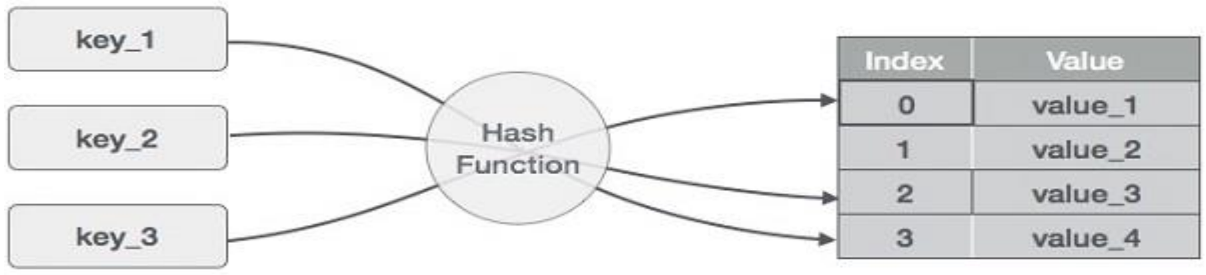
a) Graphs help us represent routes and networks in transportation, travel, and communication applications.
b) Graphs are used to display routes in GPS.
c) Graphs also help us represent the interconnections in social networks and other network-based applications.
d) Graphs are responsible for the representation of user preference in e-commerce applications.
e) Graphs also help to manage the utilization and availability of resources in an organization.
f) Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.

Hash table

In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	**Key**	**Hash**	**Array Index**
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2

3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Some Applications of Hashing

a) **Databases:**

In database indexing, hashing is extensively used to rapidly identify records based on fundamental values, boosting the efficiency of data retrieval processes.

b) **Caching:**

In caching systems, hashing is used to detect whether a particular item is present in the cache, hence avoiding repeated calculations and improving overall system efficiency.

c) **Security:**

Hash functions are critical in cryptography because they generate hash values (hash codes) that reflect data integrity. Hashing is used to improve security in digital signatures and password storage.

d) **Systems that are distributed:**

Hashing is used in distributed systems to balance load. Hash codes help determine which node in a distributed system should be responsible for storing or processing a particular piece of data.

e) **Compiler Symbol Tables:**

Compilers utilize symbol tables to store identifiers (such as variable names) and their associated characteristics. Hashing is frequently used to create efficient symbol tables for speedy lookups.

f) **DHTs (Distributed Hash Tables):**

DHTs distribute key-value pairs over a network of nodes via hashing. This is prevalent in peer-to-peer systems, where each node manages a subset of the key space.

g) **Routing on the network:**

Some network routing techniques use hashing to spread traffic over various pathways. This may result in more equitable consumption of network resources.

Review of Arrays

Introduction to Arrays

The fundamental data types, namely char, int, float, and double are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data.

In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data structure that would facilitate efficient storing, accessing and manipulation of data items.

C supports a derived data type known as **array** that can be used for such applications.

Disadvantages of arrays:

i) Size should be declared in advance.

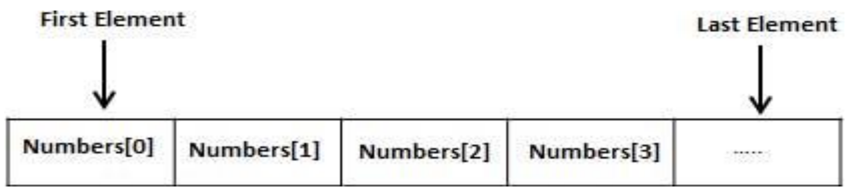
ii) If the number of elements is less than the size, lot of space is wasted.

Array:

An array is a derived data type used to store a collection of similar (same type or homogenous) data items, stored contiguously in memory under a single name.

Instead of declaring individual variables, such as number0, number1, ..., and number99, we declare one array variable such as number and use number[0], number[1], and ..., number[99] to represent individual variables. A specific element in an array is accessed by an index. An index is an integer positive number that starts with 0.

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



Array name refers to the starting address of the array. Arrays are also called as subscripted value.

Some examples where the concepts of an array can be used:

- List of employees in an organization.
- List of customers and their telephone numbers.

use arrays to represent not only simple lists of values but also tables of data in two or three or more dimensions.

Types of arrays

- One dimensional arrays (1-D)
- Two dimensional arrays(2-D)
- Multi-dimensional arrays

ONE DIMENSIONAL ARRAYS

A linear list of data items of same type which are stored contiguously in memory is called one dimensional array (1-D array).

A List of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or a one dimensional array.

DECLARATION OF ONE DIMENSIONAL ARRAYS (DECLARATION OF ARRAYS)

The general form of array declaration is : **type array_name[size];**

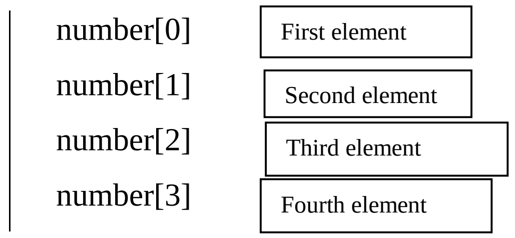
Declaration of array specifies 3 things

1) The **type** specifies the type of element that will be contained in the array such as int, float, double or char.
2) **array_name** : to identify the array.
3) **size** indicates the maximum number of elements that can be stored inside the array.

For example,

int number[4];

The computer reserves four storage locations as shown below:

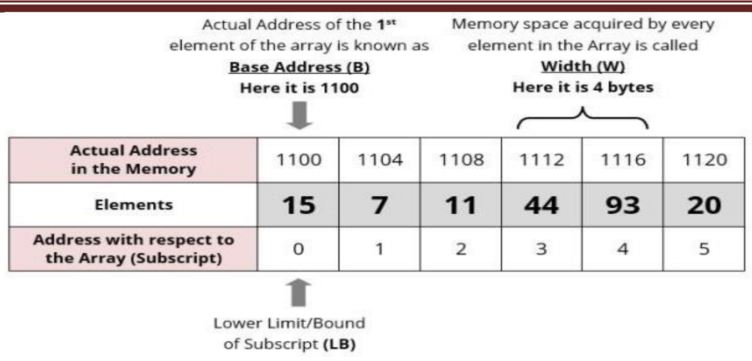


The elements of array are referenced by an index (also known as subscript). Subscript is an ordinal number which is used to identify an element of the array.

Why index starts at 0: An index refers to a relative position of an element from the first element. So , for the fourth element the index is 3 because there are 3 elements before it. For the first element, no element is present before it. So, the index is 0.

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, 2036, so that the element with index i has the address 2000 + 4 × i.

<u>Address Calculation in single (one) Dimension Array:</u>


Actual Address of the 1st element of the array is known as Base Address (B)
Here it is 1100
Memory space acquired by every element in the Array is called Width (W)
Here it is 4 bytes
Lower Limit/Bound of Subscript (LB)

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	15	7	11	44	93	20
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Array of an element of an array say "A[I]" is calculated using the following formula:

Address of A [I] = B + W * (I – LB)

Where, **B** = Base address

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

I = Subscript of element whose address is to be found

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

Example:

Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

Solution:

The given values are: B = 1020, LB = 1300, W = 2, I = 1700

Address of A [I] = B + W * (I – LB)

= 1020 + 2 * (1700 – 1300)

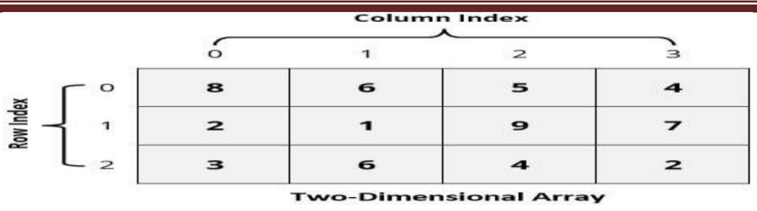
= 1020 + 2 * 400

= 1020 + 800

= 1820 **[Ans]**

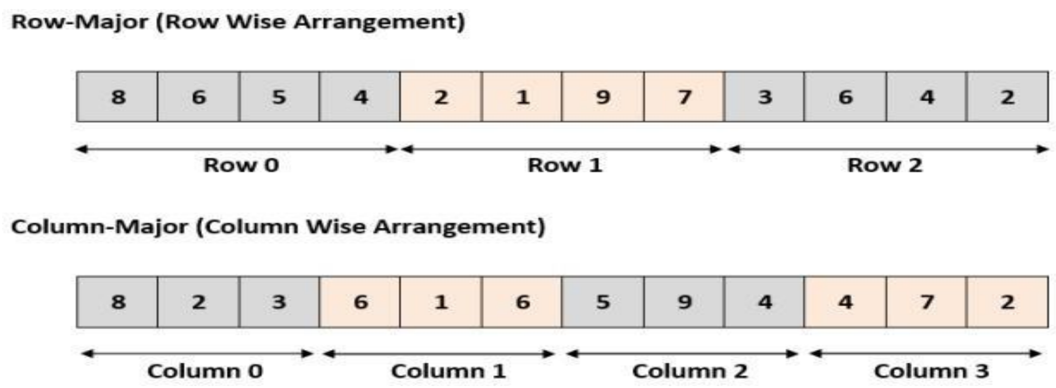
Address Calculation in Double (Two) Dimensional Array:

While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.



C allows for arrays of two or more dimensions. A two-dimensional (2D) array is an array of arrays. A three-dimensional (3D) array is an array of arrays of arrays.

In C programming an array can have two, three, or even ten or more dimensions. The maximum



dimensions a C program can have depends on which compiler is being used.

More dimensions in an array means more data be held, but also means greater difficulty in managing and understanding arrays.

How to Declare a Multidimensional Array in C

A multidimensional array is declared using the following syntax:

type array_name[d1][d2][d3][d4]………[dn];

Where each **d** is a dimension, and **dn** is the size of final dimension.

Examples:

1. **int table[5][5][20];**

2. **float arr[5][6][5][6][5];**

In Example 1:

- **int** designates the array type integer.
- **table** is the name of our 3D array.
- Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case: **5x5x20=500**.

In Example 2:

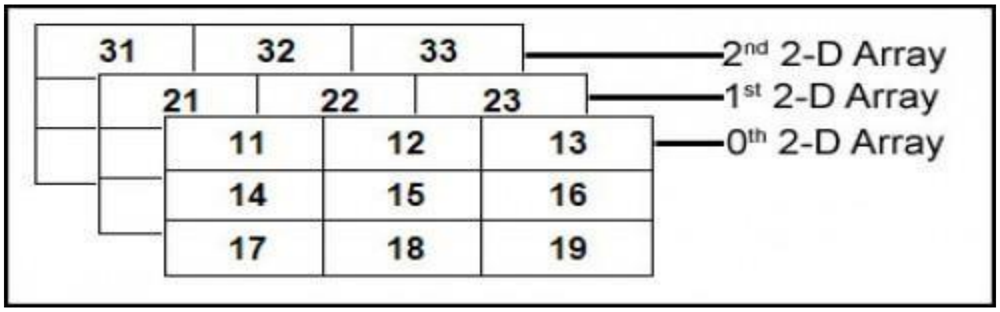
- Array **arr** is a five-dimensional array.
- It can hold 4500 floating-point elements (**5x6x5x6x5=4500**).

When it comes to holding multiple values, we would need to declare several variables. But a single array can hold thousands of values.

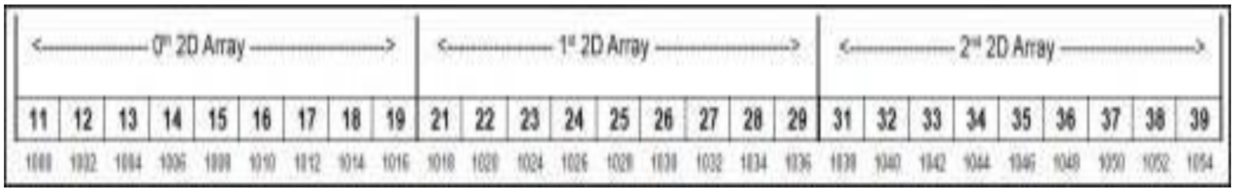
Explanation of a 3D Array

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

The diagram below shows a 3D array representation:



3D Array Conceptual View



3D array memory map.

STORING VALUES IN ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage".

An array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

we can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

1) **Full initailization**

 type array-name[size] = {list of values};

The values in the list are separated by commas. For example, the statement

int a[3] = { 1,2,3 }; will assign a[0]=1,a[1]=2,a[2]=3 with size 3.

2) **Partial initailization**

Compile time initialization may be partial. That is, the number of initializers may be less than declared size. In such cases. the remaining elements are inilialized to zero, if the array type is numeric and NULL if the type is char.

For example,

int number [5] = {10, 20}; will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0.

Similar the declaration. char city [5] = {‘B’}; will initialize the first element to 'B' and the remaining four to NULL.

3) **Without size**

The size may be omitted. In such cases, the compiler allocates enough space for all initialize elements.

For example, the statement int a[ ] = {1,1,1,1}; will declare the **a** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

4) **With character constants**

Character arrays may be initialized in a similar manner. Thus, the statement
char name [ ] = { 'J' , 'o ‘ , ‘ h ‘, ‘ n ‘, ‘\0’}; declares the name to be an array of five characters, initialized with the string "John" ending with the null character.

5) **With string**

we can assign the string literal directly as under: char name [] = "John"; In this size is five. Here null character need not be explicitly specified.

If we have more initializers than the declared size, the compiler will produce an error. That is, the statement int number [3] = {10, 20, 30, 40}; will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

```
int a [3] ;
scanf("%d%d%d", &a[0],&a[1],&a[2]);
(or)
int i,a [3] ;
for(i=0;i<3;i++)
        scanf("%d", &a[i]);
```

will initialize array elements with the values entered through the keyboard.

OPERATIONS ON ARRAYS

The following are the operations done on arrays

1) Traversing an array
2) Inserting an element in an array
3) Deleting an element in any array
4) Merging two arrays
5) Searching an element in an array

6) Sorting an array in ascending or descending order

1) Traversing an Array

Traversing an array means accessing each and every element of the array for a specific purpose.

1. Input and Output an array elements of size n

```
void main()
{
        int n,i,a[10];
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("The elements are");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
}
```

2. Biggest element in an array

```
void main()
{
        int n,i,a[10],big;
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        big=a[0];
        for(i=0;i<n;i++)
        {
                if(a[i]>big)
                        big=a[i];
        }
        printf("The biggest element is %d ",big);
}
```

3. **Sum of array elements**

```
void main()
{
        int n,i,a[10],sum=0;
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
                sum=sum+a[i];
        }
        printf("The sum  is %d ",sum);
}
```

4. **Sum of squares**

Otherwise the question for the same program will be given as under:

Write a program using a single-subscripted variable to evaluate the following expressions:

$$\text{Total} = \sum_{i=1}^{10} x_i^2$$

the values of x1 ,x2, ... are read from the terminal.

```
void main()
```

```
{
        int n,i,a[10],sum=0;
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
                sum=sum+a[i]*a[i];
        }
        printf("The sum  is %d ",sum);
}
```

5. **Add two array elements**

```
void main()
{
        int n,i,a[10],b[10],c[10];
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements of a array");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("enter the elements of b array");
        for(i=0;i<n;i++)
                scanf("%d",&b[i]);
        for(i=0;i<n;i++)
                c[i]=a[i]+b[i];
        printf("The resultant  array is");
        for(i=0;i<n;i++)
                printf("%d",c[i]);
    }
```

2) <u>Inserting an Element in an Array</u>

It means adding a new data element to an already existing array. Inserting an element at the end of array of **size n** is easy. If a is the array, a[n]=num and n=n+1.

Inserting in the middle of the array, say pos is also possible, for which we have to move all elements that have index greater than pos one position towards right and create space for the new element. Then we increment n by 1.

```
#include<stdio.h>
#include <stdio.h>
int insert(int a[],int n,int pos,int num);
int insert(int a[],int n,int pos,int num)
{
int i;
for(i=n-1;i>=pos;i--)
    a[i+1]=a[i];
a[pos]=num;
n++;
return n;
}
void main()
{
int n,i,a[10],pos,num;
printf("\n Enter size");
scanf("%d",&n);
printf("enter the elements");
for(i=0;i<n;i++)
```

```
        scanf("%d",&a[i]);
    printf("\n Enter num to insert");
    scanf("%d",&num);
    printf("\n Enter position to insert");
    scanf("%d",&pos);
    n=insert(a,n,pos,num);
    printf("The resultant  array is");
    for(i=0;i<n;i++)
        printf("\t %d",a[i]);
        }
```

3) ***Deleting an Element in an Array***

It means deleting a data element from an existing array. Deleting an element at the end of array of **size n** is easy. Simply decrement n as n=n-1.

Deletion in the middle of the array is also possible, for which we have to move all elements that have index greater than pos one position towards left. Then we decrement n by 1.

```
void main()
{
        int n,i,a[10],pos;
        printf("\n Enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\n Enter position to delete");
        scanf("%d",&pos);
        for(i=pos;i<n-1;i++)
                a[i]=a[i+1];
        n--;
```

```
            printf("The resultant  array is");
            for(i=0;i<n;i++)
                        printf("\t %d",a[i]);
}
```

4) Merging Two Arrays

Merging two arrays in a third array means first copying the contents of first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains contents of the first array followed by the contents of the second array.

```
void main()
{
            int a1[10],a2[10],a3[20],n1,n2,i,m,index=0;
            printf("\n Enter size of array1");
            scanf("%d",&n1);
            printf("\n Enter the elements of first array");
            for(i=0;i<n1;i++)
                        scanf("%d",&a1[i]);
            printf("\n Enter size of array2");
            scanf("%d",&n2);
            printf("\n Enter the elements of second array");
            for(i=0;i<n2;i++)
                        scanf("%d",&a1[i]);
            for(i=0;i<n1;i++)
            {
                        a3[index]=a1[i];
                        index++;
            }
            for(i=0;i<n2;i++)
            {
                    a3[index]=a2[i];
                    index++;
            }
            m=n1+n2;
            printf("The resultant  array is");
            for(i=0;i<m;i++)
                    printf("\t %d",a3[i]);
}
```

5) ***<u>Searching for a value in an array</u>***

Searching means to find whether a particular value is present in the array or not.

Two methods

1) Linear search
2) Binary Search

<u>Linear Search</u>

Linear search is a very simply search algorithm were search is made over all items one by one.

Every item is checked and if a match is found then that particular index is returned otherwise the search continues till the end of the array.

Algorithm

Linear Search

```
 Step 1: read n and initialize found=0
Step 2: read array elements A
Step 3: read the key item to be searched
Step 4: for i=0 till n-1
if A[i] = key
then
        print element found at index i;
found=1;
end if
end for
step 5: if found=0
then
        print element not found
step 6: stop
```

WORKING

Key = 4

```
    ↓
10      12      3       5       11      4       2       23      43      29

            ↓
10      12      3       5       11      4       2       23      43      29
             .
                                            .
                                            .
                                            ↓
10      12      3       5       11      4       2       23      43      29
```

program

```
#include<stdio.h>
int main( )

{
        int n,a[20],found=0,i,key;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements");
        for(i=0;i<n;i++)
```

```
	{
		scanf("%d",&a[i]);
	}
	printf("enter the key to be searched\n");
	scanf("%d",&key);
	for(i=0;i<n;i++)
	{
		if(a[i]==key)
		{
			printf("element found at position %d\n",i+1);
			found=1;
			break;
		}
	}
	if(found==0)
	{
		printf("element not found");
	}
}
```

Output:

```
1: enter the number of elements n 5
enter the array elements 12 18 2 4 21
enter the key to be searched
4
element found at position 4
2:
enter the number of elements n 5
enter the array elements 12 18 2 4 21
enter the key to be searched
100
element not found
```

Advantages

Linear search is simple and very easy to understand and implement

It does not require the data in the array to be stored in any particular order.

DISADVANTAGES

It is inefficient

For example, if we have 1000 elements and the key element is found at last index then it looks 1000 elements in order to find the last element.

Binary Search

Binary search looks for a particular item by comparing the middle most item of the sorted array. If a match occurs, then the index of item is returned.

If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.

Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub-array reduces to zero.

Algorithm

Binary Search

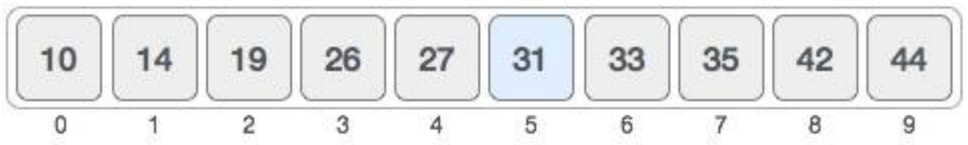
```
Step 1: read n and set found=0
Step 2: read array elements A in ascending Order
Step 3: read the key item to be searched
Step 4: set low=0
            set high=n-1
Step 5: while  key not found and low<=high
   do
           mid=(low+high)/2
           if A[mid] = key
```

```
            set found=1
            if a[mid]  < key
            set low = mid +1
           if a[mid] > key
            set high = mid -1
end while
 step 6: if found =1
then
            print element found at position mid+1
else
print element not found
step 7: stop.
```

working

For a binary search to work, it is mandatory for the target array to be sorted.

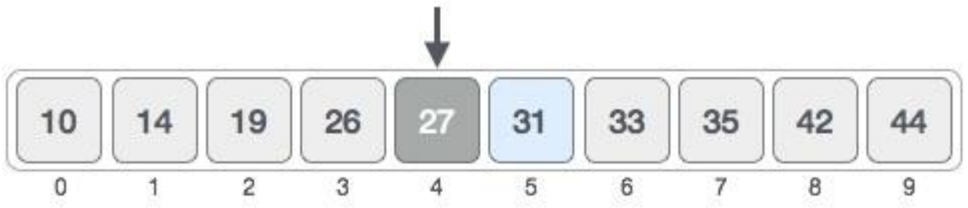
The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula

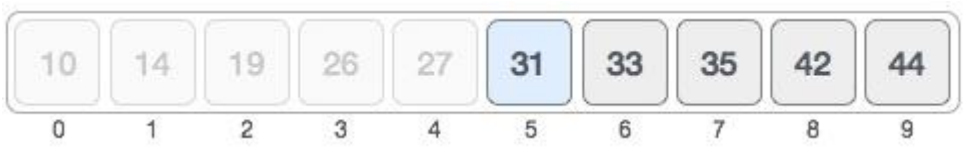
Mid=(low+high)/2

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



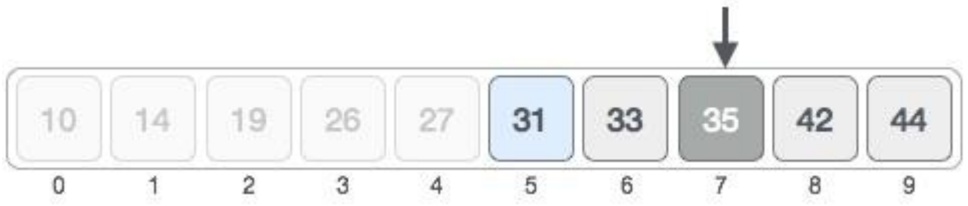
Now we compare the value stored at location 4, with the value being searched, i.e. 31.

We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

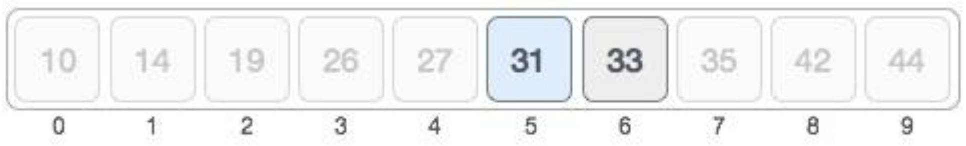


We change our low to mid + 1 and find the new mid value again.

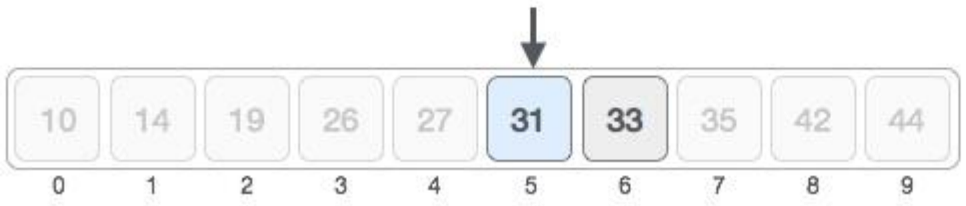
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



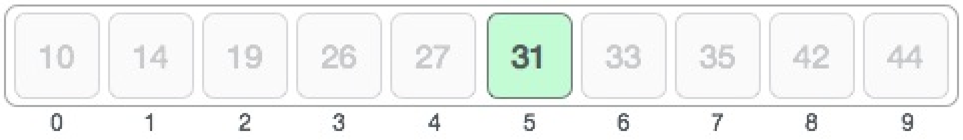
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Program

```
#include<stdio.h>
void main()
{
        int n,a[20],found=0,i,key,low,high,mid;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements in ascending order");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("enter the key to be searched\n");
        scanf("%d",&key);
        low=0;
        high=n-1;
        while(low<=high)
        {
                mid=(low+high)/2;

                if(a[mid]==key)
                {
                        printf("element found at position %d\n",mid+1);
                        found=1;
```

```
                    break;
                }
                else if(a[mid]<key)
                    low=mid+1;
                else
                    high=mid-1;
        }
        if(found==0)
        {
                printf("\n element not found at position”);
        }

}
```

Output:

```
enter the number of elements n10
enter the array elements in ascending order10 20 30 40 50 60 70 80 90 100
enter the key to be searched
70
element found at position 7
enter the number of elements n10
enter the array elements in ascending order10 20 30 40 50 60 70 80 90 100
enter the key to be searched
76
Element not found
```

Advantages

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Disadvantages

Binary search works only for sorted list.

6) ***Sorting an array in ascending or descending order***

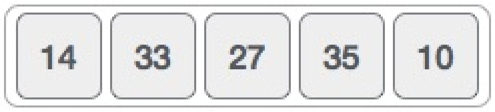
Arranging the elements of an array either in ascending or in descending order is called sorting an array.

Bubble sort

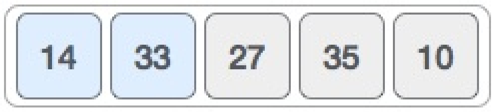
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Working

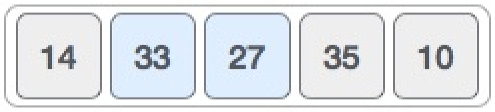
We take an unsorted array for our example.



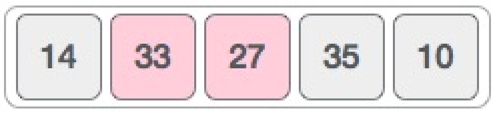
Bubble sort starts with very first two elements, comparing them to check which one is greater.



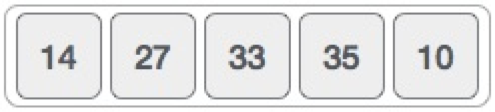
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



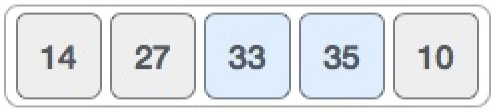
We find that 27 is smaller than 33 and these two values must be swapped.



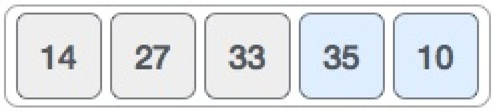
The new array should look like this –



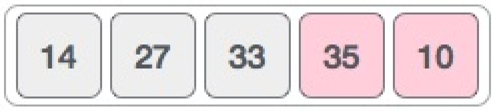
Next we compare 33 and 35. We find that both are in already sorted positions.



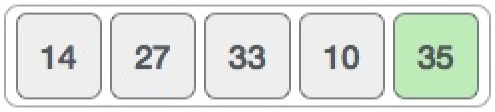
Then we move to the next two values, 35 and 10.



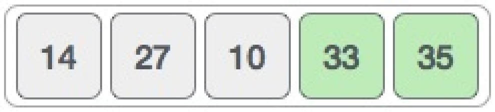
We know then that 10 is smaller 35. Hence they are not sorted.



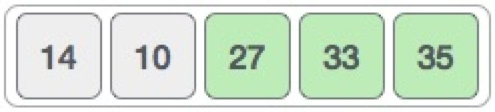
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



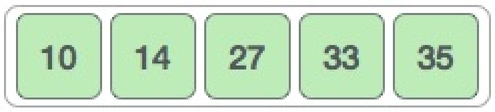
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Algorithm

```
Step 1: read n
Step 2: read array elements A in unsorted order
Step 3: print unsorted array elements A
Step 4: for i=0 till n-1
do
        for j=0 till n-i-1
do
if A[j] > A[j+1]
        swap (A[j] , A[j+1]
end if
end for
end for
step 6: print the sorted array elements A
step 7: stop
```

program

```
#include<stdio.h>
int main()
{
        int n,a[20],i,j, temp;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements in any order");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("the unsorted elements are \n");
        for(i=0;i<n;i++)
        {
                printf ("%d\t",a[i]);
        }
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }

                }
        }
        printf("\n the sorted elements are\n");
        for(i=0;i<n;i++)
        {
                printf ("%d\t",a[i]);
        }
}
```

output

```
        enter the number of elements n 5
```

```
enter the array elements in any order 50 20 40 10 30
the unsorted elements are
50      20      40      10      30
 the sorted elements are
10      20      30      40      50
```

Advantage

Easy to understand

Easy to implement

In place, no external memory is needed

Disadvantage

It does more element assignments than its counterpart.

Selection Sort

***Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.*

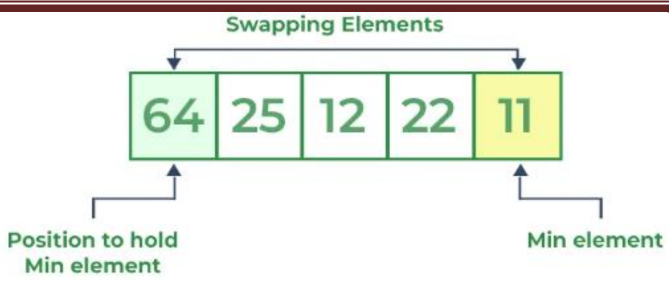
The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

How does Selection Sort Algorithm work?

Lets consider the following array as an example: ***arr[] = {64, 25, 12, 22, 11}***

First pass:

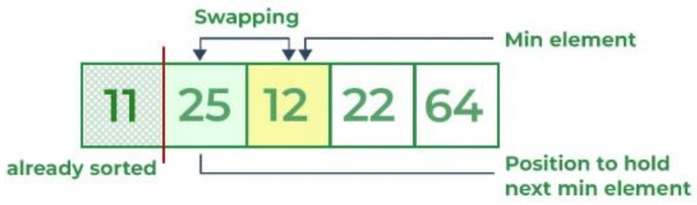
- *For the first position where 64 is present, traverse the rest of the array sequentially. After traversing whole array it is clear that **11** is the lowest value.*
- *Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*



Selection Sort Algorithm | Swapping 1st element with the minimum in array

Second Pass:

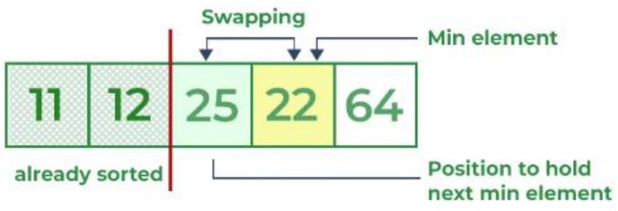
- *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*
- *After traversing, we found that* ***12*** *is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*



Selection Sort Algorithm | swapping i=1 with the next minimum element

Third Pass:

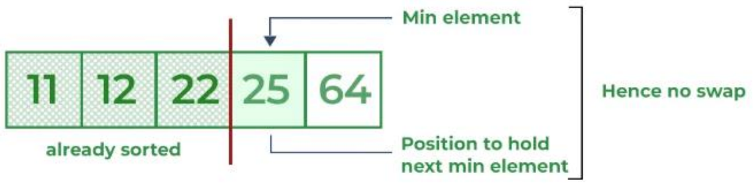
- *Now, for third place, where* ***25*** *is present again traverse the rest of the array and find the third least value present in the array.*
- *While traversing,* ***22*** *came out to be the third least value and it should appear at the third place in the array, thus swap* ***22*** *with element present at third position.*



Selection Sort Algorithm | swapping i=2 with the next minimum element

Fourth pass:

- *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
- *As **25** is the 4th lowest value hence, it will place at the fourth position.*



Selection Sort Algorithm | swapping i=3 with the next minimum element

- *At last the largest value present in the array automatically get placed at the last position in the array*
- *The resulted array is the sorted array.*

11
12
22
25
64
Sorted array

Selection Sort Algorithm | Required sorted array

```
#include <stdio.h>
void selectionsort(int a[],int n);
void selectionsort(int a[],int n)
{
        int i,j,pos,temp;
        for(i=0;i<n-1;i++)
        {
                pos=i;
                for(j=i+1;j<n;j++)
                {
                        if(a[j]<a[pos])
                                pos=j;
                }
                if(i!=pos)
                {
                        temp=a[i];
                        a[i]=a[pos];
                        a[pos]=temp;
                }
        }
}
int main()
{
        int a[100],n,i;
        printf("enter the number of elements\n");
        scanf("%d",&n);
        printf("enter the array elements\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        selectionsort(a,n);
        printf("after sorting\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        return 0;
}
```

OPERATIONS ON TWO-DIMENSIONAL ARRAYS

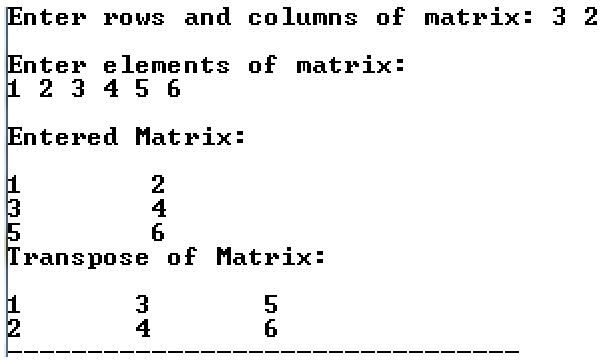
Two-dimensional arrays can be used to implement the mathematical concept of matrices. In mathematics, a matrix is a grid of numbers, arranged in rows and columns. Thus, using two dimensional arrays, we can perform the following operations on an m×n matrix:

1. ***Transpose*** Transpose of an m x n matrix A is given as a n x m matrix B, where $B_{i,j} = A_{j,i}$.

```
#include <stdio.h>
int main()
```

```
{
        int a[10][10],b[10][10],m,n,i,j;
        printf("Enter rows and columns of matrix: ");
        scanf("%d%d",&m,&n);
        printf("\nEnter elements of matrix:\n");
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                scanf("%d", &a[i][j]);
                }
        }
        printf("\nEntered Matrix: \n");
        for(i=0; i<m; i++)
        {
                printf("\n");
                for(j=0; j<n; j++)
                {
                printf("%d\t ", a[i][j]);
                }
        }
        for(i=0; i<n; i++)
        {
                for(j=0; j<m; j++)
                {
                b[i][j] = a[j][i];
                }
        }
```

```
                printf("\nTranspose of Matrix:\n");
                for(i=0; i<n; i++)
                {
                        printf("\n");
                        for(j=0; j<m; j++)
                        {
                        printf("%d\t",b[i][j]);
                        }
                }
                return 0;
        }
```



```
Enter rows and columns of matrix: 3 2

Enter elements of matrix:
1 2 3 4 5 6

Entered Matrix:

1        2
3        4
5        6
Transpose of Matrix:

1       3       5
2       4       6
--------------------------------
```

2. ***Sum*** Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing: $C_{i,j} = A_{i,j} + B_{i,j.}$

```
        void main()
        {
                int m,n,i,j,a[10][10],b[10][10,c[10][10];
                printf("Enter rows and columns of matrix: ");
                scanf("%d%d",&m,&n);
                printf("\nEnter elements of matrix A:\n");
                for(i=0; i<m; i++)
                {
```

```
		for(j=0; j<n; j++)
		{
			scanf("%d", &a[i][j]);
		}
	}
	printf("\nEnter elements of matrix B:\n");
	for(i=0; i<m; i++)
	{
		for(j=0; j<n; j++)
		{
			scanf("%d", &b[i][j]);
		}
	}
	for(i=0; i<m; i++)
	{
		for(j=0; j<n; j++)
		{
			c[i][j]=a[i][j]+b[i][j];
		}
	}
	printf("\nResultant Matrix: \n");
	for(i=0; i<m; i++)
	{
		printf("\n");
		for(j=0; j<n; ++j)
		{
			printf("%d\t ", c[i][j]);
		}
```

```
                }
        }
```

3. ***Product*** Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, m x n matrix A can be multiplied with a p x q matrix B if n=p. The dimension of the product matrix is m x q. The elements of two matrices can be multiplied by writing: $C_{i,j} = \sum A_{i,k} \times B_{k,j}$ for k=0 to n-1.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
        int a[10][10],b[10][10],c[10][10];
        int m,n,p,q,i,j,k;
        printf("Enter the order of the matrix A\n");
        scanf("%d%d",&m,&n);
        printf("Enter the order of the matrix B\n");
        scanf("%d%d",&p,&q);
        if(n!=p)
        {
        printf("\n multiplication not possible");
        exit(0);
        }
        else
        {
      printf("Enter the elements of matrix A...\n");
           for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
```

```
                scanf("%d",&a[i][j]);
                }
        }
          printf("Enter the elements of matrix B...\n");
          for(i=0;i<p;i++)
             {
                for(j=0;j<q;j++)
                {
                scanf("%d",&b[i][j]);
                }
        }
     printf("\n Matrix A \n");
          for(i=0;i<m;i++)
                {
                   printf("\n");
                         for(j=0;j<n;j++)
                                  {
                                  printf("%d\t",a[i][j]);
                         }
                   }
                printf("\n Matrix B \n");
          for(i=0;i<p;i++)
                {
                   printf("\n");
                         for(j=0;j<q;j++)
                                  {
                                  printf("%d\t",b[i][j]);
                         }
```

```
                        }

                  for(i=0;i<m;i++)
                        {
                              for(j=0;j<q;j++)
                        {
                                    c[i][j]=0;
                                    for(k=0;k<n;k++)
                                    c[i][j]=c[i][j]+a[i][k]*b[k][j];
                        }
                     }
                  printf("\n Product of A and B matrices : MATRIX C\n");
                  for(i=0;i<m;i++)
                        {
                           printf("\n");
                              for(j=0;j<q;j++)
                                    {
                                    printf("%d\t",c[i][j]);
                              }
                        }
            }
         }
```

Test cases

Test No	Input Parameters	Expected Output	Obtained Output
1	Enter the order of the matrix A 2 2 Enter the order of the matrix B 2 2 Enter the elements of matrix A... 1 2 3 4	MATRIX A 1 2 3 4 MATRIX B	MATRIX A 1 2 3 4 MATRIX B

	Enter the elements of matrix B... 5 6 7 8	5 6 7 8 Product of A and B matrices : MATRIX C 19 22 43 50	5 6 7 8 Product of A and B matrices : MATRIX C 19 22 43 50
2	Enter the order of the matrix A 2 3 Enter the order of the matrix B 2 2	multiplication not possible	multiplication not possible

1.3 Data Structures Operations

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.

5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending upon the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
   a. Compile-time
   b. Run-time

      For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

1.4 The Abstract Data type (ADT)

There are two ways of viewing the data structure:

- **Mathematical/ Logical/ Abstract models/ Views:** The data structure is the way of organizing the data that requires some protocols or rules. These rules need to be modeled that come under the logical/abstract model.

- **Implementation:** The second part is the implementation part. The rules must be implemented using some programming language.

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

ADT Array is

Objects: A set of pairs <index,value> where for each value of index there is a value from the set item. Index is a finite ordered set of one or more dimensions, for example, {0, …, n-1} for one dimension, {(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)} for two dimensions etc.

Functions:

For all A $\in$ Array, i $\in$ index, x $\in$ item, j, size $\in$ integer

Array Create(j,list) – **return** an array of j dimensions where list is a j-tuple whose ith element is the size of the ith dimension. All the Items are initially undefined.

Item Retrieve(A,i) – if *(i $\in$ index)* ***return*** *the item associated with index value i in array A* ***else return*** *error.*

Array Store(A, i, x) – **if**(i in index)

return an array that is identical to array A except the new pair <i,x> has been inserted **else return** error.

1.5 REVIEW OF POINTERS AND DYNAMIC MEMORY ALLOCATION:

1.5.1 REVIEW OF POINTERS

- A pointer is variable that contains the memory location of another variable.

- Suppose, we assign the address of quantity to a variable p.
- The link between the variables p and quantity can be visualized as shown in Fig. The address of p is 5048.

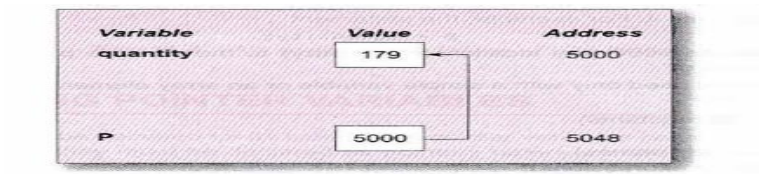


Fig : Pointer Variable

<u>Finding the size of various data types</u>

```
int main()
{
        printf("\n size of integer is %d bytes”,sizeof(int));
        printf("\n size of floating point number is %d bytes”,sizeof(float));
        printf("\n size of double is %d bytes”,sizeof(double));
        printf("\n size of character is %d byte”,sizeof(char));
        return 0;
}
```

<u>Output</u>

```
                size of integer is 4 bytes
                size of floating point number is 4 bytes
                size of double is 8 bytes
                size of character is 1 byte
```

DECLARING POINTER VARIABLES

- The general syntax of declaring pointer variables can be given as below.

  data_type *ptr_name;

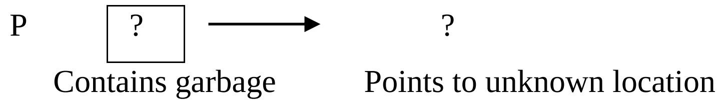
- Here, data_type is the data type of the value that the pointer will point to.
- For example,

```
int *pnum;
char *pch;
```

```
float *pfnum;
```

- In each of the above statements, a pointer variable is declared to point to a variable of the specified data type.
- The declarations cause the compiler to allocate memory locations for the pointer variables p.
- Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:

```
int *p ;
```



<u>Initialization Of Pointer Variables</u>

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.

<u>Example:</u>

```
int x=10;
int *ptr;
ptr =&x;
```

We can also combine **the initialization with the declaration**. That is,

```
int *ptr = &x;
```

is allowed. The only requirement here is that the variable x must be declared before the initialization takes place.

```
#include<stdio.h>
int main()
{
    int num, *ptr;
    ptr = &num;
    printf("\n Enter the number : ");
    scanf("%d", &num);
```

```
        printf("\n The number that was entered is : %d", *ptr);
        printf("\n The address of number that was entered is : %d", ptr);
        return 0;
}
```

Output

```
Enter the number : 10
The number that was entered is : 10
The address of number that was entered is : 6487572
```

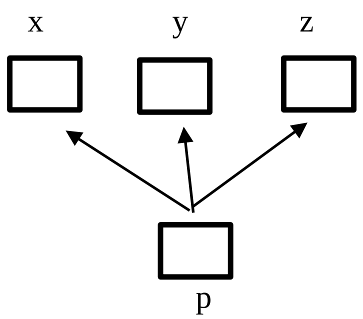
What will be the value of *(&num)?

It is equivalent to simply writing num.

POINTER FLEXIBILITY Pointers are flexible.

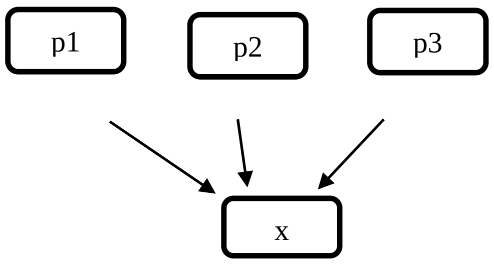
- We can make the same pointer to point to different data variables in different Statements. Example;

```
int x, y, z, *p;
p = &x;
…
p = &y;
…
p = &z;
```



- We can also use different pointers to point to the same data variable. Example;

```
int x;
int *pl = &x;
int *p2 = &x;
int *p3 =&x;
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;   / *absolute address */
```

Differences between normal variable and a pointer variable

Normal variable	**Pointer variable**
Stores values	Stores address
Not dereferenced to print the value through variable	Dereferenced with the help of * the indirection operator to print the value(content) through pointer
Dereferenced with the help of & the address of operator to print the address through variable	Not dereferenced to print the address through pointer

Program to add 2 numbers using pointers

```
void main()
{
        int a,b,c,*p,*q;
        printf("enter 2 numbers");
        scanf("%d%d",&a,&b);
        p=&a;
        q=&b;
```

```
    c=*p + *q;
    printf("sum=%d",c);
}
```

TYPES OF POINTERS

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.
- To declare a null pointer, we may use the predefined constant NULL
- We can write int *ptr = NULL;
- We can always check whether a given pointer variable stores the address of some variable or contains NULL by writing,
  ```
  if (ptr == NULL)
  {
      Statement block;
  }
  ```
- We may also initialize a pointer as a null pointer by using the constant 0
  ```
  int *ptr;
  ptr = 0; This is a valid statement in C
  ```

Generic Pointers

- 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.
- It is declared like a normal pointer variable but using the void keyword as the pointer's data type.
- For example, void *ptr;
- In C, since we cannot have a variable of type void, the void pointer will therefore not point to any data and, thus, cannot be dereferenced.

- We need to cast a void pointer to another kind of pointer before using it.
- Generic pointers are often used when you want a pointer to point to data of different types at different times.

```
#include<stdio.h>
int main()
{
        int x=10;
        char ch = ‘A’;
        void *gp;
        gp = &x;
        printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
        gp = &ch;
        printf("\n Generic pointer now points to the character= %c", *(char*)gp);
        return 0;
}
```

Output

```
Generic pointer points to the integer value = 10
Generic pointer now points to the character = A
```

Double Pointers (Chain of Pointers or Pointers to Pointers)

Double pointers is used to make a pointer to point to another pointer, thus creating a chain of pointers.

```
void main()
{
        int a=10;
        int *p1,**p2;
        p1=&a;
        p2=&p1;
        printf(“%d”,**p2);
}
```

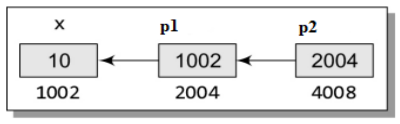


Figure Pointer to pointer

Output

10

Here p2 contains address of p1. This is also known as multiple indirections.

int **p2; tells the compiler that p2 is a pointer to a pointer of int type.

POINTER ARITHMETIC

POINTER ARITHMETIC (ADDRESS ARITHMETIC) POINTERS INCREMENT AND SCALE FACTOR

Let us consider starting address as 8000. Then the operation p++ is done to point to next subsequent element:

1. int

   p++ (i.e) p=p+1 address is 8002.
2. char

   p++ (i.e) p=p+1 address is 8001.
3. float

   p++ (i.e) p=p+1 address is 8004.
4. double

   p++ (i.e) p=p+1 address is 8008.

Let us consider starting address as 8008. Then the operation p-- is done to point to previous element:

1. int

   p-- (i.e) p=p-1 address is 8006.
2. char

   p-- (i.e) p=p-1 address is 8007.
3. float

   p-- (i.e) p=p-1 address is 8004.
4. double

   p-- (i.e) p=p-1 address is 8000.

Let us consider starting address as 8000. Then the operation p+5 is done to point to the fifth element from that element (i.e) element at index 5:

1. int

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(int))=8000+5*2

   address is 8010.

2. char

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(char))=8000+5*1

   address is 8005.

3. float

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(float))=8000+5*4

   address is 8020.

4. double

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(double))=8000+5*8

   address is 8040.

Let us consider starting address as 8040. Then the operation p-5 is done to point to the five elements before that element:

1. int

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(int))=8040-5*2

   address is 8030.

2. char

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(char))=8040-5*1

   address is 8035.

3. float

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(float))=8040-5*4

   address is 8020.

4. double

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(double))=8040-5*8

   address is 8000.

When we increment a pointer, its value is increased by the length of the data type that its points. This length is called scale factor.

- ➔ Scale factor for **int** in a 16-bit machine is **2 bytes** (i.e) the size of the datatype.
- ➔ Scale factor for **char** in a 16-bit machine is **1 byte** (i.e) the size of the datatype.
- ➔ Scale factor for **float** in a 16-bit machine is **4 bytes** (i.e) the size of the datatype.
- ➔ Scale factor for **double** in a 16-bit machine is **8 bytes** (i.e) the size of the datatype.

POINTERS AND ARRAYS

int a[5]={10,20,30,40,50};

Suppose the base address is 8000, each integer requires two bytes.

Here **a** refers to starting address. Also, &a[0] refers to the starting address.

Initialization

```
int a[5];
int *p;
p=a;
(or)
int a[5];
int *p;
p=&a[0];
```

Now, we can access the next element in the array by using p++.

```
p=&a[0] 8000
p+1=&a[1] 8002
p+2=&a[2] 8004
p+3=&a[3] 8006
p+4=&a[4] 8008
```

Program

```
void main()
{
        int a[10],n,i,*p;
        printf("enter n");
        scanf("%d",&n);
        printf("enter elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        p=a;
        printf("elements are");
        for(i=0;i<n;i++)
                printf("%d\t",*(p+i));
}
```

```
        printf("elements are");
        for(i=0;i<n;i++)
        {
                printf("%d\t",*p);
                p++;
        }
```

OUTPUT

```
enter n 5
enter elements 1 2 3 4 5
elements are 1 2 3 4 5
```

PASSING ARGUMENTS TO FUNCTION USING POINTERS

- When an array is passed to a function as an argument, only the address of the first element of the array is passed. It works like call by reference.
- Similarly, we can pass the address of a normal variable as an argument to function- It works like functions that return multiple values

Example : To swap two numbers

```
        void swap(int  *p,int  *q);
        void main()
        {
                int a,b;
                printf("enter two numbers");
                scanf("%d%d",&a,&b);
                printf("before swapping");
                printf("a=%d,b=%d",a,b);
                swap(&a,&b);
                printf("after swapping");
                printf("a=%d,b=%d",a,b);
        }
        void swap(int  *p,int  *q)
        {
                int t;
                t=*p;
                *p=*q;
                *q=t;
        }
```

Output:

```
enter two numbers 5 4
before swapping
a=5,b=4
after swapping
a=4,b=5
```

Troubles with Pointers (Drawbacks of Pointers)

In most of the cases, compiler may not detect the error and produce unexpected results, when we use pointers. The output does not give us a clue regarding where we went wrong.

Some possible errors

- Assigning values to uninitialized pointers
  ```
          int m=10,*p;
  *p=m;
  ```
- Assigning a value to a pointer
  ```
          int m=10,*p;
  p=m;
  ```
- Not dereferencing to print the value
  ```
          int m=10,*p;
          p=&m;
  ```

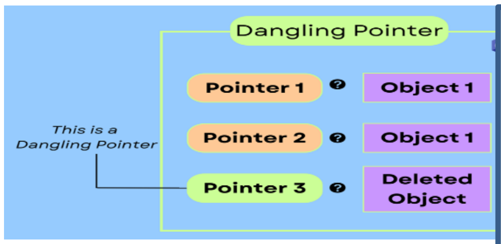
```
printf("%d",p);
```

- Assigning the address of uninitialized variable.

```
int m,*p;
p=&m;
```

- Comparing pointers that point to different objects

```
char name1[20],name2[20];
char *p1=name1;
char  *p2=name2;
if(p1>p2)
```



A dangling pointer in C is a pointer that points to a memory location that has been deallocated or is no longer valid. Dangling pointers can cause various problems in a program, including segmentation faults, memory leaks, and unpredictable behavior.

1.5.2 DYNAMIC MEMORY ALLOCATION FUNCTIONS

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

malloc()	Allocates single block of requested memory.
calloc()	Allocates multiple block of requested memory.
realloc()	Reallocates the memory occupied by malloc() or calloc() functions.
free()	Frees the dynamically allocated memory.

Differences between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation

memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

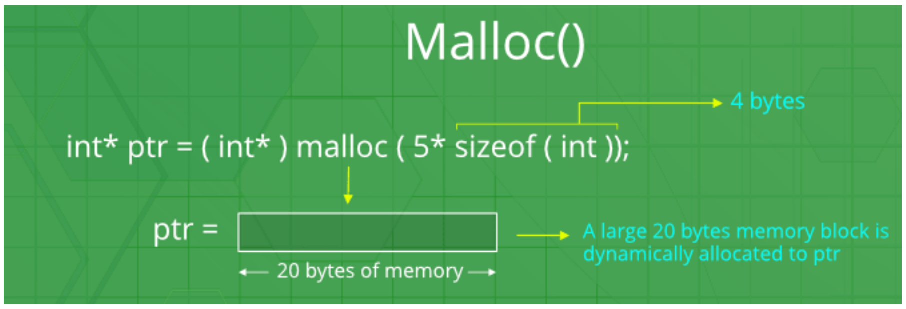
malloc() function in C

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Ex - int *ptr = (int*) malloc(5*sizeof(int));


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory

calloc() function in C

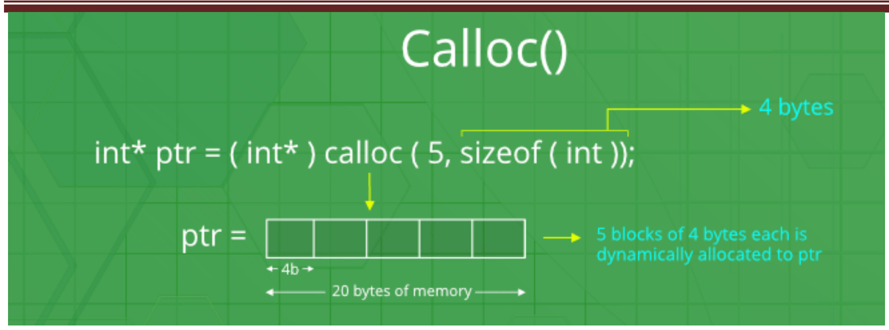
The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

ptr=(cast-type*)calloc(number, byte-size)

Ex - int *ptr = (int*) calloc(5,sizeof(int));

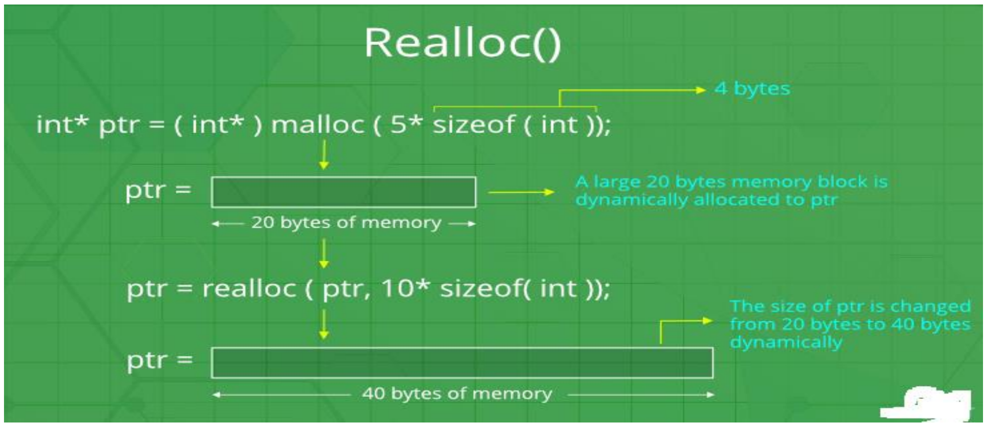

Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr

realloc() function in C

If memory is not sufficient for malloc() or calloc(), we can reallocate the memory by realloc() function. In short, it changes the memory size.

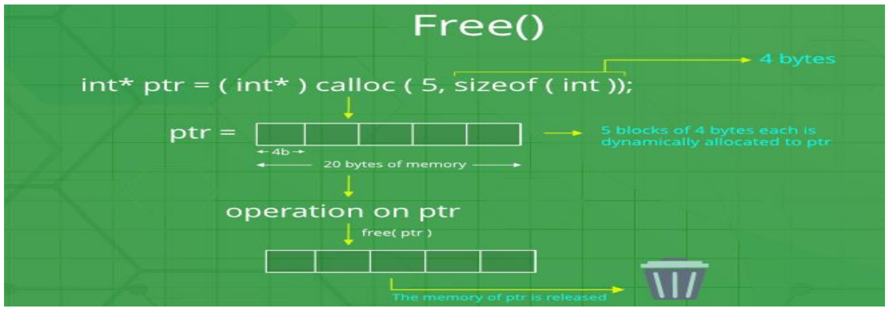
ptr=realloc(ptr, new-size)

Ex – ptr=realloc(ptr,10*sizeof(int));


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

free() function in C

The memory occupied by malloc () or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit. free(ptr)


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr
operation on ptr
free(ptr)
The memory of ptr is released

```
  #include <stdio.h>
  #include <stdlib.h>
  int main() {
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Enter Elements:\n");
    for(i = 0; i < n1; ++i)
      scanf("%d",ptr + i);
    printf("\nEnter the new size: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2 * sizeof(int));
   printf("Enter Elements for new size:\n");
    for(i = 0; i < n2; ++i)
      scanf("%d",ptr + i);
     printf("Elements are:\n");
    for(i = 0; i < n2; ++i)
      printf("%d\t",*(ptr + i));
    free(ptr);
    return 0;
}
```

1.6 Programming Examples (Dynamic 1D array, 2D array),

Dynamic 1D array

```
  #include <stdio.h>
  #include <stdlib.h>
  int main() {
    int *a, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);
   a = (int*) malloc(n1 * sizeof(int));
  printf("\nEnter the new size: ");
    scanf("%d", &n2);
  a = realloc(a, n2 * sizeof(int));
  printf("Enter the elements\n");
    for(i = 0; i < n2; i++)
      scanf("%d", &a[i]);
  printf("After Realloc\n");
    for(i = 0; i < n2; ++i)
      printf("%d\n", a[i]);
    free(a);
   return 0;
}
```

Dynamic 2D array

```
  #include <stdio.h>
  #include<stdlib.h>
  int main()
  {
    int r = 3, c = 4, i, j, count;
    int* arr[r];
    for (i = 0; i < r; i++)
        arr[i] = (int*)malloc(c * sizeof(int));
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // Or *(*(arr+i)+j) = ++count
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);
    for (int i = 0; i < r; i++)
        free(arr[i]);
    return 0;
}
```

1.7 Review of Structures

A structure is a collection of elements of different data type under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure.

Arrays Vs Structures

Arrays	**Structures**
Collection of related data items of same type	Collection of elements of different data type
Array is a derived data type	Structures is a user defined data type
To use an array, we must first declare it	To use a structure we must first define and then declare it

Structure Definition & Declaration

A structure is defined using the keyword struct followed by the structure name.

A structure type is generally defined by using the following syntax:

```
struct struct–name
{
        data_type var–name;
        data_type var–name;
        ..............
};
```

For example,

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

Now the structure has become a user-defined data type.

Each variable name declared within a structure is called a member of the structure.

The structure definition (structure template), however, does not allocate any memory or consume storage space.

Like any other data type, memory is allocated for the structure when we declare a variable of the structure. For example, we can define a variable of student by writing: **struct student stud1;**

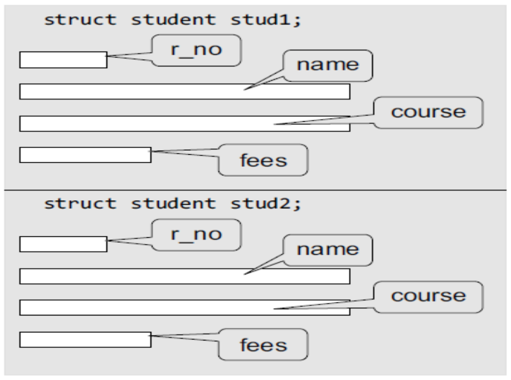
Here, struct student is a data type and stud1 is a variable.

In the following syntax, the variables are declared at the time of structure definition.

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
```

```
                float fees;
        } stud1, stud2;
```

When we declare variables of the structure, separate memory is allocated for each variable.



Memory allocation for a structure variable

```
Declare a structure to store
customer information.
struct customer
{
        int cust_id;
        char name[20];
        char address[20];
};
```

```
Declare a structure to store
information of a particular date.
struct date
{
        int day;
        int month;
        int year;
};
```

Program

```
void main()
{
    struct book
    {
            char title[20];
                char author [15];
                int pages;
                float price;
        }b1;
        printf("\n Enter Values");
        scanf("%s%s%d%f",b1.title,b1.author,&b1.pages,&b1.price);
        printf("\n%s\t%s\t%d\t%f",b1.title,b1.author,b1.pages,b1.price);
}
```

Output

```
  Enter Values
 Networks balagurusamy 250 550
```

Networks balagurusamy 250 550

Initialization of Structures

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables. Note that the compile-time initialization of a structure variable must have the following elements:

(1)
```
struct book
{
        char title[20];
        char author[15];
        int pages;
        float price;
}b1={“ansi c”,”balaguruswamy”,350,550};
```

(2)
```
struct book
{
        char title[20];
        char author[15];
        int pages;
        float price;
};
struct book b1={“ansi c”,”balaguruswamy”,350,550};
struct book b2={“basic c”,”balaguruswamy”,350,350};
```

(3) Partial initialization
```
struct book
{
        char title[20];
        char author[15];
        int pages;
        float price;
}b1={"ansi c"};
```

Uninitialized members will be automatically assigned to

- Zero for integer and floating point numbers.
- '\0' for characters and strings

Typedef Declarations

The typedef (derived from type definition) keyword enables the programmer to create a new data type name by using an existing data type.

By using typedef, no new data is created, rather an alternate name is given to a known data type.

Syntax for typedef:

typedef existing datatype new datatype;

```
typedef int integer;
integer a,b,c;
```

typedef in structures

```
typedef struct student
{
        int rollno;
        char name[20];
}stud;
stud s1,s2,s3;
```

```
struct student
{
        int rollno;
        char name[20];
};
typedef struct student stud;
```

Here stud is the type name. We have not written struct student s1,s2,s3.

Program: use typedef to input and output one employee details consisting employee number and name.

```
void main()
{
        struct employee
        {
                int empno;
                char name[20];
```

```
    };
    typedef struct employee emp;
    emp e1;
        printf("\n Enter Values");
        scanf("%d%s”,&e1.empno,e1.name);
        printf("\n%d\t%s”,e1.empno,e1.name);
    }
```

Output

```
 Enter Values
201 balagurusamy
201 balagurusamy
```

NESTED STRUCTURES (STRUCTURES WITHIN STRUCTURES)

A structure can be placed within another structure, i.e., a structure may contain another structure as its member. A structure that contains another structure as its member is called a *nested structure*. It means nesting of structures. Also called NESTED STRUCTURES.

It can be implemented in two ways:

First method: Placing the definition of a structure within the definition of another structure.

```
struct student
{
    int rollno;
    char name[20];
    struct dateofbirth
    {
        int day;
        char month[10];
        int year;
        }dob;
}s1;
```

Second method: Placing the declaration (already defined structure) of a structure within the definition of another structure.

```
struct dateofbirth
{
            int day;
    char month[10];
    int year;
    };
struct student
{
    int rollno;
    char name[20];
    struct dateofbirth dob;
}s1;
```

The members can be accessed as s1.rollno,s1.name,s1.dob.day,s1.dob.month,s1.dob.year. The innermost structure in a nested structure can be accessed by chaining all the concerned structure variables. We can also nest more than one structure.

Program:

```
void main()
{
        struct student
        {
            int rollno;
            char name[20];
            struct dateofbirth
            {
                  int day;
                  char month[10];
```

```
                int year;
                }dob;
     }s1;
        printf("\n Enter Values");
        scanf("%d%s%d%s%d”,&s1.rollno,s1.name,&s1.dob.day,s1.dob.month,&s1.dob.year);
        printf("%d\t%s\t%d\t%s\t%d”,s1.rollno,s1.name,s1.dob.day,s1.dob.month,s1.dob.year);
}
```

Output

```
Enter values
1 neha 16 jan 2010
1 neha 16 jan 2010
```

ARRAYS OF STRUCTURES

Let us first analyse where we would need an array of structures.

In a class, we do not have just one student. But there may be at least 30 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures.

The general syntax for declaring an array of structures can be given as,

```
        struct struct_name
        {
                data_type member_name1;
                data_type member_name2;
                data_type member_name3;
                .......................
        };
        struct struct_name struct_var[index];
```

Example program: Array of structures print the names of students who have got more than 75.

<u>Program</u>

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

```
struct student
{
   int rollno;
   char name[20];
   float marks;
};
int main()
{
   int i,n;
   struct student s[10];
   printf("\nEnter the number of student details");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
           printf("\nenter the %d student details",i+1);
           printf("\n enter roll number:");
           scanf("%d",&s[i].rollno);
           printf("\n enter student name");
           scanf("%s",s[i].name);
           printf("\n enter the marks:");
           scanf("%f",&s[i].marks);
   }
   printf("\nStudent details are\n");
   printf("\nRollno\t\tName\t\tMarks\n");
   for(i=0;i<n;i++)
       printf("%d\t\t%s\t\t%f\n",s[i].rollno,s[i].name,s[i].marks);
   printf("\n students scoring above 75 marks\n");
   for(i=0;i<n;i++)
```

```
  {
          if(s[i].marks>=75)
          {
                  printf("%s\t",s[i].name);
          }
  }
  return 0;
  }
```

STRUCTURES AND FUNCTIONS

There are three methods in which we pass a structure to a function as an argument.

1. We can pass each member of a structure. It is not efficient, so we do not use it.
2. We can pass a copy of entire structure to a called function (Call by value)
3. We can pass the entire structure by passing its address (call by reference)

Passing Individual Members

To pass any individual member of a structure to a function, we refer to the individual members.

```
#include <stdio.h>
typedef struct POINT
{
        int x;
        int y;
}POINT;
void display(int, int);
int main()
{
        POINT p1 = {2, 3};
        display(p1.x, p1.y);
        return 0;
}
```

```
void display(int a, int b)
{
        printf(" The coordinates of the point are: %d %d", a, b);
}
```

Output

```
The coordinates of the point are: 2 3
```

Call by Value (Passing the Entire Structure)

In the function call, the structure variable is used. The argument in the function header and declaration is to be declared as a structure variable to receive the entire structure.

Program:

```
void display(struct student stu);
      struct student
      {
          int rollno;
          char name[20];
      };
        void main()
        {
            struct student s1;
            printf("\n Enter Values");
            scanf("%d%s”,&s1.rollno,s1.name);
            display(s1);
        }
        void display(struct student stu)
        {
                printf("%d\t%s”,stu.rollno,stu.name);
        }
```

Output

```
Enter values
1 neha
1 neha
```

Call by Reference (Passing Structures through Pointers)

In the function call, the address of the structure variable is used. The argument in the function header and declaration is to be declared as a pointer to a structure variable.

➔ is the member selection operator. (minus sign followed by greater than symbol)

Program:

```
void display(struct student  *stu);
      struct student
      {
         int rollno;
         char name[20];
      };
        void main()
        {
            struct student s1;
            printf("\n Enter Values");
                scanf("%d%s”,&s1.rollno,s1.name);
                display(&s1);
        }
        void display(struct student  *stu)
        {
                printf("Roll number=%d\t”,stu->rollno);
                printf("Name is %s”,stu->name);
        }
```

Output

```
Enter values
```

1 neha

SELF-REFERENTIAL STRUCTURES

Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
        int val;
        struct node *next;
};
```

Here, the structure node will contain two types of data: an integer val and a pointer next. Actually, self-referential structure is the foundation of other data structures.

Purpose: It is used in linked lists, trees, and graphs.

UNIONS

Similar to structures, a union is a collection of variables of different data types. The difference between a structure and a union is that in case of unions, we can only store information in one field at any one time. When a new value is assigned to a field, the existing data is replaced with the new data. Thus, unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

Declaring a Union

The syntax for declaring a union is the same as that of declaring a structure. The differences between a structure and a union is that in case of unions, we can only store information in one field at any one time. Other difference is that instead of using the keyword struct, the keyword union would be used.

The syntax for union declaration can be given as

```
union union–name
{
        data_type var–name;
        data_type var–name;
        ..................
};
```

Again the typedef keyword can be used to simplify the declaration of union variables. The most important thing to remember about a union is that the size of a union is the size of its largest field. This is because sufficient number of bytes must be reserved to store the largest sized field.

Accessing a Member of a Union

A member of a union can be accessed using the same syntax as that of a structure. To access the fields of a union, use the dot operator (.), i.e., the union variable name followed by the dot operator followed by the member name.

Initializing Unions

The difference between a structure and a union is that in case of a union, the fields share the same memory space, so new data replaces any existing data.

```
#include <stdio.h>
typedef struct POINT1
{
        int x, y;
}POINT1;
typedef union POINT2
{
        int x,y;
}POINT2;
int main()
{
        POINT1 P1 = {2,3};
        POINT2 P2;
        printf("\n The coordinates of P1 are %d and %d", P1.x, P1.y);
        P2. x = 4;
        printf("\n The x coordinate of P2 is %d", P2.x);
        P2.y = 5;
        printf("\n The y coordinate of P2 is %d", P2.y);
        return 0;
}
```

Output

```
The coordinates of P1 are 2 and 3
The x coordinate of P2 is 4
The y coordinate of P2 is 5
```

UNIONS INSIDE STRUCTURES

Generally, unions can be very useful when declared inside a structure. Consider an example in which we want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario:

```
#include <stdio.h>
struct student
{
        union
        {
        char name[20];
        int roll_no;
        };
        int marks;
};
int main()
{
        struct student stud;
        char choice;
        printf("\n You can enter the name or roll number of the student");
        printf("\n Do you want to enter the name? (Y or N): ");
        choice=getchar();
        if(choice=='y' || choice=='Y')
        {
        printf("\n Enter the name:");
        scanf("%s", stud.name);
        }
        else
        {
        printf("\n Enter the roll number: ");
        scanf("%d", &stud.roll_no);
        }
        printf("\n Enter the marks: ");
        scanf("%d", &stud.marks);
        if(choice=='y' || choice=='Y')
        printf("\n Name: %s ", stud.name);
        else
        printf("\n Roll Number: %d ", stud.roll_no);
        printf("\n Marks: %d", stud.marks);
        return 0;
}
```

Now in this code, we have a union embedded within a structure. We know the fields of a union will share memory, so in the main program we ask the user which data he/she would like to store and depending on his/her choice the appropriate field is used.

1.8 POLYNOMIAL

It is a collection of terms, and each term consist of coefficient, variable and exponent.

Representation of polynomial can be done in 2 ways:

1) Array Representation
2) Structure Representation

Array Representation

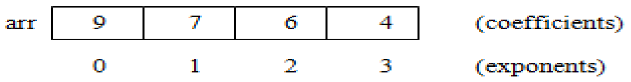
Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array.

Here index represents exponents and coefficients are stored in that index and fill 0 for missing terms.

Ex: $P(x) = 6x^3+3x^2+4$

0	1	2	3
4	0	3	6

$P(x) = 4x^3+6x^2+7x+9$



arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

Drawback: Memory waste for missing terms and this can be avoided by using structures

Structure Representation

A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. T he structure definition may be given as shown below:

```
struct polynomial
{
int coefficient;
int exponent;
};
```

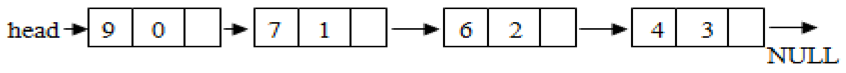
$6x^3+3x^2+4$

6	3	4
3	2	0

Structure Representation using linked list

$P(x) = 4x^3+6x^2+7x+9$

Thus the above polynomial may be represented using linked list as shown below:



```
struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

Structure representation program

```
#include<stdio.h>
#include<math.h>
struct polynomial
{
   int coeff;
   int expo;
};
typedef struct polynomial poly;
void readpolynomial(poly p[], int n);
int addpolynomial(poly p1[], poly p2[], int n1, int n2, poly p3[]);
void printpolynomial(poly p[], int n);
void evaluate(poly p[],int n);
void readpolynomial(poly p[], int n)
{
    int i;
   for (i = 0; i < n; i++)
{
        printf("\n Enter the Coefficient and the Exponent");
        scanf("%d%d",&p[i].coeff,&p[i].expo);
    }
}
int addpolynomial(poly p1[], poly p2[], int n1, int n2, poly p3[])
{
    int i=0, j=0, k=0;
    while (i < n1 && j <n2)
   {
         if (p1[i].expo == p2[j].expo)
         {
                    p3[k].coeff = p1[i].coeff + p2[j].coeff;
                    p3[k].expo = p1[i].expo;
                    i++;
                    j++;
                    k++;
        }
       else if (p1[i].expo > p2[j].expo)
       {
                    p3[k]=p1[i];
                    i++;
                    k++;
        }
        else
         {
                    p3[k] = p2[j];
                    j++;
                    k++;
        }
     }
     while (i < n1)
     {
        p3[k] = p1[i];
        i++;
        k++;
      }
    while (j < n2)
   {
         p3[k] = p2[j];
         j++;
         k++;
     }
     return (k);
  }
  void printpolynomial(poly p[], int n)
  {
     int i;
     for (i = 0; i < n - 1; i++)
        printf("%d(x^%d)+", p[i].coeff, p[i].expo);
     printf("%d(x^%d)", p[n - 1].coeff, p[n - 1].expo);
  }
```

```
void evaluate(poly p[],int n)
{
        int sum=0,i,x;
        printf("\n enter the value of x:");
        scanf("%d",&x);
        for (i = 0; i < n; i++)
                sum=sum+p[i].coeff*pow(x,p[i].expo);
        printf("\n Result=%d\n",sum);
}
int main()
{
        int n1, n2, n3;
        poly p1[10],p2[10],p3[10];
        printf("enter number of terms in first polynomial");
        scanf("%d",&n1);
        readpolynomial(p1,n1);
        printf("enter number of terms in second polynomial");
        scanf("%d",&n2);
        readpolynomial(p2,n2);
        printf(" \n First polynomial : ");
        printpolynomial(p1, n1);
        printf(" \n Second polynomial : ");
        printpolynomial(p2, n2);
        n3 = addpolynomial(p1,p2,n1,n2,p3);
        printf(" \n Resultant polynomial after addition : ");
        printpolynomial(p3, n3);
        evaluate(p3, n3);
        return 0;
```

}

1.9 SPARSE MATRICES

Sparse matrix

A **sparse matrix** is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense.** Operations using standard dense- matrix structures and algorithms are slow and inefficient when applied to large sparse matrices as processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus require significantly less storage.

Storing a sparse matrix

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation (triplet representation)
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)

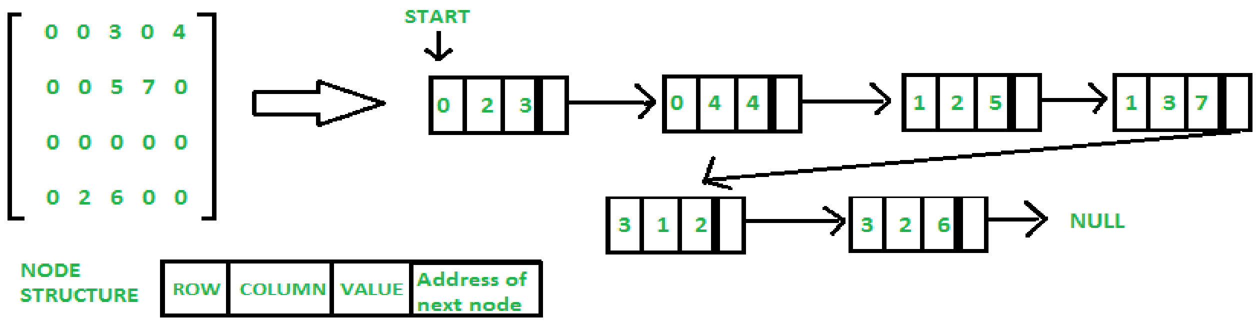
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Method 2: Using Linked Lists:

In linked list, each node has four fields. These four fields are defined as:

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non-zero element located at index – (row,column)
- Next node: Address of the next node



```
#include <stdio.h>
struct sparse
{
   int row;
   int col;
   int val;
} s[10];
void readsparsematrix()
{
   int i,j,r,c,ele,pos=0;
   printf("Enter rows and
cols:\n");
   scanf("%d%d", &r,&c);
   for (i = 0; i < r; i++)
```

```
        {
          for (j = 0; j < c; j++)
              {
                    scanf("%d", &ele);
                    if (ele != 0)
                    {
                            pos++;
                            s[pos].row = i;
                            s[pos].col = j;
                            s[pos].val = ele;
                    }
              }
         }
      s[0].row = r;
      s[0].col = c;
      s[0].val = pos;
  }
  void triplet()
  {
        int i;
        printf("\nTriplet representation:\n");
        printf("Row  Col  Value\n");
        for (i = 0; i <= s[0].val; i++)
        {
          printf("%d    %d    %d\n", s[i].row, s[i].col, s[i].val);
        }
  }
  void search()
  {
        int found=0,key,i;
        printf("Enter key:\n");
        scanf("%d", &key);
        for (i = 1; i <= s[0].val; i++)
        {
          if (s[i].val == key)
           {
              printf("%d    %d    \n", s[i].row, s[i].col);
              found=1;
              break;
            }
      }
      if(found==0)
        printf("\n element not  found:");
  }
  void transpose()
  {
```

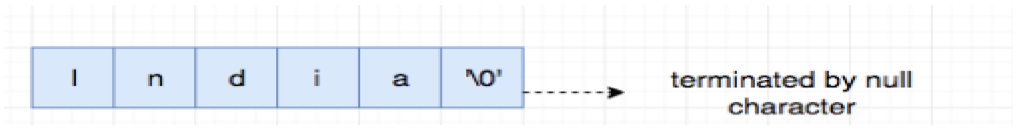
```
int i,j;
struct sparse trans[10];
trans[0].row = s[0].col;
trans[0].col = s[0].row;
trans[0].val = s[0].val;
int k = 1;
for (i = 0; i < s[0].col; i++)
{
    for (j = 1; j <= s[0].val; j++)
     {
        if (s[j].col == i)
         {
            trans[k].row = s[j].col;
            trans[k].col = s[j].row;
            trans[k].val = s[j].val;
            k++;
         }
     }
 }
for (i = 0; i <= s[0].val; i++)
{
    s[i] = trans[i];
}
```

```
    triplet();
  }
  int main()
  {
      readsparsematrix();
      triplet();
      search();
      transpose();
      return 0;
  }
```

1.10 STRING PATTERN MATCHING

A String in C programming is a sequence of characters terminated with a null character ‘\0’. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character ‘\0’.

Ex – char s[]=”India”;



Pattern Matching

Assume two strings, **string** and **pat**, where **pat** is a pattern to be searched for in **string**.

The built- in function **strstr** can be used to perform this operation.

char pat[MAX_SIZE], char string[MAX_SIZE], char *t;

The built in function **strstr** is illustrated as follows,

```
if( t = strstr(string, pat))
```

```
                    printf("The string from strstr is: %s\t",t);
          else
                    printf("The pattern was not found with strstr\n");
```

The call **(t = strstr(string,pat))** returns a null pointer if **pat** is not in **string**. If **pat** is in **string**, **t** holds a pointer to the start of **pat** in **string**. The entire string beginning at position **t** is printed out. Techniques to improve pattern matching,

- By quitting when strlen(pat) is greater than the number of remaining characters in the string.
- Checking the first and last characters of pat and string before we check the remaining characters.

1. Naive algorithm for Pattern Searching (nfind)

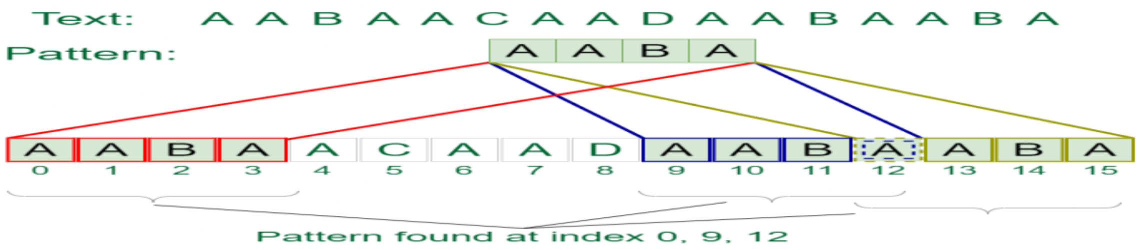
Given **text** string with length **n** and a **pattern** with length **m,** the task is to prints all occurrences of **pattern** in **text**.

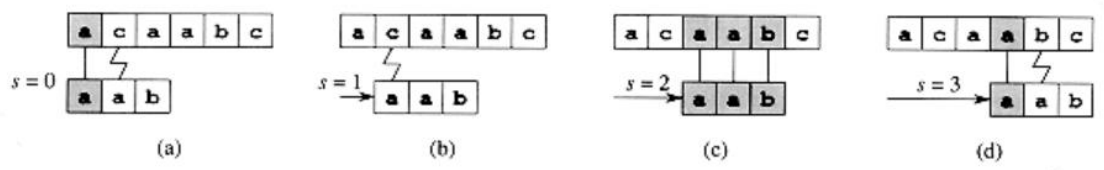
Note: We assume that n > m.

Examples:

Input: text = "THIS IS A TEST TEXT", pattern = "TEST"
Output: Pattern found at index 10

Input: text = "AABAACAADAABAABA", pattern = "AABA"
Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12


Text: A A B A A C A A D A A B A A B A
Pattern: A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12



Program : nfind

```
#include <stdio.h>

#include <string.h>

void nfind(char* pat, char* txt)

{

    int i,j;

    int m = strlen(pat);

    int n = strlen(txt);

    for (i = 0; i <= n - m; i++)

    {

        for (j = 0; j < m; j++)

        {

            if (txt[i + j] != pat[j])

             {

               break;
```

```
                        }
                }
                if (j == m)
                {
                        printf("Pattern found at index %d\n", i);
                }
        }
}
int main()
{
        char txt1[] = "AABAACAADAABAABA";
        char pat1[] = "AABA";
        nfind(pat1, txt1);
        return 0;
}
```

2. Kruth, Morris, Pratt (KMP) Pattern Matching Algorithm

Given two strings txt and pat of size N and M, where N > M. String txt and pat represent the text and pattern respectively. The task is to print all indexes of occurrences of pattern string in the text string.

The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

Matching Overview

txt = "AAAAABAAABA"

pat = "AAAA"

We compare first window of txt with pat

txt = "AAAAABAAABA"

pat = "AAAA" [Initial position]

We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.

txt = "AAAAABAAABA"

pat = "AAAA" [Pattern shifted one position]

In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing? –failure array

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array failure[] that tells us the count of characters to be skipped.

j	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	a	b	c	a	c	a	b
failure	-1	-1	-1	0	1	2	3	-1	0	1

j	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	d	a	b	e	a	b	f
failure	-1	-1	-1	-1	0	1	-1	0	1	-1

j	0	1	2	3	4	5	6	7	8	9	10	11	12
Pat	a	a	b	c	a	a	a	a	a	a	a	b	c
failure	-1	0	-1	-1	0	1	1	1	1	1	1	2	3

j	0	1	2	3
Pat	a	a	a	b
failure	-1	0	1	-1

```
void fail(char *pat)
{
    int n = strlen(pat);
    failure[0] = -1;
    int i,j;
    for(j=1; j<n; j++)
    {
            i = failure[j-1];
            while((pat[j] !=pat[i+1]) && (i>=0))
                    i=failure[i];
            if(pat[j]==pat[i+1])
```

```
                    failure[j]=i+1;
                else
                    failure[j] = -1;
            }
        }
```

Consider txt[] = “**AAAAABAAABA**“, pat[] = “**AAAB**“

lens=11, lenp=4

failure**[] = {-1,0,1,-1}**

j	0	1	2	3
Pat	a	a	a	b
failure	-1	0	1	-1

-> i = 0, j = 0: string[i] and pat[j] match, do i++, j++

-> i = 1, j = 1: string[i] and pat[j] match, do i++, j++

-> i = 2, j = 2: string[i] and pat[j] match, do i++, j++

-> i = 3, j = 3:

string[i] and pat[j] did not match, and j!=0 reset **j = failure[j-1] + 1**= failure[3-1]+1 = failure[2]+1=1+1=2

-> i = 4, j = 2: string[i] and pat[j] match, do i++, j++

-> i = 4, j = 3: string[i] and pat[j] did not match, and j!=0 reset **j = failure[j-1]** + 1 = failure[3-1]+1=failure[2]+1=1+1=2

-> i = 4, j = 2: string[i] and pat[j] match, do i++, j++

-> i = 5, j = 3: string[i] and pat[j] match, do i++, j++

-> i = 5, j = 3: string[i] and pat[j] match, do i++, j++

-> i = 6, j = 4, return i-lenp= 6-4 =2

```
int pmatch(char *string, char *pat)
{
    int i=0, j=0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while(i<lens && j<lenp)
    {
            if(string[i] == pat[j])
            {
             i++; j++;
            }
            else if(j==0)
            i++;
            else
                    j=failure[j-1]+1;
    }
            return ((j == lenp) ? (i-lenp) : -1);
}
```