MODULE - V

Pointers and PreProcessors.

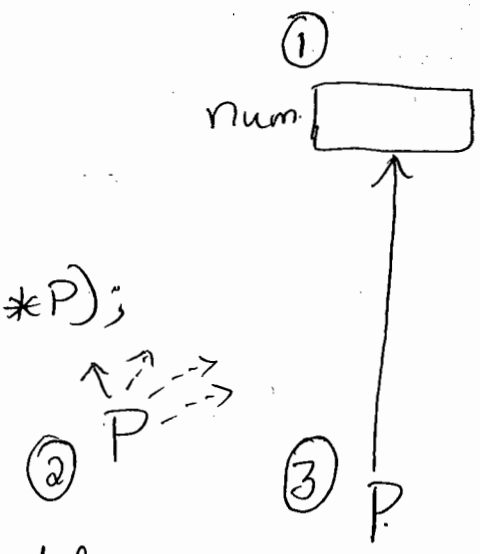
Definition:

A Pointer is a Variable which can hold the address of another Variable. It also Provides an alternative method to access the contents of a memory location.

The Steps to be followed How to USe Pointers:-

1. Declare a data Variable.
2. Declare a Pointer Variable.
3. Initialize a Pointer Variable.
4. Access data Using Pointer Variable.

Example:
```
1. int num;
2. int *P;
3. P = &a;
4. Printf("%d", *P);
```



Where, P is a Pointer Variable Created.

Pointer Declaration and definition:

Declaration of a Pointer is a Process of creating a Pointer declaration.

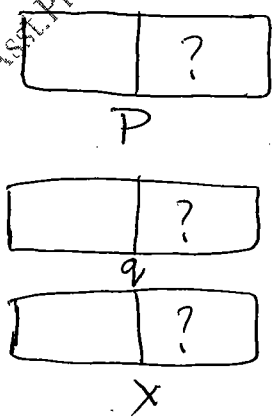
Syntax:

```
datatype  *Variable ;
```

datatype → integer, Char, float.

*Variable → Pointer (indirection Operator).

```
int *P ;
float *q ;
double *x ;
```



Pointer to User defined data type:

```
typedef int Marks ;
```

↑ User defined datatype.

```
Marks a, b, c ;
```

```
Marks *P;
```

Here, the Variable P is a Pointer Variable Created using User defined datatype Marks.

Pointer to derived datatypes:

```
Struct College
{
    Char name[25];
    int id;
};
Struct College *C;
```

Here, *C is a Structure Variable Created using Pointers to the derived datatypes.

Dangling Pointers:

```
int *P;
```

A Pointer Variable does not Contain a Valid address is Called dangling Pointers.

Here, the P is a Pointer Variable and the Corresponding memory location Should Contain address of an integer Variable, but declaration will not initialize the memory location & the memory Contains garbage Value.

NULL Pointer:

A NULL Pointer is defined as a Special Pointer Value that Points to nowhere in the memory.

Ex:- int *P = NULL;

Initializing a Pointer Variables:-

Initialization of a Pointer Variable is a Process of assigning an address to the Pointer Variable.

The Steps followed to initialize are

→ Declare a data Variable.

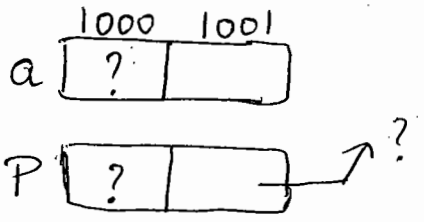
→ Declare Pointer Variable.

→ Assign address of a data Variable to Pointer Variable.

Example:

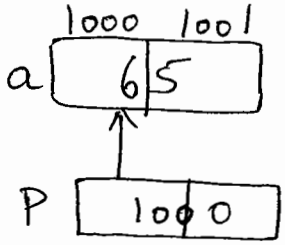
① int a;

int *P;



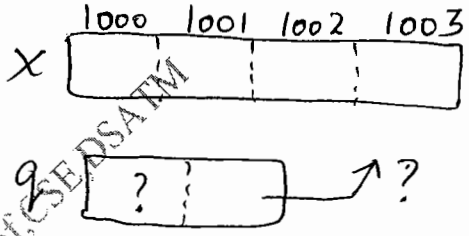
a = 65;

P = &a;



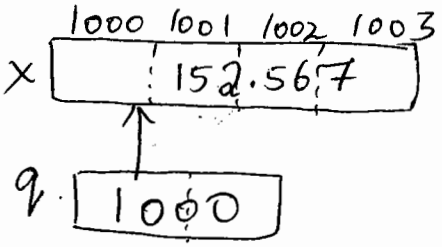
② float x;

float *q;



x = 152.567;

q = &x;



Accessing Variables Through Pointers:-

The Contents of a data Variable Can be accessed by de-referencing a Pointer. The * operator acts as a dereference operator.

Ex:- num 365

P

where, P is a Pointer Variable.

The Value 365 Can be accessed in 2 ways.

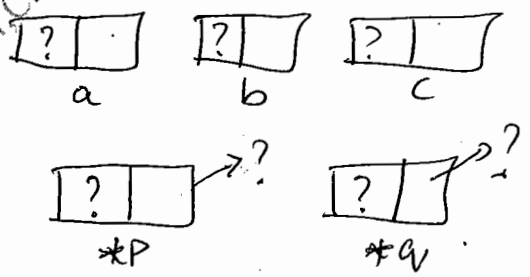
(i) Printf("%d", num); o/p = 365

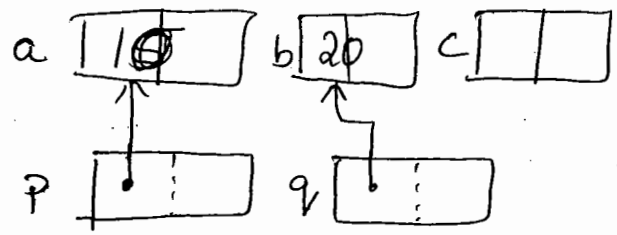
(ii) Printf("%d", *P); o/p = 365

↑ accessing Variable by dereferencing the Pointer P.

Example:-

```
#include <stdio.h>
Void main()
{
    int a, b, c;
    int *P, *q;
    a = 10;
    b = 20;
    P = &a;
    q = &b;
    c = *P + *q;
    Printf("%d", c);
}
```



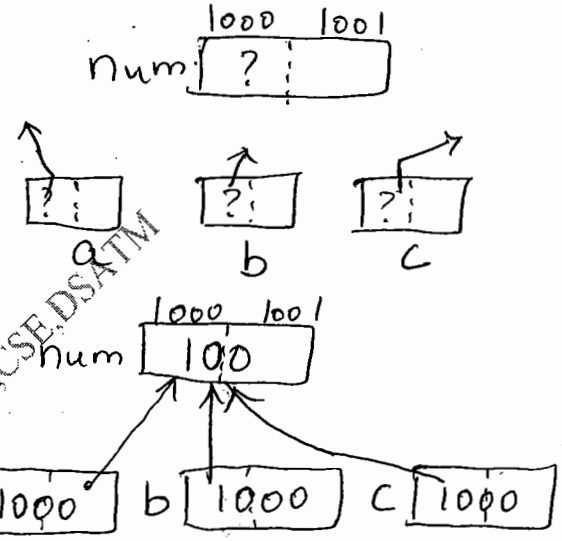


Output = 30

The many number of Pointers Pointing to a Same Variable is also Possible.

Example:

```
#include <Stdio.h>
Void main()
{
  int num;
  int *a, *b, *C;
  num = 100;
  a = &num;
  b = &num;
  C = &num
  Printf(" Value of num = %d ", num);
  Printf(" Value of num = %d", *a);
  Printf (" Value of num = %d", *b);
  Printf (" Value of num = %d", *c);
}
```



Output:-

Value of num = 100
Value of num = 100
Value of num = 100
Value of num = 100.

The Only one Pointer Point to more than one Variables.

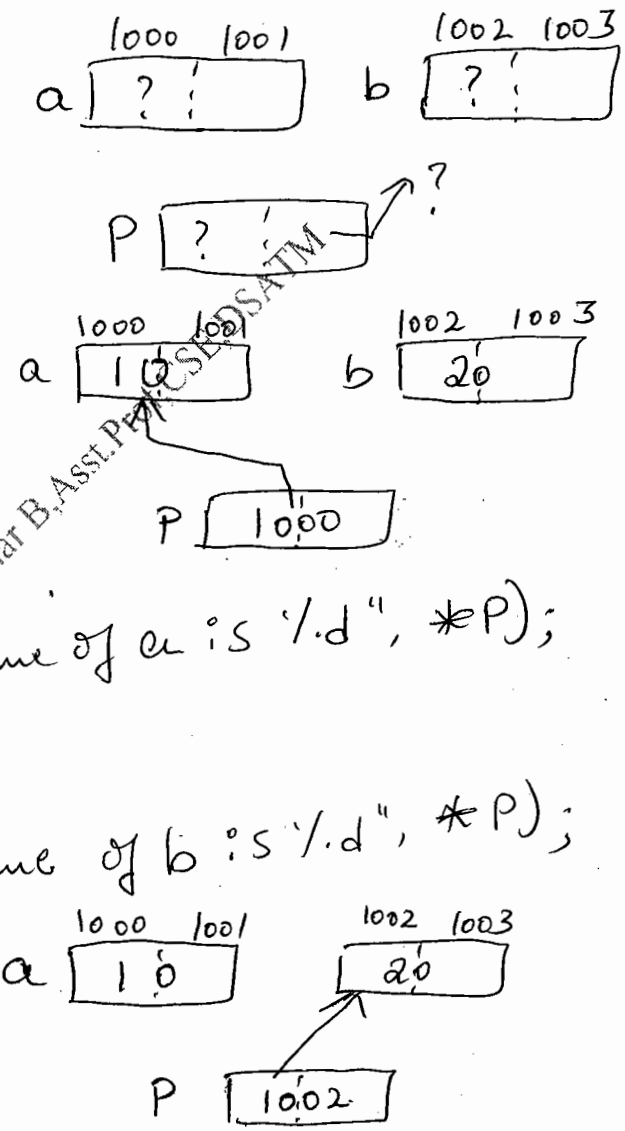
Example:

```
#include<stdio.h>
Void main()
{
    int a,b;
    int *P;

    a = 10;
    b = 20;

    P = &a;
    Printf("Value of a is %d", *P);

    P = &b;
    Printf("Value of b is %d", *P);
}
```



Output:

Value of a is 10
Value of b is 20.

The Difference between Pointer Variable and Normal Variable:

Pointer Variable	Normal Variable.
(i) A Pointer Variable holds the address	(i) A Normal Variable holds the data
(ii) The General form is int *P; ↑ Pointer operator	(ii) The General form is int a;
(iii) Dereference a Pointer Variable to acces the data	(iii) No-need to de-reference a normal Variable to access data.

Pointers And Functions:

As the normal Variables Passed to the functions the address of the Variable is Passed through the Pointers.

The Values Can be Passed to the function Called Pass by Value, and the address Passed to the function is also Called Pass by reference.

Example:-

```
#include <stdio.h>
Void Swap (int *P, int *q)
{
    int temp;
    temp = *P;
    *P = *q;
    *q = temp;
}
Void main()
{
    int x=20, y=30;
    Printf("Values before swap");
    Printf("%d %d", x, y);
    Swap(&x, &y);
    Printf("Values after Swap");
    Printf("%d %d", x, y);
}
```


Output

```
Values before Swap
20 30
Values after Swap
30 20
```

Passing Structures By Reference:

As normal Variables the Structure Variables also Passed as a reference to the functions.

```
#include <stdio.h>
Void display(Struct Emp *e);
Struct Emp
{
   Char name[25];
    int no;
};
Void main()
{
   Struct Emp e1;
   Printf("Enter name, num\n");
    Scanf("%s %d", e1.name, &e1.no);
     display (&e1);
}
Void display (Struct Emp *e)
{
    Printf("Name=%.s", e->name);
    Printf("No = %.d", e->no);
}
```

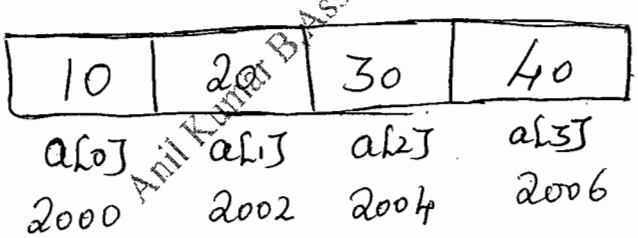
OUTPUT:-

```
Enter name & no.
ABC   10
Name = ABC
No.  = 10.
```

Pointers and Arrays:

The Pointers Can also be Created to arrays. Consider an array Created as

```
int a[4] = {10, 20, 30, 40};
```

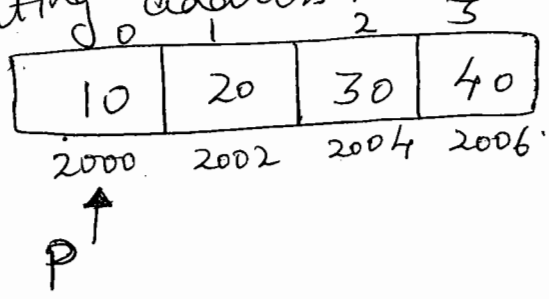



```
int *P;
P = &a[0];
```

or

```
P = a;
```

The Pointer P would be Pointing to the Starting address.

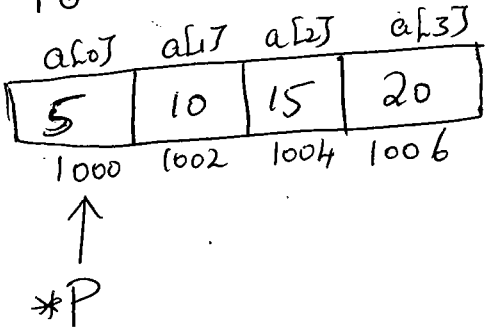


Example:-

```
#include<stdio.h>
Void main()
{
   int a[4]={5,10,15,20};
   int *P;
   int i;
   P=a;
   for(i=0; i<4; i++)
   {
      Printf("%d\t", *P);
      P++;
   }
   getch()
}
```


OUTPUT:

5 10 15 20



When P++ is executed in the for loop the Pointer to the next address 1002 in the array.

Pointer Arithmetic:

The Various arithmetic operations Performed on Pointers are incrementation, decrementation, addition, Subtraction and Comparision.

Incrementation on Pointers:-

The increment operator (++) increases the value of a Pointer by the size of data the Pointer Pointing to.

-> If Pointer Pointing to Character type then increase Value by 1

-> If Pointer Pointing to Integer type then increase Value by 2.

-> If Pointer Pointing to float then increase Value by 4.

-> If Pointer Pointing to double type the increase Value of the Pointer by 8.

Example:- Pointer to Char

```
#include <stdio.h>
Void main()
{
    Char a;
    Char *Ptr = &a;
    Printf(" Before increment %u :", Ptr);
    Ptr++;
    Printf(" After increment %u", Ptr);
}
```

OutPut:

Before increment: 3405703

After increment : 3405704.

Pointer to int (integer):-

```
#include <stdio.h>
Void main()
{
    int  x;
    int  *P = &x;
    Printf(" Before increment %u", P);
    P++;
    Printf(" After increment %u", P);
}
```

OutPut:

Before increment : 5045351

After increment : 5045353

Pointer to float:

```
#include<stdio.h>
Void main()
{
    float P;
    float *Ptr = &P;
   Printf(" Before increment %u", Ptr);
    Ptr++;
   Printf(" After increment %u", Ptr);
}
```

Output:

Before increment: 5307940
After increment: 5307944

Decrementation on Pointers:

The decrement operator (--) decreases the value of a pointer by the size of the data pointer pointing to.

→ If Pointer Pointing to the data types

float - decrement by 4
int - decrement by 2
Char - decrement by 1
double - decrement by 8.

Example: Pointer to Char

```
#include <Stdio.h>
Void main ()
{
    Char a;
    Char *P=&a;
    Printf(" Before decrement %u", P);
     P--;
     Printf(" After decrement %u", P);
}
```

O/P

Before decrement 3472147

After decrement 3472146

Pointer to float:

```
#include <Stdio.h>
Void main ()
{
    float x;
    float *P = &x;
    Printf(" Before decrement %u", P);
     P--;
     Printf(" After decrement %u", P);
}
```

Before decrement 3210640

After decrement 3210636

Pointer Addition:

The final Value of the Pointer Variable Can be Computed using.

Final Value = Current Value of Pointer + (Integer number * Size of datatype);

Example:

Pointer to integer:

```
#include <Stdio.h>
Void main()
{
    int x;
    int *P = &x;
   Printf(" Before addition %u", P);
    P= P+10;
   Printf(" After addition %u", P);
}
```


Final Value of Pointer = 2490131 + (10 * Size of int)

= 2490131 + (10 * 2)

= 2490131 + 20

= 2490151.

Pointer to double:

```
#include <stdio.h>
Void main ()
{
    double a;
    double *P = &a;
    Printf(" Value Pointer befre add '/.u", P);
    P = P + 5
    Printf(" After addition '/. u", P);
}
```


OutPut:

Value of Pointer Before add: 3866260

Value of Pointer After add = 3866300

Final Value of Pointer = Current Value + (Integer num * Size of type);

= 3866260 + (5 * 8)

= 3866260 + 40

= 3866300.

Pointer Subtraction:

The final Value of Pointer Variable. Calculated as

Final Value of Pointer = Current Value of Pointer − (Integer num * Size of type);

Example: Pointer to int

```
#include <stdio.h>
 Void main()
{
    int a;
    int *P = &a;
   Printf(" Before Subtraction %u", P);
    P = P - 10;
   Printf(" After Subtraction %u", P);
}
```


Output:

Before Subtraction: 3406251

After Subtraction: 3406241

Final Value of Pointer = 3406251 − (5 * 2)

= 3406251 − 10

= 3406241

Comparision of Pointers:

The two Pointers can be Compared if both the Pointers are pointing to the similar datatype.

Example: int a[4] = {10, 20, 30, 40};

```
int *ptr1, *ptr2;
int *fptr;
ptr1 = a;
ptr2 = &a[3];
```

→ ptr2 != ptr1;
→ ptr1 == ptr2;
→ ptr1 <= ptr2;
→ ptr1 >= ptr2;

} Valid Statements

→ fptr != ptr1; //Invalid Statement fptr of datatype float.

Program:

```
#include <stdio.h>
{
    int a[] = { 5, 10, 15, 20};
    int *p;
    int *q;
```

```
P = &a[0];    /* Point to first element */
q = &a[3];    /* Points to Last element */.
while (P<=q)  /* Comparing two Pointers*/
{  Printf(" %d ",*P);
   P++;
  }
}
```

Output:

5 10 15 20.

Character Pointer and Function:-

As the various Parameters Passed to the function we can Pass arrays or Character Pointers to the functions.

Example:-

→ Arrays

```
Void fname(int a[])
{
    Accessed by using
    a[i]
}
```

→ Pointers

```
Void fname(int *a)
{
    accessed
    by using
    *(a+i)
}
```

Pointer to Pointer:-

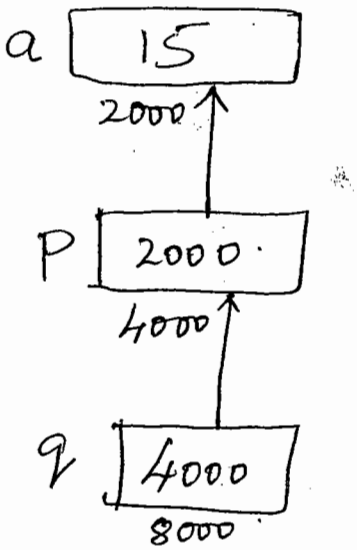
The double Pointers are used to Store the address of Pointer Variables.

Ex:- int a=15;
int *P;
int **q;

P=&a;
q=&P;

Program:-

```
#include<stdio.h>
Void main()
{
   int a=15;
   int *P;
   int **q;
    P= &a;
    q= &P;
   Printf("%d", a);
   Printf("%d", *P);
   Printf("%d", **q);
}
```



Output:
15
15
15

Program :- C-Program to find the Length of String using Pointers & user defined function mystrlen();

```
#include<stdio.h>
int mystrlen(char *str)
{
    int i=0;
    while (*(str+i))
      i++;
    return i;
}
Void main()
{
   Char str[20];
   int res;
   Printf(" enter String ");
   gets(str);
   res = mystrlen(str);
   Printf(" Length = %d", res);
}
```


Output:

Enter String : WELCOME

Length = 7.

Dynamic Memory allocation Methods:

Dynamic memory allocation is the Process of allocating memory during execution time. For an unPredictable Storage requirement dynamic allocation technique is used.

The different memory allocation functions in C are:

- → malloc()
- → Calloc()
- → realloc()
- → free().

malloc():

The malloc() function can be used to allocate memory dynamically. malloc stands for "memory allocation".

Syntax:

```
Ptr = (datatype *) malloc (size);
```

- → Ptr is a Pointer Variable.
- → size is number of bytes required.

Example:

```
#include <stdio.h>
#include <stdlib.h>
Void main()
{
    int i, n;
    int *P;
   Printf("Enter number of elements");
   Scanf("%d", &n);
    P= (int *)malloc (size of(int) *n);
    if (P==NULL)
    {
      Printf("In sufficient");
       return;
    }
   for(i=0; i<n; i++)
     Scanf("%d", P+i);
   Printf("elements are");
   for(i=0; i<n; i++)
     Printf("%d", *(P+i));
}
```


Output:

```
Enter number of elements 5
10 20 30 40 50
Elements are.
10 20 30 40 50.
```

Calloc (n, size);-

The Calloc function used to allocate multiple blocks of memory. Calloc Stands for Contiguous allocation of multiple blocks.

Syntax:

Ptr = (datatype *) Calloc (n, size);

n -> number of blocks to be allocated

Size -> Number of bytes in each block.

Example:-

```
#include <stdio.h>
#include <stdlib.h>
Void main()
{
    int *P, i, n;
    Printf(" enter no. of elements ");
    Scanf("%d", &n);
    P = (int *) Calloc (n, sizeof(int));
    if (P == NULL)
    {
        Printf(" Insufficient memory ");
        return;
    }
```

```
Printf("enter elements");
  for(i=0;i<n;i++)
    Scanf("%d", &p[i]);
 Printf("elements are");
  for(i=0; i<n; i++)
    Printf("%d", p[i]);
    free(P);
}
```

Output:

Enter number of elements: 4
Enter elements: 5 10 15 20

Elements are
5 10 15 20

realloc (Ptr, Size):

The allocated memory is not Sufficient Sometimes then the realloc function used to extend the additional memory space required.

Syntax:

Ptr = (datatype *) realloc (Ptr, Size)

-> Ptr is a Pointer to a block of Previously allocated using malloc() or Calloc().

-> Size is new size of the block.

Example :

```
#include <stdio.h>
#include <Stdlib.h>
#include <String.h>
Void main()
{
   Char *S;
   S = (Char *) malloc (10);
   Strcpy (S, "Computer");
   S = (Char *) realloc (S, 40);
   Strcpy (S, "Computer science Engg");
}
```


free (Ptr) :

The function used to deallocate the allocated block of memory which is allocated by using the functions Calloc(), malloc() and realloc(). The Programmer Should deallocate the memory whenever it is not required by Program and initialize the Pointer to Null.

Example:

```
#include <stdio.h>
#include <stdlib.h>
Void main()
{
    int *Ptr;
    Ptr = (int *) malloc (sizeof(int));
    *Ptr = 100;
    free (Ptr);
    Ptr = NULL;
}
```


Advantages & disadvantages of Pointers:

Advantages:

→ More than one Value can be returned using Pointers.

→ Compact Code can be written using Pointers

→ Data accessing is faster using Pointers compared to arrays.

Disadvantages:

* Uninitialized Pointers can cause System Crash
* The bugs are very difficult to identify and correct. if Pointers used incorrectly.

Introduction to PreProcessors.

PreProcessor:

The PreProcessor is used automatically by the C-Compiler to transform all C-Programs before Compilation. The PreProcessing Statements are also called PreProcessor directives Starts with Symbol #.

PreProcessor directives:

The PreProcessor directives are lines included in our Programs Start with Character #. These lines are not Program Statements but they are the instructions for the PreProcessor.

Example:

```
#define MAX 100
#include <stdio.h>
```

The advantages of Pre Processor:

- → Program becomes Simple
- → Easy to modify.
- → Program be Comes easy to read.

The different Commonly Used PreProcessor Statements / directives are:

(i) #include.

The include directive used to include the header files.

Example:-

```
#include <Stdio.h>
#include <Conio.h>
Void main()
{
   Clrscr();
   Printf("WELCOME");
}
```

OutPut;

WELCOME.

(ii) #define

The #define is used to define macros.

Example:

```
#include <Stdio.h>
#define PI 3.1415
Void main()
{
   int r;
   float area;
```

```
    Printf(" Enter radius ");
    Scanf(" %d", &r);
    area = PI *r*r;
    Printf(" Area of Circle = %f", area);
}
```

Output:

Enter radius: 10

Area of Circle = 314.1499.

(iii) #if, #else, #endif.

These are also called Conditional Compilation Preprocessor directive.

Example:-

```
#include <stdio.h>
Void main()
{
  #if ((10%2)==0)
   Printf(" Number is even ");
  #else
   Printf(" Number is odd ");
  #end if.
}
```

Output:- Number is even.

Macros:

A macro is a name given to group of Statements. Each time macro is called in our Program, the Preprocessor replaces this name with the defined group of Statement.

A macro is defined using #define directive

Example:

```
#include <stdio.h>
#define SQUARE(x) (x*x)
Void main()
{
    int m,n;
     m=10;  n=5;
    Printf("Square 5 = %d", SQUARE(n));
    Printf("Square 10 = %d", SQUARE(m));
    Printf("Square 10+5 = %d", SQUARE(m+n));
}
```


Output:

Square 5 = 25

Square 10 = 100

Square 10+5 = 65 (wrong output for the correct Square)

(10+5 * 10+5)

(10+50+5) = 6 [Arithmetic Operator Precedence.]

Data Structures

Definition:

A DataStructure is a way of organizing data in a Computer so that it can be used efficiently.

The two types of data types are:

* Primitive data type
* Non-Primitive data type

Primitive datatype:

-> Primitive datatypes are also called basic data types.

-> Primitive data can Contain only Single Values, they does not Contains any Special Capabilities.

-> The examples are int, Char, float, etc.

Non-Primitive data type:-

-> A non Primitive datatype is normally a derived data type from Primitive datatypes

-> The non-Primitive data types are used to Store group of Values.

-> Examples - Stacks, Queues, Lists, Arrays.

A data Structures Store the data in Sequentially manner. Called as linear data Structures.

Ex:- Array, Stack, Queue & link list.

The non-linear data Structures allow for a more complex relationship among elements they Contain.

Ex:- Trees, Graphs, etc.

STACKS:

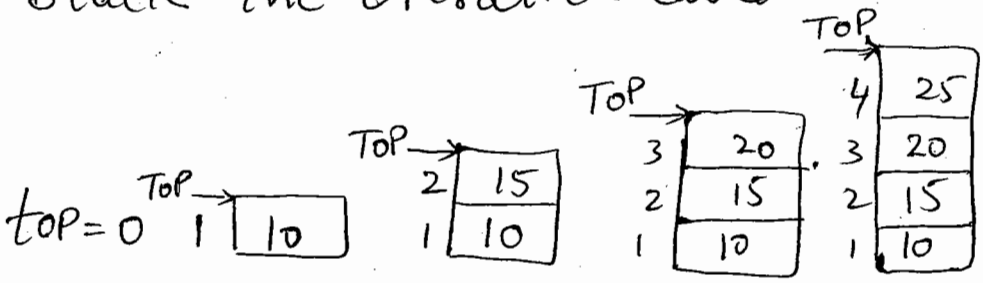
A Stack is a data Structure in which addition of a new element or deletion of an existing element always takes Palace at the Same end. Also Called at TOP of the Stack. [LIFO]- Last in first out.

The three Operations Performed on Stack are:

- PUSH
- POP
- DISPLAY

(i) PUSH:

When the element is added to the Stack the operation called as Stack.

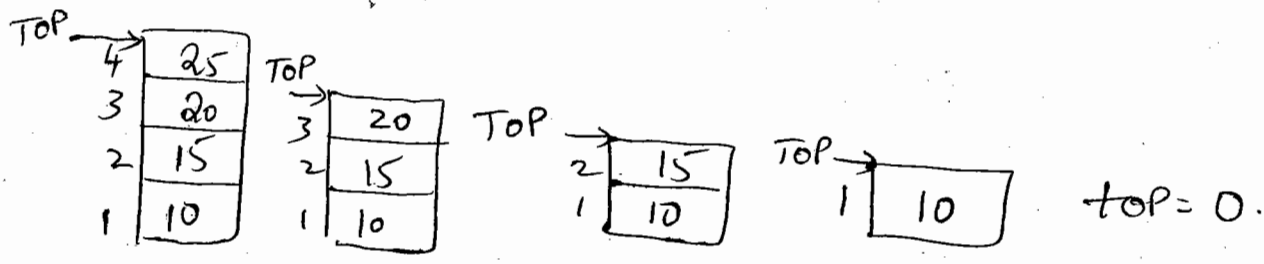


→ Increment the top as top = top+1

→ Insert or Push Number on to Stack until the condition [top = Num]

(ii) POP:

Deleting an existing element from the Stack called as POP operation. POP operation Starts from top of the Stack.



-> Decrement top to POP the elements.
i.e top = top-1

(iii) DISPLAY:

The Elements of the Stack displayed on the output Screen called as the Display Operation. Logically exist between First & last element of Stack.

Program: C Program to demonstrate PUSH, POP & display operations on Stack

```
#include<stdio.h>
#include<conio.h>
Static int Stack[10], top=-1;
Void main()
{
    Void Push(int);
    Void POP(void);
    Void display(void);
   Printf("PUSH Operation");
    Push(10);
    Push(20);
    Push(30);
    Push(40);
    display();
   Printf("POP operation");
    POP();
    POP();
    POP();
   POP();
    display();
}
```

```
Void Push(int x)
{
    top++;
    Stack[top] = x;
}
Void POP()
{
    Stack[top] = 0;
    top--;
}
Void display()
{
    int x;
    Printf("Elements of stack are");
    for(x=0 ; x<10;x++)
    Stack[x] != 0 ? Printf("%d", Stack[x]):
                    Printf(" ");
}
```


OUTPUT:

Push Operation.

10 20 30 40.

Pop Operation.

Elements of stack are

10 20 30

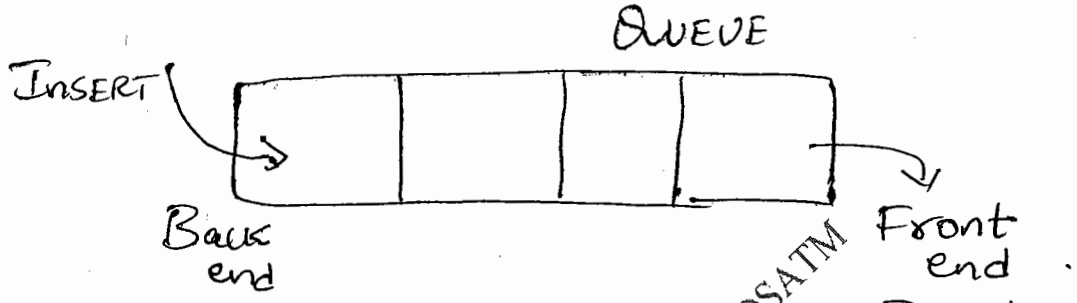
Elements of stack are

10 20.

⋮

QUEUES:

Queue is a Linear data structure where addition of elements takes Place at rear end [back end] and deletion of element at the front end.



Therefore the Queue can be called as [FIFO] - First In First Out data structure

There are four different types of Queues.

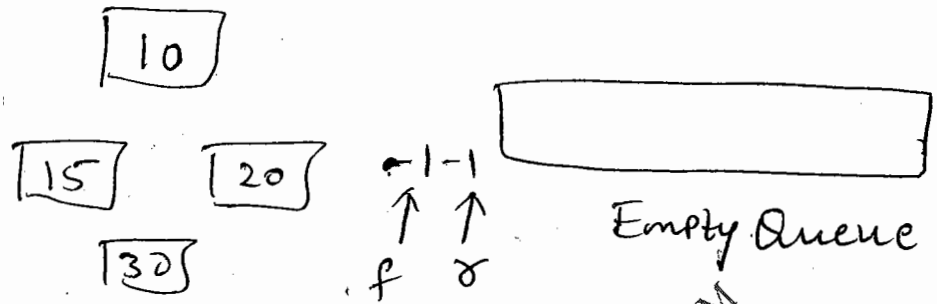
- → Ordinary Queue
- → Circular Queue
- → Double ended Queue
- → Priority Queue.

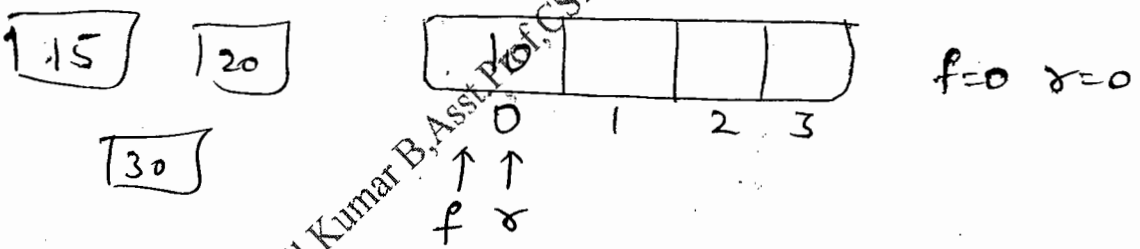
The Operations Performed on Queue are:

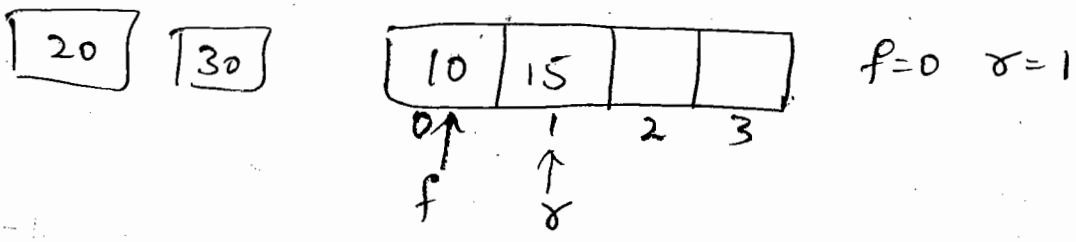
- → Insert Operation (Adding element)
- → Delete Operation (Deleting elements)
- → Display Operation (displaying the elements).

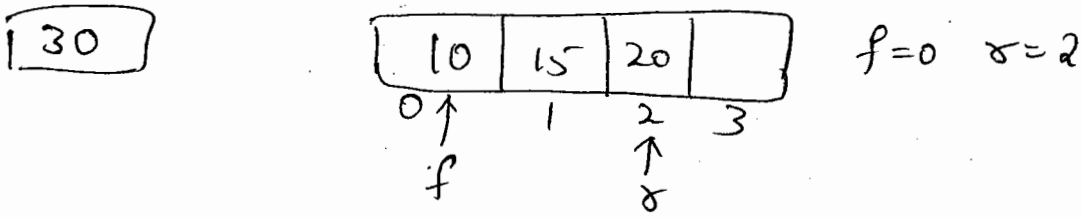
Insert Operation:

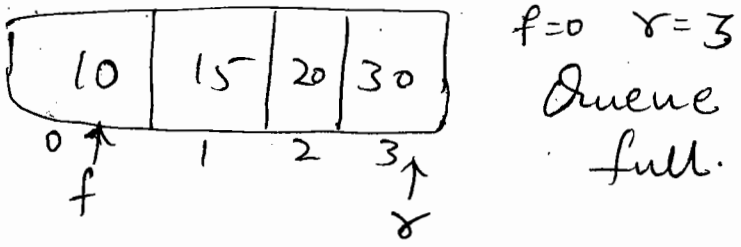
The Variables front-f and rear-r has to be monitored in a Queue. In an empty Queue both front & rear would be -1







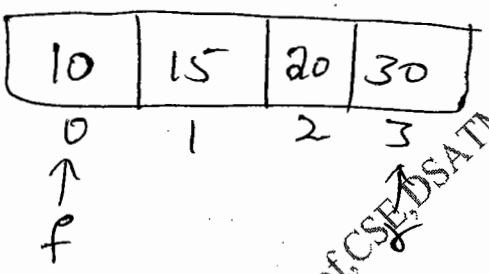




Delete operation:

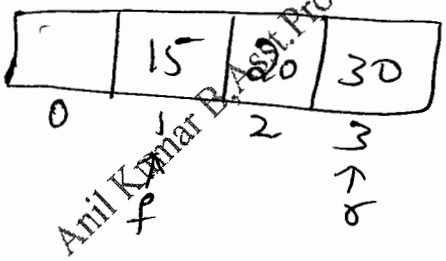
Deletion of elements would be done at the front end of the Queue. The front end (f) of Queue will be incremented after deletion of each element from the Queue.

(i)



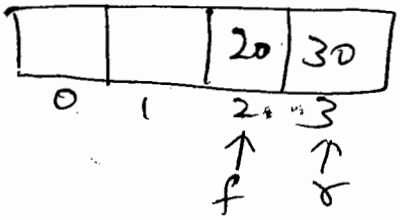
Initial Queue data
f = 0 r = 3

(ii)



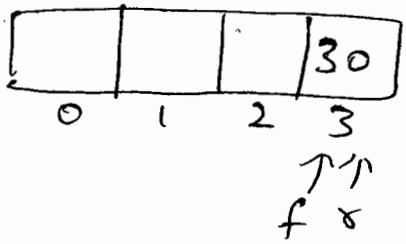
f = 1 r = 3

(iii)



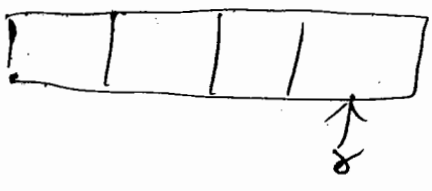
f = 2 r = 3

(iv)



f = 3 r = 3

(v)



f > r r = 3

C-function to insert element into Queue

```
Void insert()
{
   if (r == size-1)
    {
      Printf(" insertion not possible");
    }
   else
  { r++;
    q[r] = item;
    if(f==-1)
     { f++;
     }
  }
}
```


C function to delete element from Queue:

```
Void delete()
{  if (f>0)
    { Printf(" deletion Not Possible");
    }
   else
   { Printf(" Deleted element is %d", q[f++]);
   }
}
```

C - function to display elements of Queue:

```
Void display()
{
    int i;
    if (front == -1 || front > rear)
    {
        Printf("No elements in Queue");
    }
    else
    {
        for(i=front; i<rear; i++)
            Printf("%d", a[i]);
    }
}
```


Advantages of Ordinary Queue:

-> Used in Job Scheduling algorithms

-> Used in Printers to store requests made for Printing. When Printer is busy.

Disadvantages of ordinary Queue:

-> Element cannot be added or deleted from Queue on Priority basis.

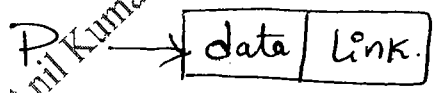
-> Insert at front end & deletion at rear end is not Possible.

LINKED LIST:

A Linked List is a linear Collection of data elements. These data elements are called nodes. Linked lists are the data structures that can be used to implement other data structures.

Example:

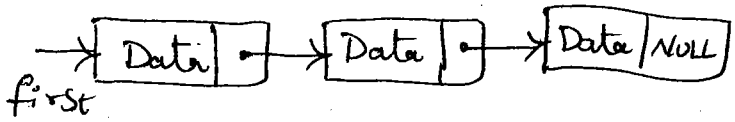
```
Struct node
{ int data;
  Struct node *link;
}
P = malloc(size of (Struct node));
```



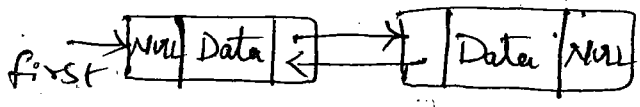
Here P→data and the link field can be accessed through P→link

Types of Linked Lists:

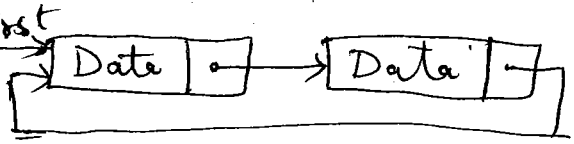
→ Single linked list



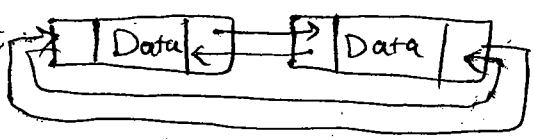
→ Doubly linked list



→ Circular Linked list



→ Circular doubly linked list



Operations Performed on Singly linked List:

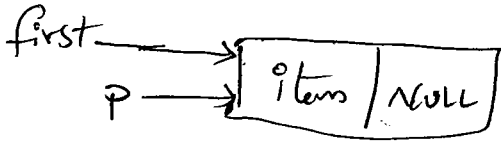
- Insert → at front end
  - at rear end
- Delete → at front
  - at rear.
- Display Operations.

Insert at front:

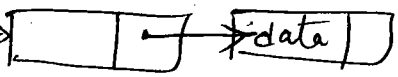
To insert a new node at front end of a linked list, a new node must be created using malloc() function

→ When the linked list does not exist the newly node created as first node.

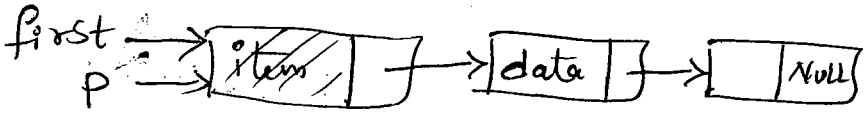
Create new node p → | item | NULL |



→ When linked list already exist. We must attach the node the new node becomes first.

Before 

After



C- function to insert element at front.

```
Void insertfront(int item)
{
    Struct node *P;
    P= malloc (sizeof (Struct node));
    P->data = item;
    P-> link = NULL;
    if (first == NULL)
    {
        first = P;
    }
    else
    {
        P-> link = first;
        first = P;
    }
}
```


Delete at front:

-> If first == NULL then linked list does not exist.

-> If first -> link is NULL then only one node in the list.

-> If first -> link is not NULL then more the one nodes are in the list to delete.

C function to delete element at front.

```
Void deletefront()
{
    if (first == NULL)
    {
        Printf("Deletion not Possible");
        return;
    }
    if (first -> link == NULL)
    {
        Printf("element deleted is %d",
                                first -> data);
        first = NULL;
        return;
    }
}
```


Display Operation On Single linked list:

Display operation is the process of traversing the entire list starting from first upto last node in the list.

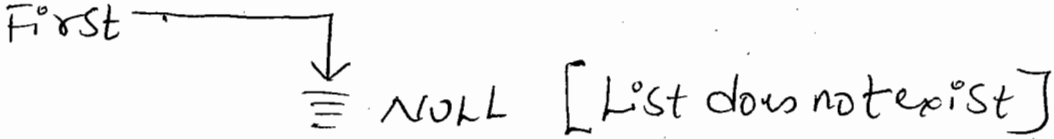
→ If first is NULL no elements or does not exist.

→ If nodes in the list, the Pointer q is Pointing to the first node.

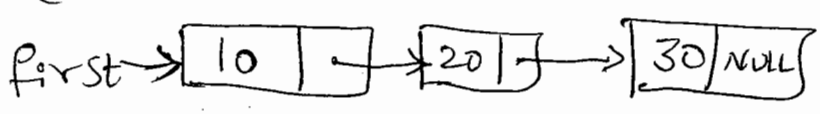
C- function to display Singly linked list:

```
Void display()
{
  Struct node *q;
  if (first == NULL)
  {
     Printf(" List does not exist");
     return;
  }
  else
  {
     q = first;
     while (q != NULL)
     {
        Printf("%d", q->data);
     }
  }
}
```


Case (i) If first == NULL



(ii)



Output: 10 20 30

TREE :-

A Tree is a Non-linear data Structure that represent Parent-Child relationship between the data items Stored in it. A tree Comprises nodes and edges. Here, each node is associated with Some data.

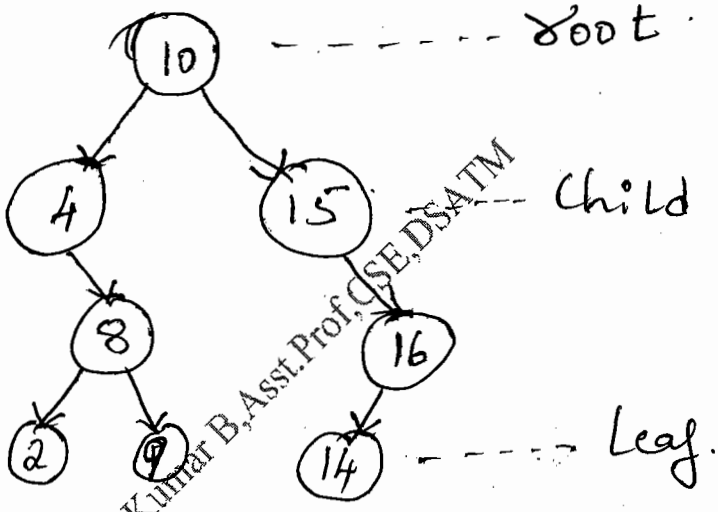


Fig:- Illustrates Tree.

Node :

A node is a Structure that Stores a Value or a Condition. represents a data Structure.

Edge :

A Connection between one node to another node Called as Edge or link.

Root node:

A node that does not have any Parent node called as root node.

Child node:

Each node may have a one or more. Children associated with it., Here a node can have only one Parent.

A child node is below the Parent node and it is connected with an edge.

The different types of Tree data structures are:

* Binary tree
* Binary Search tree.
* AVL tree.
* B Tree.

Binary tree:

A Binary tree is a tree in which each node can have at most two children. These children are distinct. One designated as left child and other designated as right child.

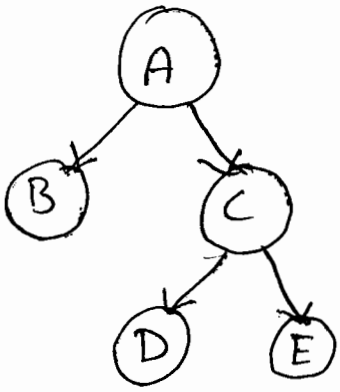


Fig: Binary Tree

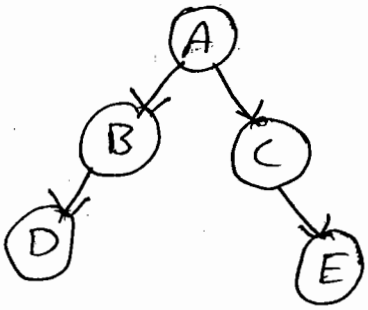


Fig: Not a Binary tree

Advantages Of Trees:

-> Data Can be Stored in hierarchical fashion

-> Searching data becomes Simple.

-> Easy to represent Sorted list of data.