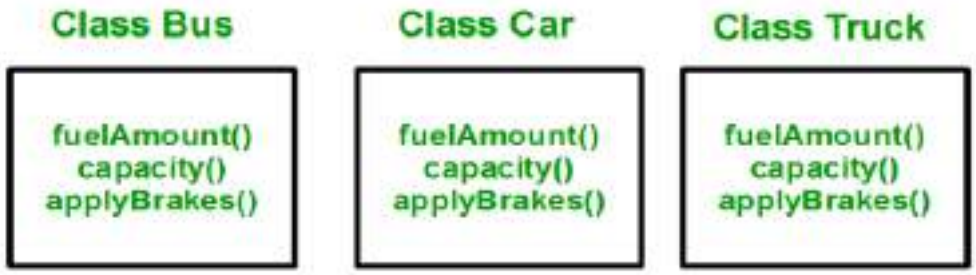
UNIT-2

Inheritance and Polymorphism, Exception Handling, throwing an exception, the try block, catching an exception, exception objects, exception specifications, catching all exceptions, Introduction to linked lists, stacks, queues and applications of stacks.

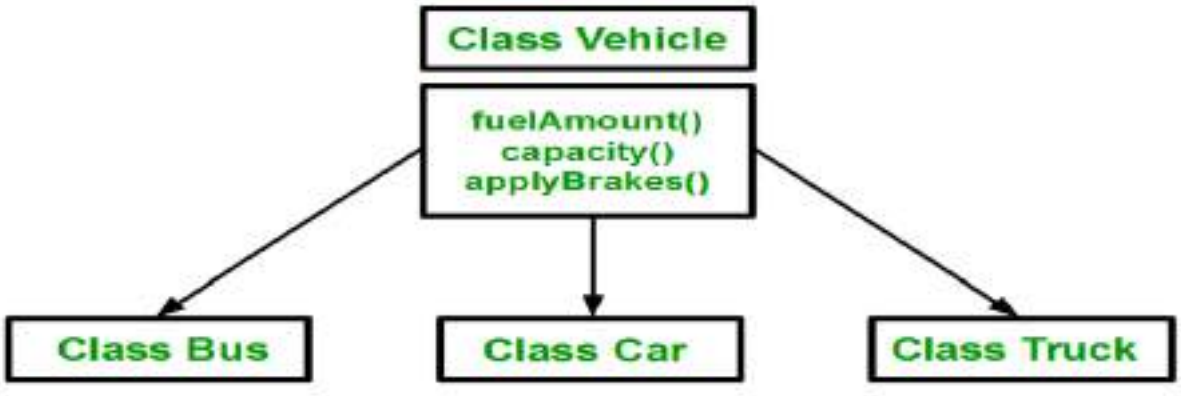
Inheritance

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERITENCE'. This is often referred to as IS-A' relationship because every object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class or 'PARENT' class and the new class is called 'DERIEVED' class or 'CHILD' class.

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming. When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.



The above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++: For creating a **sub-class(child class/Derived class)** that is inherited from the base class we have to follow the below syntax.

Derived Classes:

A Derived class is defined as the class derived from the base class.

Syntax:

```
class <derived_class_name> : <access-specifier> <base_class_name>
{
    // members of derived class


}
```

Where

class — keyword to create a new class

derived_class_name — name of the new class, which will inherit the base class

access-specifier — either of private, public or protected. If neither is specified, **PRIVATE is taken as default**

base-class-name — name of the base class/arentclass/superclass

Note: **A derived class doesn't inherit *access* to private data members.**

However, it does inherit a full parent object, which contains any private members which that class declares.

The colon : indicates that the a-class name is derived from the base class name. **The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private.** Visibility mode describes the status of derived features e.g.

Example:

```
class XYZ                    //base class
{
members of XYZ
};
class ABC : public XYZ    //public derivation
{
members of ABC
};
class ABC : XYZ //private derivation (by default)
{
members of ABC
};
```

In the inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class.

Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Following are the examples of Inheritance:

Example 1

```
#include<iostream>

using namespace std;

class Vehicle                    // Base class

 {
  public:
    string brand = "Ford";
    void honk()

 {
      cout << “honk honk ! \n" ;
    }
};

class Car: public Vehicle       // Derived class

{
  public:
    string model = "Mustang";
};

int main()

 {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " +myCar.model;
  return 0;
}
```

Output:

Honk honk!

Ford Mustang

Example 2: Define member function without argument within the class

```
#include<iostream>

using namespace std;

 class Person     //parentclass

{
```

```
    int id;
    char name[100];
      public:
       void set_p()
       {
          cout<<"Enter the Id:";
          cin>>id;
          cout<<"Enter the Name:";
          cin.get(name,100);
       }
        void display_p()
       {
   cout<<endl<<id<<"\t"<<name<<"\t";
       }
};
  class Student: private Person
{
//childclass
    char course[50];
    int fee;
     public:
    void set_s()
       {
          set_p();
          cout<<"Enter the Course Name:";
          cin.getline(course,50);
          cout<<"Enter the Course Fee:";
          cin>>fee;
       }
     void display_s()
       {
```

```
        display_p();
        cout<<course<<"\t"<<fee<<endl;
    }
};
int main()
{
   Student s;
   s.set_s();
   s.display_s();
   return 0;
}
```

Output:

```
Enter the Id: 101
Enter the Name: Dev
 Enter the Course Name: GCS
Enter the Course Fee:70000
101 Dev GCS 70000
```

Example 3: C++ program to demonstrate implementation of Inheritance

```
 #include <iostream>
using namespace std;
 // Base class
class Parent
 {
public:
   int id_p;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
public:
   int id_c;
```

```
};
// main function
int main()
{
   Child obj1;
   // An object of class child has all data members  and member //functions of class parent
   obj1.id_c = 7;
   obj1.id_p = 91;
   cout << "Child id is: " << obj1.id_c << '\n';
   cout << "Parent id is: " << obj1.id_p << '\n';
    return 0;
     }
```

Output:

Child id is : 7

Parent id is: 91

In the above program, the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

Modes of Inheritance:

There are 3 modes of inheritance.

Public Mode: If we derive a subclass from a public base class. **Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.**

Protected Mode: If we derive a subclass from a Protected base class. **Then both public members and protected members of the base class will become protected in the derived class.**

Private Mode: If we derive a subclass from a Private base class. **Then both public members and protected members of the base class will become Private in the derived class.**

An access declaration takes this general form:

base-class::member;

Example: When a base class is inherited private:

```
class base
{
public:
int j; // public in base
```

};

// Inherit base as private.

class derived: private base

{

public:

// here is access declaration

base::j; // make j public again

.

.

.

};

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

base::j;

as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.

Example: C++ Implementation to show that a derived class doesn't inherit access to private data members. However, it does inherit a full parent object.

```
class A
{
public:
    int x;

protected:
    int y;

private:
    int z;
};

class B : public A
{
    // x is public
```

```
    // y is protected
    // z is not accessible from B
};
        class C : protected A
       {
    // x is protected
    // y is protected
    // z is not accessible from C
        };

class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

Private: when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The private and protected members of a class can be accessed by:

a. A function i.e. friend of a class.

b. A member function of a class that is the friend of the class.

c. A member function of a derived class.

Types of Inheritance in C++:-

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

Syntax for Single Inheritance:

class subclass_name : access_mode base_class

{ // body of subclass

};

Following are the examples to implement Single Inheritance:

Example 1:

```
#include<iostream>

using namespace std;

class Vehicle          // base class

 {
```

```
public:
        Vehicle()
        {
        cout << "This is a Vehicle\n";
        }
};
// sub class derived from a single base classes
class Car : public Vehicle
{
};
int main()          // main function
{
// Creating object of sub class will
// invoke the constructor of base classes

Car obj;
return 0;
}
```

Output This is a Vehicle

Example 2:

```
#include<iostream.h>
#include<conio.h>
class worker
{
int age;
char name [10];
public:
void get ( );
void show( );
};
```

```
void worker : : get ( )
{
cout <<"yout name please"
cin >> name;
cout <<"your age please" ;
cin >> age;
}
void worker :: show ( )
{
cout <<"My name is :"<<name<< endl;
cout <<"My age is:"<< age<<endl;
}
class manager : public worker    //derived class (publicly)
{
int n;
public:
void get ( ) ;
void show ( ) ;
};
void manager : : get ( )
{
worker : : get ( ) ;       //the calling of base class input function
cout << "number of workers under you";
cin >> n;
}
//( if they were public )
void manager :: show ( )
{
worker :: show ( );   //calling of base class o/p fn.
cout <<"in No. of workers under me are: " << n;
}
```

```
 int main ( )
{
clrscr ( ) ;
worker W1;
manager M1;
M1 .get ( );
M1.show ( ) ;
}
```

If you input the following to this program:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

Example 3:

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker            //Base class declaration
{
int age;
char name [10] ;
public:
void get ( ) ;
void show ( ) ;
```

```
};
void worker : : get ( )
{
cout << “your name please” ;
cin >> name;
cout << “your age please”;
cin >>age;
}
void worker : show ( )
{
cout << “in my name is: “ <<name<< “in” << “my age is : “ <<age;
}
class manager : worker       //Derived class (privately by default)
{
int now;
public:
void get ( ) ;
void show ( ) ;
};
void manager : : get ( )
{
worker : : get ( ); //calling the get function of base
cout << “number of worker under you”; class which is
cin >> now;
}
void manager : : show ( )
{
worker : : show ( ) ;
cout << “in no. of worker under me are : “ <<now;
}
main ( )
```

```
{
clrscr ( ) ;
 worker wl ;
 manager ml;
ml.get ( ) ;
ml.show ( );
}
```

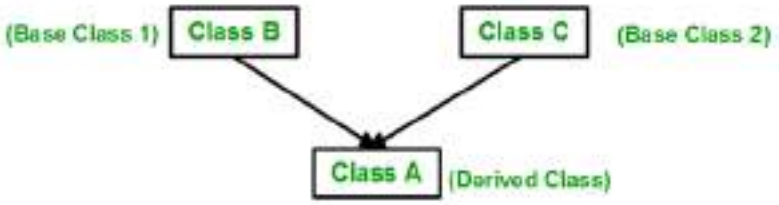
Example 4:

The following program shows the single inheritance using protected derivation

```
#include<conio.h>
#include<iostream.h>
class worker                    //Base class declaration
{
 protected:
int age; char name [20];
public:
void get ( );
void show ( );
};
void worker :: get ( )
{
cout >> "your name please";
cin >> name;
cout << "your age please";
cin >> age;
}
void worker :: show ( )
{
cout << "in my name is: " << name << "in my age is " <<age;
}
```

```
class manager:: protected worker      // protected inheritance
{
int now;
public:
void get ( );
void show ( ) ;
};
void manager : : get ( )
{
cout << “please enter the name In”;
cin >> name;
cout<< “please enter the age In”; //Directly inputting the data
cin >> age; members of base class
cout << “ please enter the no. of workers under you:”;
cin >> now;
}
void manager : : show ( )
{
cout « "your name is : "«name«" and age is : "«age; cout «"In no. of workers under your are : "«now;
main ( )
{
clrscr ( ) ;
manager ml;
ml.get ( ) ;
cout « "\n \n";
ml.show ( );
}
```

2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



Syntax:

class subclass_name : access_mode base_class1, access_mode base_class2, ...

{

// body of subclass

};

Example:

class B

{

... .

};

class C

{

... .

};

class A: public B, public C

{

... ...

};

Here, the number of base classes will be separated by a comma (', ') and the access mode for every base class must be specified.

Example to demonstrate multiple inheritance:

```
#include <iostream>

using namespace std;

class Vehicle      // first base class

{

public:
```

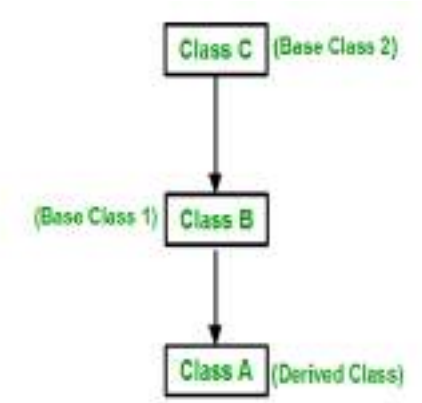
```
    Vehicle()
{
cout << "This is a Vehicle\n";
}
};
 class FourWheeler     // second base class
{
public:
    FourWheeler()
    {
 cout << "This is a 4 wheeler Vehicle\n";
    }
};
// sub class derived from two base classes
class Car : public Vehicle,public FourWheeler
{
};
 int main()  // main function
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

This is a Vehicle

This is a 4 wheeler Vehicle

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



Syntax:

class C

{

... .

};

class B:public C

{

... .

};

class A: public B

{

... ...

};

Example1 to demonstrate Multilevel Inheritance:

```
#include<iostream>
using namespace std;
class A
{
protected:
int a;
public:
void set_A()
```

```
{
cout<<"Enter the Value of A=";
cin>>a;
}
void disp_A()
{
cout<<endl<<"Value of A="<<a;
}
};
class B: public A
{
protected:
int b;
public:
void set_B( )
{
cout<<"Enter the Value of B=";
cin>>b;
}
void disp_B( )
{
cout<<endl<<"Value of B="<<b;
}
};
class C: public B
{
int c,p;
public:
void set_C()
{
cout<<"Enter the Value of C=";
```

```
cin>>c;
}
void disp_C()
{
cout<<endl<<"Value of C="<<c;
}
void cal_product()
{
p=a*b*c;
cout<<endl<<"Product of "<<a<<" * "<<b
<<" * "<<c<<" = “<<p;
}
};
int main()
{
C  objc;
objc.set_A();
objc.set_B();
objc.set_C();
objc.disp_A();
objc.disp_B();
objc.disp_C();
objc.cal_product();
return 0;
}
```

Example 2 to implement Multilevel Inheritance:

```
#include <iostream>
using namespace std;
class Vehicle          // base class
{
public:
```

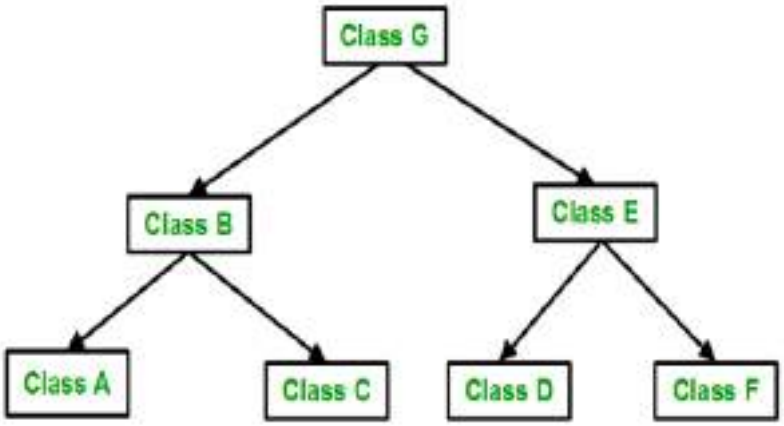
```
Vehicle()
{
 cout << "This is a Vehicle\n";
}
};
// first sub_class derived from class vehicle
class fourWheeler : public Vehicle
 {
public:
fourWheeler()
{
cout << "Objects with 4 wheels are  vehicles\n";
}
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler
 {
public:
Car()
 {
 cout << "Car has 4 Wheels\n";
 }
};
int main()      // main function
{
Car obj; // Creating object of sub class will // invoke the constructor of    //base classes first and then its own class.
return 0;
}
```

Output: This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

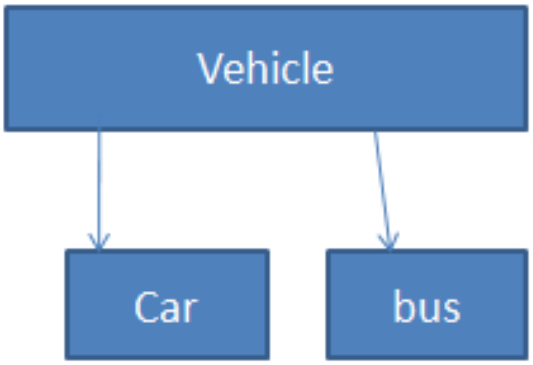
4. **Hierarchical Inheritance**: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:

```
class A
{
// body of the class A.
}
class B : public A
{
// body of class B.
 }
class C : public A
{
// body of class C.
}
class D : public A
{
// body of class D.
}
```

Example to implement Hierarchical Inheritance:




```
#include <iostream>
using namespace std;
class Vehicle    // base class
{
 public:
Vehicle()
 {
 cout << "This is a Vehicle\n";
 }
};
class Car : public Vehicle     // first sub class
{
};
class Bus : public Vehicle     // second sub class
{
};
int main()                    // main function
{
// Creating object of sub class will
// invoke the constructor of base class.
Car obj1;
Bus obj2;
```

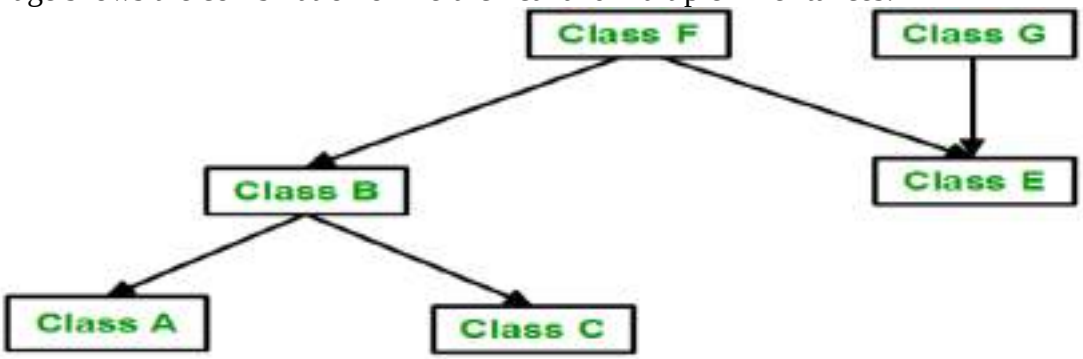
return 0;

}

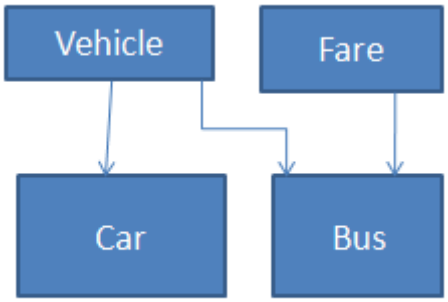
Output: This is a Vehicle

This is a Vehicle

5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritances:



Example 1 to implement Hybrid Inheritance:



#include <iostream>

using namespace std;

class Vehicle **// base class**

{

public:

Vehicle()

{

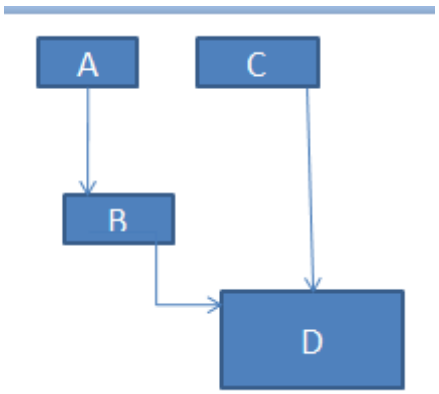
cout << "This is a Vehicle\n";

```
}
};

class Fare     // base class
{
public:
Fare()
{
cout << "Fare of Vehicle\n";
}
};
class Car : public Vehicle   // first sub class
{
};
class Bus : public Vehicle, public Fare  // second sub class
{
};

int main() // main function
{
        // Creating object of sub class will
        // invoke the constructor of base class.
        Bus obj2;
        return 0;
}
```

Output

```
This is a Vehicle
 Fare of Vehicle
```

Example 2 to demonstrate Hybrid Inheritance:




```
#include <iostream>
using namespace std;
class A
{
        protected:
        int a;
        public:
        void get_a()
        {
        cout << "Enter the value of 'a' : ";
        cin>>a;
        }
};
class B : public A
{
        protected:
        int b;
        public:

        void get_b()
        {
          cout << "Enter the value of 'b' : ";
        cin>>b;
        }
```

```
};
class C
{
        protected:
        int c;
        public:
        void get_c()
        {
        cout << "Enter the value of c is : ";
        cin>>c;
        }
};
class D : public B, public C
{
        protected:
        int d;
        public:
        void mul()
        {
        get_a();
        get_b();
        get_c();
cout << "Multiplication of a,b,c is : " <<a*b*c;
        }
};
int main()
{
        D  objd;
        objd.mul();
        return 0;
}
```

Output:

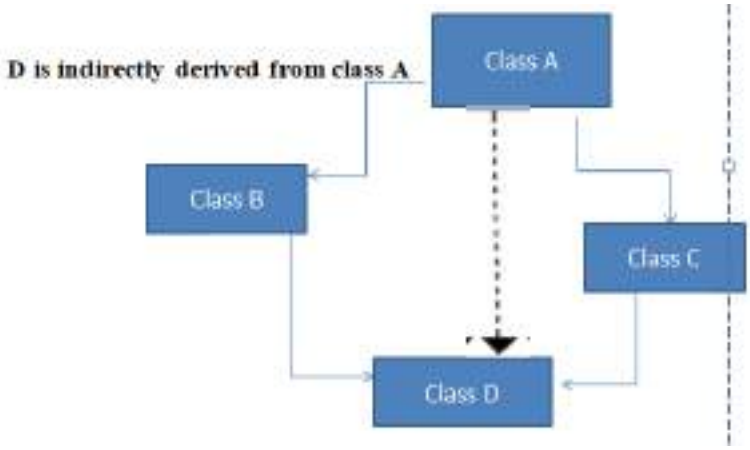
Enter the value of a: 10

Enter the value of b:5

Enter the value of c: 2

Multiplication of a,b,c is:100

6. **A special case of hybrid inheritance: Multipath inheritance**: A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. This type of inheritance involves other inheritance like multiple, multilevel, hierarchical etc. Ambiguity can arise in this type of inheritance.



Example 1to implement Multipath Inheritance :

```
#include <iostream>
using namespace std;
class ClassA
{
public:
    int a;
};
class ClassB : public ClassA
{
public:
    int b;
};
class ClassC : public ClassA
 {
public:
```

```
        int c;
};
class ClassD : public ClassB, public ClassC
{
public:
        int d;
};
int main()
{
ClassD obj;
// obj.a = 10;   // Statement 1, Error
// obj.a = 100;  // Statement 2, Error
obj.ClassB::a = 10;    // Statement 3,valid
obj.ClassC::a = 100;    // Statement 4, valid
obj.b = 20;
obj.c = 30;
obj.d = 40;
cout << " a from ClassB : " << obj.ClassB::a;
cout << "\n a from ClassC : " << obj.ClassC::a;
cout << "\n b : " << obj.b;
cout << "\n c : " << obj.c;
cout << "\n d : " << obj.d << '\n';
}
```

Output :

```
a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40
```

There are 2 Ways to Avoid this Ambiguity:

1) **Avoiding ambiguity using the scope resolution operator:** Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

```
obj.ClassB::a = 10;        // Statement 3
obj.ClassC::a = 100;       // Statement 4
```

Note: Still, there are two copies of Class A in Class-D.

2) Avoiding ambiguity using the virtual base class:

```
#include<iostream>
using namespace std;
class ClassA
{
public:
    int a;
};
class ClassB : virtual public ClassA
{
public:
    int b;
};
class ClassC : virtual public ClassA
{
public:
    int c;
};
class ClassD : public ClassB, public ClassC
{
public:
    int d;
};
int main()
{
```

```
    ClassD obj;
    obj.a = 10;      // Statement 3
    obj.a = 100;     // Statement 4
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output:

```
a : 100
b : 20
 c : 30
 d : 40
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

Note:

- When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

Virtual Base Classes

Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes ‘parent1’ and ‘parent2’ which themselves has a common base class ‘grandparent’. The child inherits the traits of ‘grandparent’ via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as ‘INDIRECT BASE CLASS’. Now, the inheritance by the child might cause some problems. All the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via ‘parent2’. So, there occurs a duplicacy which should be avoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class:

Syntax to declare Virtual Base class:

```
class g_parent
{
//Body
};
class parent1: virtual public g_parent
{
// Body
};
class parent2: public virtual g_parent
{
// Body
};
class child: public parent1, public parent2
{
// body
};
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class.

Note that keywords 'virtual' and 'public' can be used in either order .

Example to show the demonstration of virtual base class:

```
#include<iostream.h>
#include<conio . h>
class student  // Base class declaration
{
protected:
 int r_no;
 public:
void get_n (int a)
```

```
{
r_no = a;
}
void put_n (void)
{
cout << “Roll No. “ << r_no<< “ln”;}
};
class test : virtual public student // Virtually declared common base class 1
 {
protected:
int part1;
int part2;
public:
void get_m (int x, int y)
 {
 part1= x;
part2=y;
}
void putm (void)
{
cout << “marks obtained: “ << “\n”;
cout << “part1 = “ << part1 << “\n”;
cout << “part2 = “<< part2 << “\n”;
}
};
class sports : public virtual student
// virtually declared common
{ //base class 2
protected:
int score;
public:
```

```
void get_s (int a)
 {
score = a ;
}
void put_s (void)
{
cout << “sports wt.: “ <<score<< “\n”;
}
};
class result: public test, public sports                //derived class
{
private: int total ;
public:
void show (void) ;
};
void result : : show (void)
{
total = part1 + part2 + score ;
put_n ( );
put_m ( );
put_s ( ) ; cout << “\n total score= “ <<total<< “\n” ;
}
int main ( )
{
result r ;
r.get_n (345)
r.get_m (30, 35) ;
r.get_s(7) ;
r. show ( ) ;
return 0;
}
```

Example to show hybrid inheritance using virtual base classes :

```
#include<iostream>
using namespace std;
Class A
{
protected:
int x;
 public:
void get (int) ;
void show (void) ;
};
void A : : get (int a)
{
x = a ;
}
void A : : show (void)
{
cout << X ;
}
Class A1 : Virtual Public A
{
protected:
int y ;
public:
void get (int) ;
void show (void);
};
void A1 :: get (int a)
{
y = a;
}
```

```
void A1 :: show (void)
{
cout <<y ;
}
class A2 : Virtual public A
{
protected:
int z ;
public:
void get (int a)
{
z =a;
}
void show (void)
{
 cout << z;
}
};
class A12 : public A1, public A2
{
int r, t ;
public:
void get (int a)
{
 r = a;
}
void show (void)
{
 t = x + y + z + r ;
cout << “result =” << t ;
}
```

```
};
int main ( )
{
A12 r ;
r.A : : get (3) ;
r.A1 : : get (4) ;
r.A2 : : get (5) ;
r.get (6) ;
r . show ( ) ;
return 0;
}
```

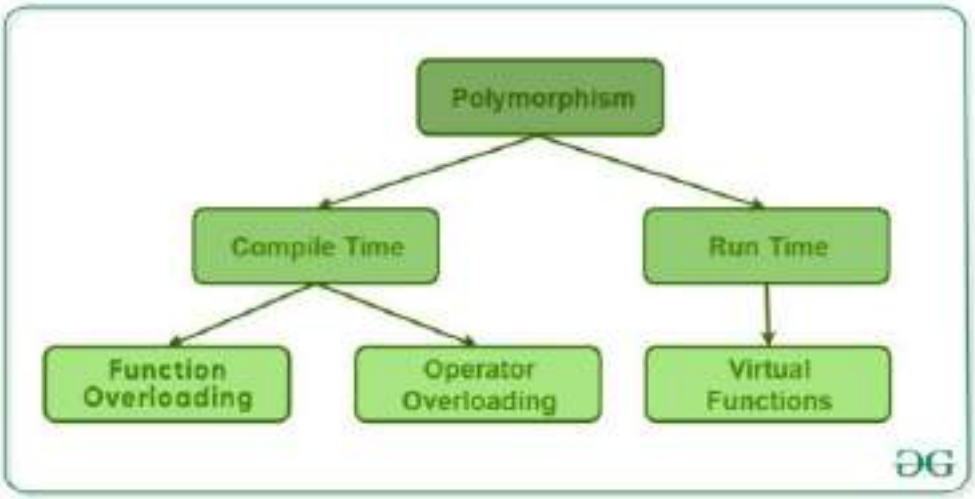
Polymorphism

Introduction:

The word "polymorphism" means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism:

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



Types of Polymorphism

1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading. Functions can be overloaded by **changing the number of** arguments or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name.

Example to demonstrate function overloading or Compile-time Polymorphism

```
#include <bits/stdc++.h>

using namespace std;

class Poly

{

public:

        // Function with 1 int parameter

        void func(int x)

        {

                cout << "value of x is " <<

                                x << endl;

        }

        // Function with same name but  1 double parameter

        void func(double x)

        {

        cout << "value of x is " <<x << endl;

        }



// Function with same name and 2 int parameters

        void func(int x, int y)

        {

                cout << "value of x and y is " <<

                                x << ", " << y << endl;

        }

};

// Driver code
```

```
int main()
{
        Poly obj1;

        // Function being called depends on the parameters passed
        // func() is called with int value
        obj1.func(7);  // func() is called with double value
        obj1.func(9.132);       // func() is called with 2 int values
        obj1.func(85, 64);
        return 0;
}
```

Output

value of x is 7

value of x is 9.132

value of x and y is 85, 64

B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Example to demonstrate Operator Overloading or Compile-Time Polymorphism:

```
#include <iostream>
using namespace std;
class Complex
 {
private:
        int real, imag;
public:
        Complex(int r = 0,
                        int i = 0)
```

```
        {
                real = r;
                imag = i;
        }
// This is automatically called when '+' is used with between two //Complex objects
Complex operator+(Complex const& obj)
        {
                Complex res;
                res.real = real + obj.real;
                res.imag = imag + obj.imag;
                return res;
        }
        void print()
        {
        cout << real << " + i" <<
                        imag << endl;
        }
};
// Driver code
int main()
{
        Complex c1(10, 5), c2(2, 4);
        // An example call to "operator+"
        Complex c3 = c1 + c2;
        c3.print();
}
```

Output

12 + i9

2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Syntax to override a function:

```
class parent
{
public:
void fun()

{

Statements;

}

class child:public parent

{

public:

void fun()

{

statements;

}

int main()

{

child obj;

obj.fun();

return 0;

}
```

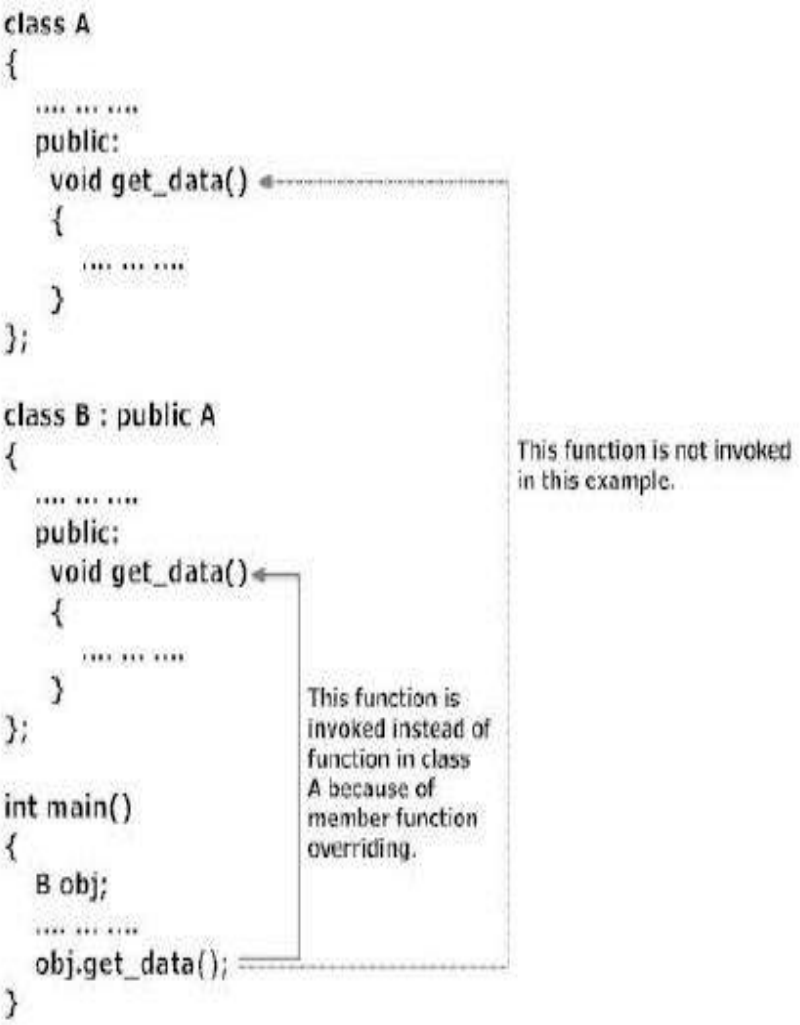


Figure: Member Function Overriding in C++

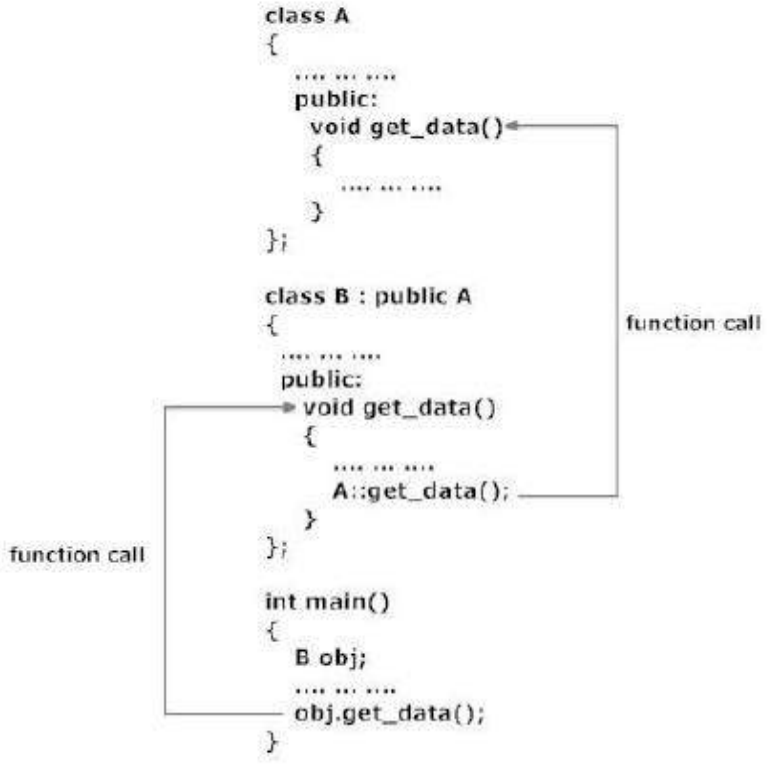
Accessing the Overridden Function of Base Class From Derived Class

To access the overridden function of base class from derived class, **scope resolution operator :: is used.**

For example: If you want to access get_data() function of base class from derived class in above example then, the following statement is used in derived class.

A::get_data; // Calling get_data() of class A.

It is because, if the name of class is not specified, the compiler thinks get_data() function is calling itself.



Example:

Example to demonstrate Function Overriding

```
#include <bits/stdc++.h>

using namespace std;

class Parent

{

public:

        void fun()

        {

            cout << "Base Function" << endl;

        }

};

class Child : public Parent

 {
```

```
public:
        void fun()
        {
            cout << "Derived Function" << endl;
        }
};
int main()
{
        Child obj
        obj.fun();
        return 0;
}
```

Output
Derived Function

B.Virtual Function

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Polymorphism is also accomplished using pointers in C++. **It allows a pointer in a base class to point to either a base class object or to any derived class object.**

```
class base
{
//Data Members
//Member Functions
};
class derived : public base
```

{

//Data Members

//Member functions

};

void main ( )

{

base *ptr; //pointer to class base

derived obj ;

ptr = &obj ; //indirect reference obj to the pointer

//Other Program statements

}

The pointer ptr points to an object of the derived class obj.

But, a pointer to a derived class object may not point to a base class object without explicit casting.

For example, the following assignment statements are not valid

void main ( )

{

base obj;

derived *ptr;

ptr = &obj; //invalid.... .explicit casting required

//Other Program statements

}

A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.

void main ( )

{

base obj ;

derived *ptr; // pointer of the derived class

ptr = (derived *) & obj; //correct reference

//Other Program statements

}

The general syntax of the virtual function declaration is:

class classname

{

private:

public:

virtual return_type function_name1 (arguments);

virtual return_type function_name2(arguments);

virtual return_type function_name3(arguments);

};

NOTE:

- A destructor member function does not take any argument and no return type can be specified for it not even void.
- It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.
    - Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declared virtual.

Example to demonstrate the Virtual Function:

```
#include <iostream>
using namespace std;
// Declaring a Base class
class A {

public:
        // virtual function
        virtual void display()
        {
cout << "Called virtual Base Class function" <<"\n\n";
        }

        void print()
        {
cout << "Called class A print function" <<"\n\n";
        }
};

// Declaring a Child Class
class B : public A
{

public:
```

```
        void display()
        {
cout << "Called B Display Function" <<"\n\n";
        }

        void print()
        {
cout << "Called B print Function" <<"\n\n";
        }
};

// Driver code
int main()
{

        A* base;

        B child;

        base = &child;

        // This will call the virtual function
        base->A::display();

        // this will call the non-virtual function
        base->print();
}
```

Output
Called virtual Base Class function

Called B print function

Virtual destructors

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors can not be virtual. Virtual Destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructor of all its base classes are invoked. If destructor is made as non virtual destructor in the base class, only the base class's destructor is invoked when the object is deleted.

Example to demonstrate Virtual Destructors:

```
#include<iostream.h>

#include<string.h>

class father

{

protected:

char *fname;
```

```
public:
father(char *name)
{
fname=new char(strlen(name)+1);
strcpy(fname,name);
}
virtual ~father()
{
delete fname;
cout<<"~father is invoked…";
}
virtual void show()
{
cout<<"father name…"<<fname;
}
};
class son: public father
{
protected:
char *s_name;
public:
son(char *fname,char *sname):father(fname)
{
sname=new char[strlen(sname)+1];
strcpy(s_name,sname);
}
~son()
{
delete s_name;
cout<<"~son() is invoked"<<endl;
}
```

```
void show()
{
cout<<"father's name"<<fname;
cout<<"son's name:"<<s_name;
}
};
void main()
{
father *basep;
basep =new father ("Ben");
cout<<"basep points to base object…"
basep->show();
delete basep;
basep=new son("David","Ben");
cout<<"base points to derived object…";
basep->show();
delete basep;
}
```

Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task. Pure virtual Functions are virtual functions with no definition. **They start with virtual keyword and ends with = 0**. **Here is the syntax for a pure virtual function,**

```
virtual void f() = 0;
```

Example to demonstrate Pure Virtual Function :

The class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class. The methods of the derived class are invoked by the pointer to the base class.

Example:

```
#include<iostream>
using namespace std;
class employee
 {
```

```
int code
char name [20] ;
public:
virtual void getdata ( ) ;
virtual void display ( ) ;
};
class grade: public employee
{
char grd [90] ;
float salary;
public :
void getdata ( ) ;
void display ( );
};
void employee :: getdata ( )
{
}
void employee:: display ( )
{
}
void grade : : getdata ( )
{
cout<< " enter employee's grade ";
cin> > grd ;
cout<< "\n enter the salary " ;
cin>> salary;
}
void grade : : display ( )
{
cout« " Grade salary \n";
cout« grd« " "« salary« endl;
```

```
}
int main ( )
{
employee *ptr ;
grade obj ;
ptr = &obj ;
ptr->getdata ( ) ;
ptr->display ( ) ;
return 0;
}
```

Output

```
enter employee's grade A
enter the salary 250000
Grade salary
A 250000
```

Abstract Class

- **Abstract Class is a class which contains atleast one Pure Virtual function in it.**
- **Abstract classes are used to provide an Interface for its sub classes.**
- **Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.**

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

2. Abstract class can have normal functions and variables along with a pure virtual function.

3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example to demonstrate Abstract Class

```
class Base              //Abstract base class
{
public:
virtual void show() = 0;    //Pure Virtual Function
```

```
};
class Derived:public Base
{
public:
void show()
{
cout << "Implementation of Virtual Function in Derived class"; }
};
int main()
{
// Base obj;                    //Compile Time Error
Base *b;
Derived d;
b = &d;
b->show();
}
```

Output : Implementation of Virtual Function in Derived class

Exception Handling

Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism of C++ is generally referred to as exception handling. Generally , exceptions are classified into synchronous and asynchronous exceptions.

The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.

Examples of Synchronous Exception: Errors such as out of range, overflow, underflow and so on.

The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.

Examples of Asynchronous Exception include errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

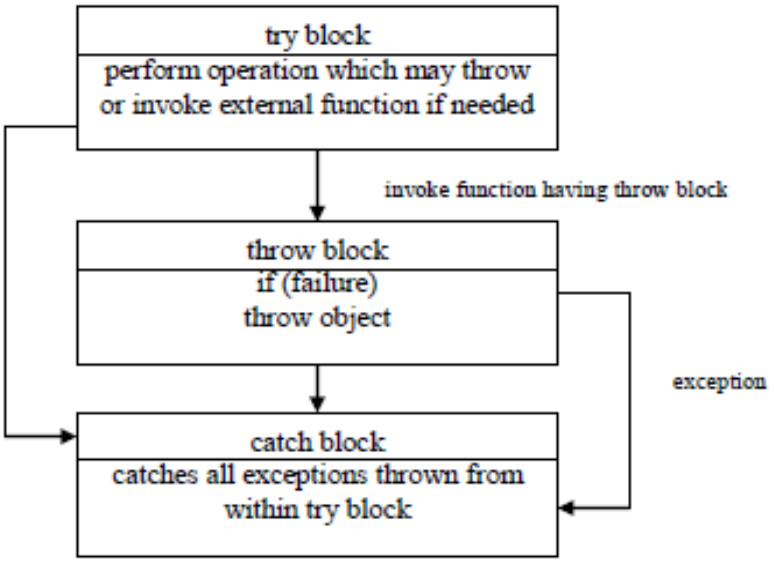
Exception Handling model:

When a program encounters an abnormal situation for which it in not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.

The exception handling mechanism uses three blocks: try, throw and catch.

The try block must be followed immediately by a handler, which is a catch block. If an exception is thrown in the try block the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function.

The relationship of these three exceptions handling constructs called the exception handling model is shown in figure**:**




```
try
{
    - - - - - - - - - - -
    - - - - - - - - - - -
    throw val
    - - - - - - - - - - -
    - - - - - - - - - - -
}
catch( data-type  arg)
{
    - - - - - - - - - - -
    - - - - - - - - - - -
    - - - - - - - - - - -
}
```

throws exception value

throw construct

The keyword throw is used to raise an exception when an error is generated in the comutation. **the throw expression initialize a temporary object of the typeT used in throw (T arg).**

Syntax:

throw T;

catch construct

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch Each handler will only evaluate an exception that matches.

Syntax:

catch(T)

{

// error messages

}

try construct

The try keyword defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted.

Syntax:

try

{

…

if (failure)

throw T;

}

catch(T)

{

…

}

Example to demonstrate Divide by Zero Exception:

```
#include<iostream>

using namespace std;

void main()

{

int a,b;

cout<<"enter two numbers:";
```

```
cin>>a>>b;
try
{
if (b= =0)
throw b;
else
cout<a/b;
}
catch(int x)
{
cout<<"2nd operand can't be 0";
}
}
```

Example to demonstrate Array reference out of bound exception:

```
#define max 5
class array
{
private:
int a[max];
public:
int &operator[](int i)
{
if (i<0 || i>=max)
throw i;
else
return a[i];
}
};
void main()
{
array x;
```

```
try
{
cout<<"trying to refer a[1]…"
x[1]=3;
cout<<"trying to refer a[13]…"
x[13]=5;
}
catch(int i)
{
cout<<"out of range in array references…";
}
}
```

Multiple catches :

When an exception is thrown, the exception handler are searched in order for an appropriate match.

• It is possible that arguments of several catch statements match the type of an exception. In such cases the first handler that matches the exception type is executed

Example to implement multiple catches :

```
#include<iostream>
using namespace std;
void test(int x)
{
try
{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw 's';
}
catch (int i)
```

```
{
cout<<"caught an integer…";
}
catch (float s)
{
cout<<"caught a float…";
}
catch (char c)
{
cout<<"caught a character…";
}
}
void main()
{
test(1);
test(-1);
test(0);
}
```

Program to implement catch all

```
#include<iostream>
using namespace std;
 void test(int x)
{
try
{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw 's';
```

}

catch (…)

{

cout<<"caught an error…";

}

Exception Objects in C++

- The exception object holds the error information about the exception that had occurred. The information includes the type of error i.e., logic error or run time error and state of the program when the error occurred.
- An exception object is created as soon as exception occurs and it is passed to the corresponding catch block as a parameter.
- This catch block contains the code to catch the occurred exception. It uses the methods of exception object for handling the exceptions.

Syntax :

try

{

throw exception object;

}

catch (Exception &exceptionobject)

{

}

To retrieve the message attached to the exception, exception object uses a method called what (). This method returns the actual messages.

Syntax for what () Method

exceptionobject.what();

To retrieve the message attached to the exception, exception object uses a method called what (). This method returns the actual messages.

Example to demonstrate exception object:

#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception

```
{
public:
const char * what() const throw()
{
return" divide by zero exceptionl\n";
}
};
int main()
{
try
{
int x, y;
cout << "Enter the two numbers :. \n";
cin>>x>>y;
if(y==0)
{
MyException z;
throw z;
}
else
{
cout <<"x/y =" <<x/y << endl;
}
}
catch(exception &e)
{
cout<<e.what();
}
}
```

Output

Exception Specification

Exception specification is a feature which specifies the compiler about possible exception to be thrown by a function.

Syntax

return_type FunctionName(arg_list) throw (type_list)

In the above syntax 'return_type' represents the type of the function.

FunctionName represents the name of the function, arg_list indicates list of arguments passed to the function, throw is a keyword used to throw an exception by function and type_list indicates list of exception types.

An empty exception specification can indicates that a function cannot throw any exception.

If a function throws an exception which is not listed in exception specification then the unexpected function is called. This function terminates the program. In the function declaration, if the function does not have any exception specification then it can throw any exception.

Examples:

1. void verify(int p) throw(int)

In the above example, void is the return type, and verify() is the name of the function, int p is the argument. It is followed by exception specification i.e., throw(int), the function verify() is throws an exception of type 'int'.

2. void verify(int p) throw()

Here, the function can throw any exception because it has empty exception specification.

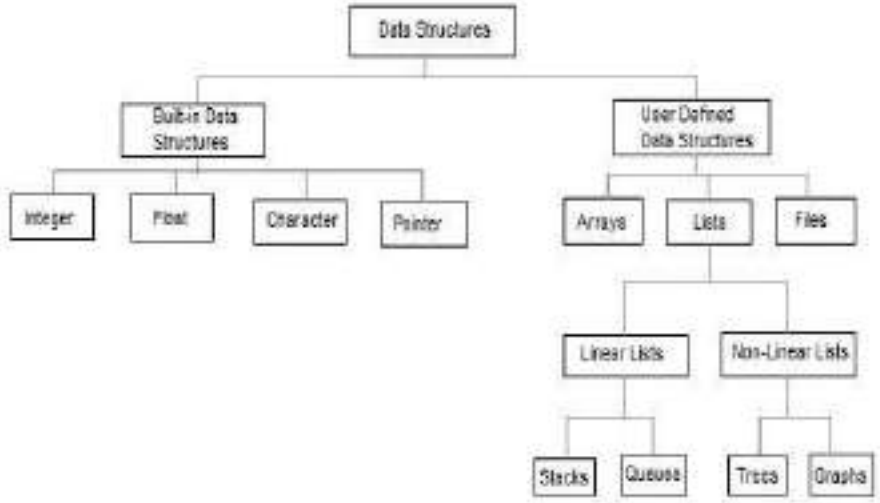
Program to demonstrate Exception specification:

```
#include<iostream>

#include<exception>

using namespace std;

void verify(int p) throw(int)

{

if(p==1)

throw p;
```

```
cout<<"\nEnd of verify().function( )";
}
int main()
{
try
{
cout<<p==1\n";
verify(1);
}
catch(int i)
{
cout<<"Interger exception is caught \n";
}
cout<<"\nEnd of main() function";
//getch();
return 0;
}
```

Data structure

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.



Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

When a class is used as a type, it is an abstract type that refers to a hidden representation.

In this model an ADT is typically implemented as a class, and each instance of the ADT is usually an object of that class. In ADT all the implementation details are hidden

- Linear data structures are the data structures in which data is arranged in a list or in a sequence.
- Non linear data structures are the data structures in which data may be arranged in a hierarchial manner.

LIST ADT

List is basically the collection of elements arranged in a sequential manner. In memory we can store the list in two ways:

- One way is we can store the elements in sequential memory locations. That means we can store the list in arrays.
- The other way is we can use pointers or links to associate elements sequentially. This is known as linked list.

LINKED LISTS

- The linked list is very flexible dynamic data structure : items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

The linked allocation has the following draw backs:

1. No direct access to a particular element.
2. Additional memory required for pointers.

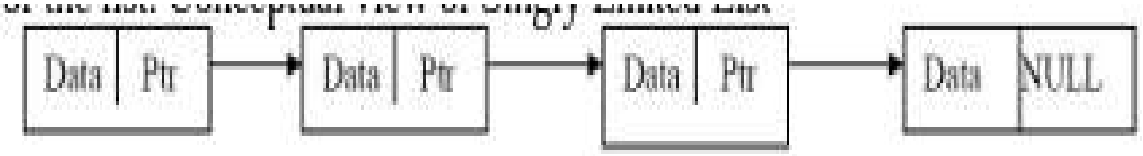
Linked list are of 3 types:

1. Single Linked List
2. Double Linked List
3. Circular Linked List

SINGLE LINKED LIST

A single linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as linear linked list.

- Each item in the list is called a node.
- * Each node of the list has two fields:
- 1. Information- contains the item being stored in the list.
- 2. Next address- contains the address of the next item in the list.
- * The last node in the list contains NULL pointer to indicate that it is the end of the list.
- Conceptual view of Singly Linked List



-

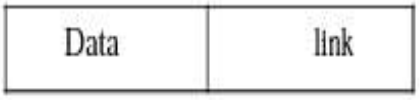
Operations on Single linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

Structure of a node:

Method -1:

```
struct node
{
        int data;
        struct node *link;
};
```



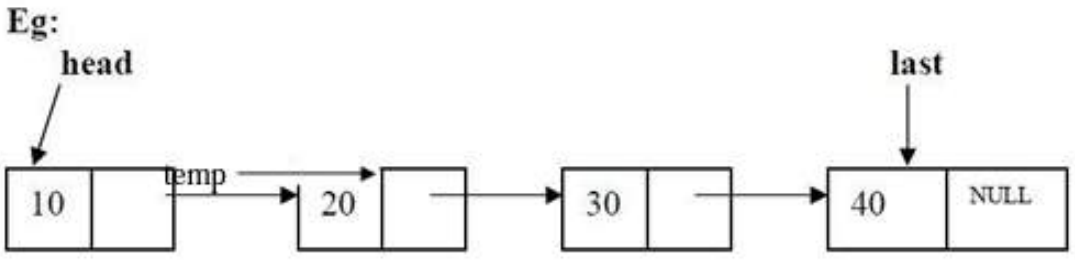
Method -2:

```
class node
{
public:
        int data;
        node *link;
};
```

Insertions: To place an elements in the list there are 3 cases :

1. At the beginning
2. End of the list
3. At a given position

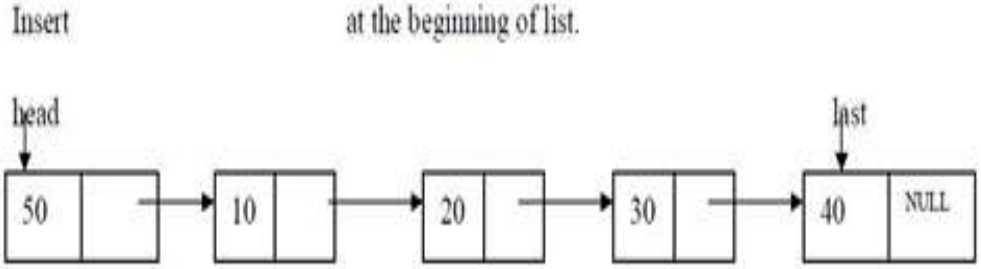
case 1:Insert at the beginning



head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->link=head;
head=temp;
```

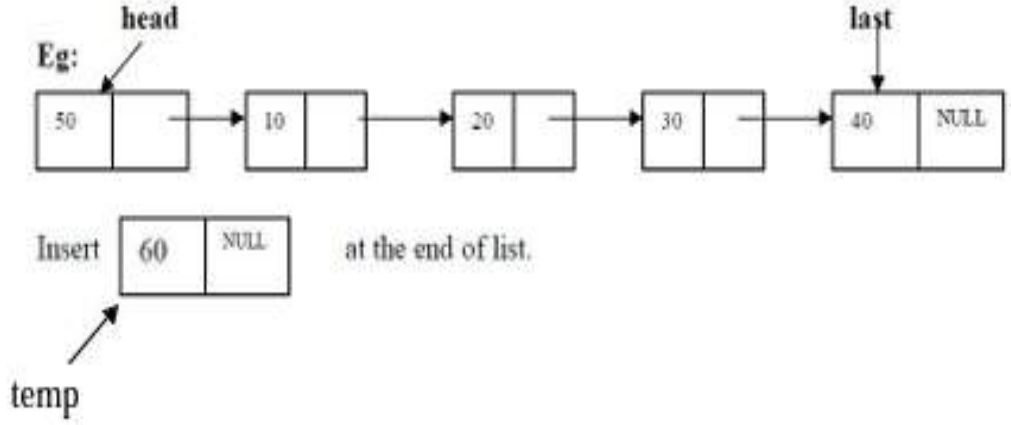
After insertion:



Code for insert front:-

```
template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>item;
        temp=create_node(item);
        if(head==NULL)
                head=temp;
        else
        {temp->link=head;
                head=temp;
        }
}
```

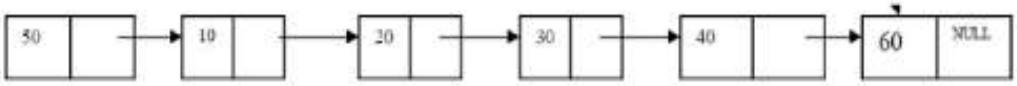
case 2: Inserting end of the list



head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
t=head;
while(t->link!=NULL)
{
        t=t->link;
}
t->link=temp;
```

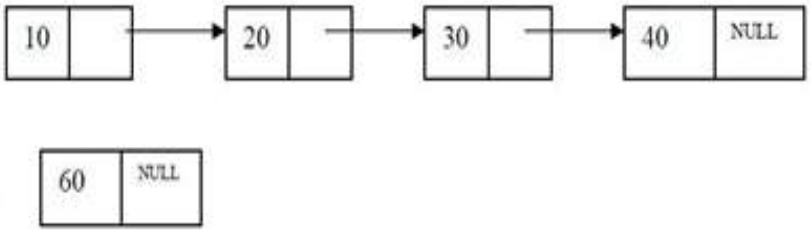
After insertion the linked list is



Code for insert End:-

```
template <class T>
void list<T>::insert_end()
{
struct node<T> *t,*temp;
int n;
        cout<<"Enter data into node:";
        cin>>n;
        temp=create_node(n);
        if(head==NULL)
                head=temp;
        else
        {t=head; while(t-
                >link!=NULL)
                        t=t->link;
                t->link=temp;
        }
}
```

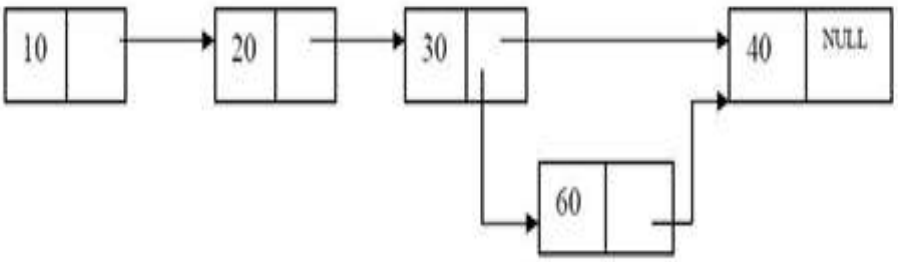
case 3: Insert at a position



insert node at position 3

head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
c=1;
while(c<pos)
{
        prev=cur;
        cur=cur->link;
        c++;
}
prev->link=temp;
temp->link=cur;
```



Code for inserting a node at a given position:-

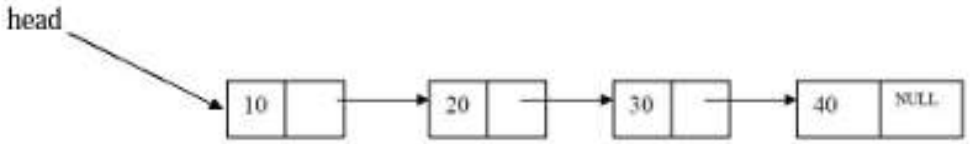
```
template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
        cout<<"Enter data into node:";
        cin>>item
        temp=create_node(item);
        if(head==NULL)
                head=temp;
         else
         {
                 prev=cur=head;
                if(pos==1)
                {
                        temp->link=head;
                        head=temp;
                }
                else
                {
                while(c<pos)
                {c++;
                        prev=cu
                        r;
                        cur=cur->link;
                }
                prev->link=temp;
                temp->link=cur;
                }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

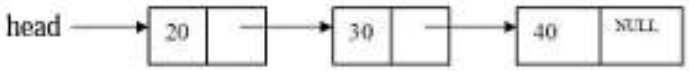
Case 1: Delete a node at beginning of the list



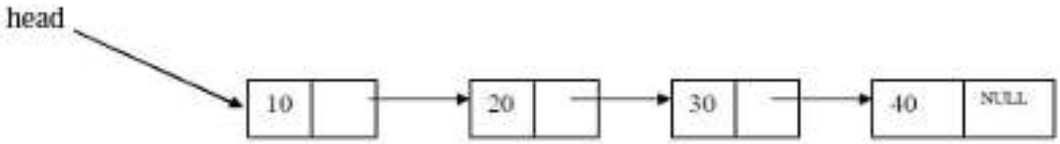
head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);
```

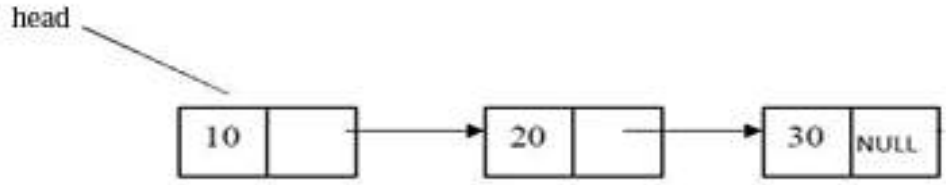


Case 2. Delete a node at end of the list



To delete last node , find the node using following code

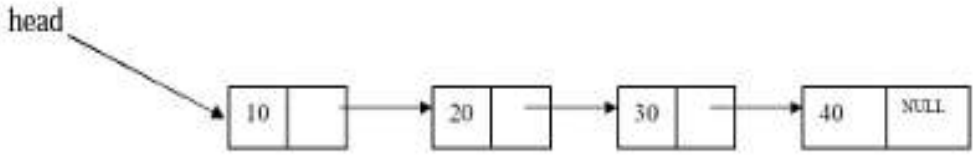
```
struct node<T>*cur,*prev;
cur=prev=head;
while(cur->link!=NULL)
{prev=cur; cur=cur-
        >link;
}
prev->link=NULL;
cout<<"node "<<cur->data<<" Deletion is sucess";
free(cur);
```



Code for deleting a node at end of the list

```
template <class T>
void list<T>::delete_end()
{
struct node<T>*cur,*prev;
        cur=prev=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {cur=prev=head; if(head-
                >link==NULL)
                {
                        cout<<"node "<<cur->data<<" Deletion is
                        sucess"; free(cur);
                        head=NULL;
                }
else
                {while(cur->link!=NULL)
                        {prev=cur; cur=cur-
                                >link;
                        }
                        prev->link=NULL;
                        cout<<"node "<<cur->data<<" Deletion is sucess";
                        free(cur);
                }
        }
}
```

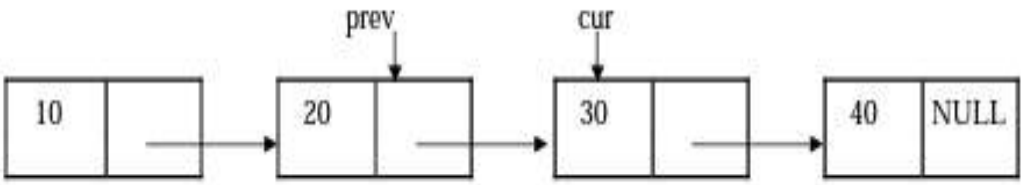
CASE 3. Delete a node at a given position



Delete node at position 3

head is the pointer variable which contains address of the first node. Node to be deleted is node
containing value 30.
Finding node at position 3

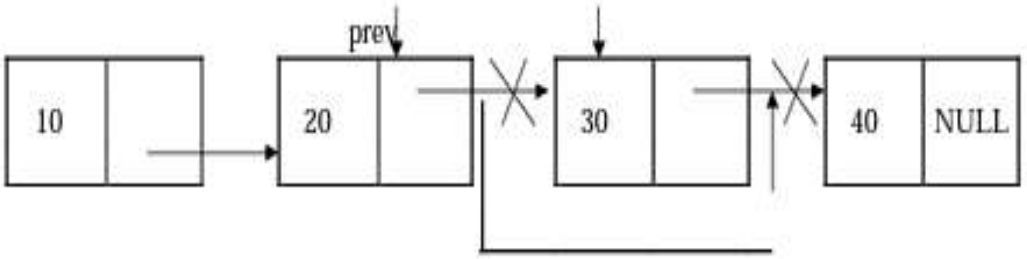
```
c=1;
while(c<pos)
{c++;
        prev=cu
        r;
        cur=cur->link;
}
```



cur is the node to be deleted . before deleting update links

code to update links

```
prev->link=cur->link;
cout<<cur->data <<"is deleted successfully";
delete cur;
```



Traversing the list: Assuming we are given the pointer to the head of the list, how do we get the end of the list.

```
template <class T>
void list<T>:: display()
{
        struct node<T>*t;

        if(head==NULL)
        {
                cout<<"List is Empty\n";
        }
        else
        {t=head;
                while(t!=NUL
                L)
                {cout<<t->data<<"->";
                        t=t->link;
                }
        }
}
```

Example:

Write a program to perform the following operations on singly linked list:
i) Creation ii) Insertion iii) Deletion iv) Traversal v) length of the linked list

```
#include<iostream> #include<cstdio>
#include<cstdlib>

using namespace std;
/*

* Node Declaration

*/

struct node

{

int info;

struct node *next;

}*start;

/*

* Class Declaration

*/

class single_llist

{

public:

node* create_node(int);

void insert_begin();

void insert_pos();
void insert_last();void
delete_pos();void display();
int length_list();single_llist()
{

start = NULL;

}
};

main()

{

int choice, nodes, element, position, i;

single_llist sl;
start = NULL;
while (1)
```

```
{
cout<<endl<<"-------------------------------------"<<endl;
cout<<endl<<"Operations on singly linked list"<<endl;cout<<endl<<"----"<<endl;
cout<<"1.Create node"<<endl;
cout<<"2.Insert Node at beginning"<<endl; cout<<"3.Insert node at
last"<<endl;
cout<<"4.Insert node at position"<<endl; cout<<"5.Delete node at a
position"<<endl; cout<<"6.Display Linked List"<<endl;
cout<<"7.Length of the Linked List"<<endl;cout<<"8.Exit "<<endl;
cout<<"Enter your choice : ";cin>>choice;
switch(choice)
{
   int val;
   case 1:
cout<<"Creating a node: "<<endl; cout<<"Enter a
value:"<<endl; cin>>val;
sl.create_node(val);
cout<<endl;
break;
case 2:
cout<<"Inserting Node at Beginning:"<<endl;sl.insert_begin();
cout<<endl;break;
case 3:
cout<<"Inserting Node at Last: "<<endl;sl.insert_last();
cout<<endl;
break;
case 4:
cout<<"Inserting Node at a given position:"<<endl;sl.insert_pos();
cout<<endl;
break;
case 5:
cout<<"Delete a particular node: "<<endl;sl.delete_pos();
cout<<endl;break;
case 6:
cout<<"Display elements of linked list"<<endl;
sl.display();
cout<<endl;break;
case 7:
cout<<"Length of the linked list = "<<sl.length_list()<<endl;
cout<<endl;
break;
```

```
case 8: cout<<"Exiting..."<<endl;
exit(1);
break; default:
cout<<"Wrong choice"<<endl;

}

}

}

/*

* Creating Node

*/

node *single_llist::create_node(int value)

{

struct node *temp, *s; temp =
new(struct node); if (temp ==
NULL)
{

cout<<"Memory not allocated "<<endl;return 0;
}

else

{

temp->info = value;

temp->next = NULL;
cout<<"A node "<<temp->info<<" is created"<<endl;
return temp;
}

}

/*
* Inserting element in beginning

*/

void single_llist::insert_begin()

{

int value;

cout<<"Enter the value to be inserted: ";cin>>value;
struct node *temp, *p;
temp = create_node(value);
if (start == NULL)
{

start = temp;
```

```
start->next = NULL;
}
else
{
p    =    start;
start = temp;
start->next = p;
}
cout<<"Element Inserted at beginning"<<endl;
}
/*
* Inserting Node at last
*/
void single_llist::insert_last()
{
int value;
cout<<"Enter the value to be inserted: ";
cin>>value;
struct  node *temp, *s;
temp = create_node(value);
s = start;
while (s->next != NULL)
{
s = s->next;
}
temp->next = NULL;
s->next = temp;
cout<<"Element Inserted at last"<<endl;
}
/*
* Insertion of node at a given position
*/
void single_llist::insert_pos()
{
```

```
int value, pos, counter = 0;
cout<<"Enter the value to be inserted: ";
cin>>value;
struct node *temp, *s, *ptr; temp =
create_node(value);
cout<<"Enter the postion at which node to be inserted: ";cin>>pos;
int i;
s = start;
while (s != NULL)
{
s = s->next;
counter++;
}
if (pos == 1)
{
if (start == NULL)
{
start = temp;
start->next = NULL;
}
else
{
ptr = start;
start = temp;
start->next = ptr;
}
}
else if (pos> 1 &&pos<= counter)
{
s = start;
for (i = 1; i<pos; i++)
{
ptr = s;
s = s->next;
}
```

```
ptr->next =temp;
temp->next = s;
}
else
{
cout<<"Positon out of range"<<endl;
}
}
/*
* Delete element at a given position
*/
void single_llist::delete_pos()
{
int pos, i, counter = 0;
if (start == NULL)
{
cout<<"List is empty"<<endl;return;
}
cout<<"Enter the position of value to be deleted: ";cin>>pos;
struct node *s, *ptr;s = start;
if (pos == 1)
{
start = s->next;
}
else
   {
while (s != NULL)
{
s       =       s->next;
counter++;
}
if (pos> 0 &&pos<= counter)
{
s = start;
for (i = 1;i<pos;i++)
```

```
{
ptr = s;
s = s->next;
}
ptr->next = s->next;
}
else
{
cout<<"Position out of range"<<endl;
}
free(s);
cout<<"Element Deleted"<<endl;
}
}
/*
* Display Elements of a link list
*/
int single_llist::length_list()
{
struct node *temp;
int count=0;
if (start == NULL)
{
cout<<"The List is Empty"<<endl;
return 0;
}
temp = start;
while (temp != NULL)
{
count++;
temp = temp->next;
}
return count;
```

```
}
void single_llist::display()
{
struct node *temp;
if (start == NULL)
{
cout<<"The List is Empty"<<endl;return;
}
temp = start;
cout<<"Elements of list are:" <<endl;
while (temp != NULL)
{
cout<<temp->info<<"->";
temp = temp->next;
}
cout<<"NULL"<<endl;
}
```

Input/Output:

```
---------------------------------
Operations on singly linked list
---------------------------------
1.Create node
2.Insert Node at beginning
3.3.Insert node at last
4.Insert node at position
5.Delete node at a position
6.Display Linked List
7.Length of the Linked List
8.Exit
Enter your choice : 1
Creating a node:
 Enter a value:
10
A node 10 is created
Operations on singly linked list
---------------------------------
```

Create node

1.Insert Node at beginning

2..Insert node at last

4.Insert node at position

5.Delete node at a position

6.Display Linked List

7.Length of the Linked List

8.Exit
Enter your choice : 2
Inserting Node at Beginning:
Enter the value to be inserted: 10
A node 10 is created
Element Inserted at beginning

DOUBLY LINKED LIST

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields:

a) One to store data

b) Remaining are self referential pointers which points to previous and next nodes in the list .

Implementation of node using structure

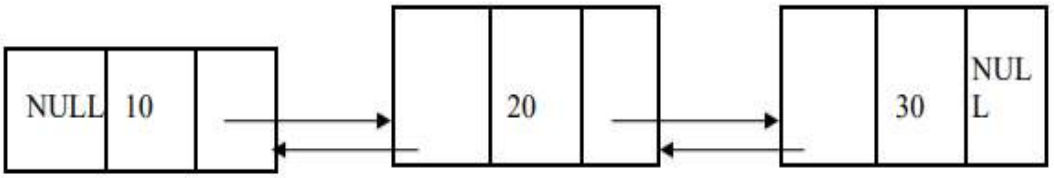
Method -1:

```
struct node
{
        int data;
        struct node *prev;
        struct node * next;
};
```

Implementation of node using class

Method -2:

```
class node
{
public:
        int data;
        node *prev;
        node * next;
};
```



Operations on Doubly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

Doubly linked list ADT:

```
template <class T>
class dlist
{
        int data;
        struct dnode<T>*head;
public:
        dlist()
        {
                head=NULL;
        }
        void display();
        struct dnode<T>*create_dnode(int n);
        void insert_end();
        void insert_front();
        void delete_end();
        void delete_front();
        void dnode_count();

          void Insert_at_pos(int pos);
          void Delete_at_pos(int pos);
};
```

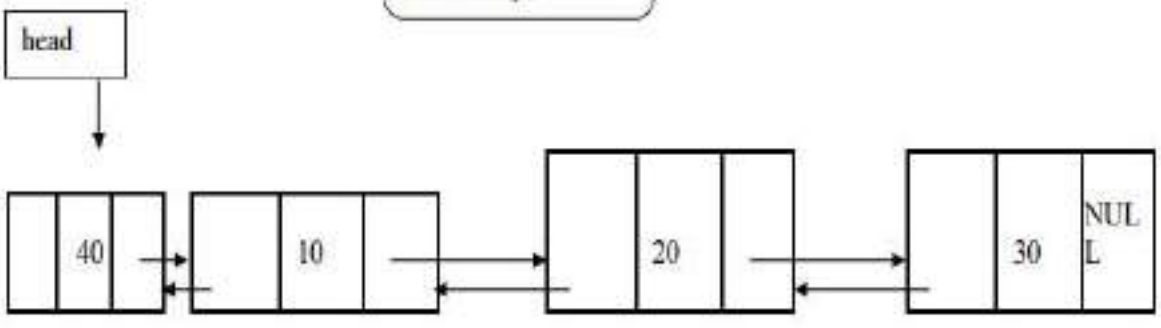
Insertions: To place an elements in the list there are 3 cases

- 1. At the beginning
- 2. End of the list
- 3. At a given position

case 1:Insert at the beginning

head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->next=head;
head->prev=temp;
head=temp;
```



Code for insert front:-

```
template <class T>
void DLL<T>::insert_front()
{
struct dnode <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>data;
        temp=create_dnode(data);
        if(head==NULL)
                head=temp;
        else
        {temp-
                >next=head
                ; head-
                >prev=temp
                ;
                head=temp;
        }
}
```

Code to insert a node at End:-

```
template <class T>

void DLL<T>::insert_end()
{
struct dnode<T> *t,*temp;
int n;
        cout<<"Enter data into dnode:";
        cin>>n;
        temp=create_dnode(n);
        if(head==NULL)
                head=temp;
        else

        {t=head; while(t-
                >next!=NULL)
                        t=t->next;
                 t->next=temp;

                 temp->prev=t;
         }
 }
```

Code to insert a node at a position

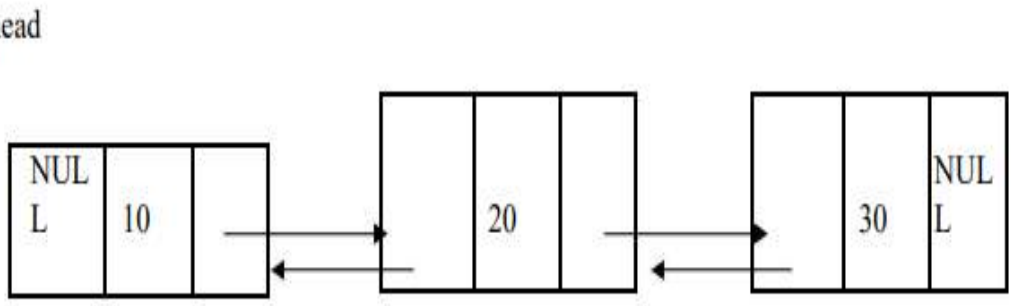
```
template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
        cout<<"Enter data into dnode:";
        cin>>data;
        temp=create_dnode(data);
        display();
        if(head==NULL)
        {//when list is empty
                head=temp;
        }
         else
         {pr=cr=head;
                if(pos==1)
                {//inserting at pos=1
                        temp-
                        >next=head;
                        head=temp;
                }
                else
                {
                        while(count<pos)
                        {count++;
                                pr=cr;
                                cr=cr->next;
                        }
                    pr->next=temp;
                    temp->prev=pr;
                    temp->next=cr;
                    cr->prev=temp;
                }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

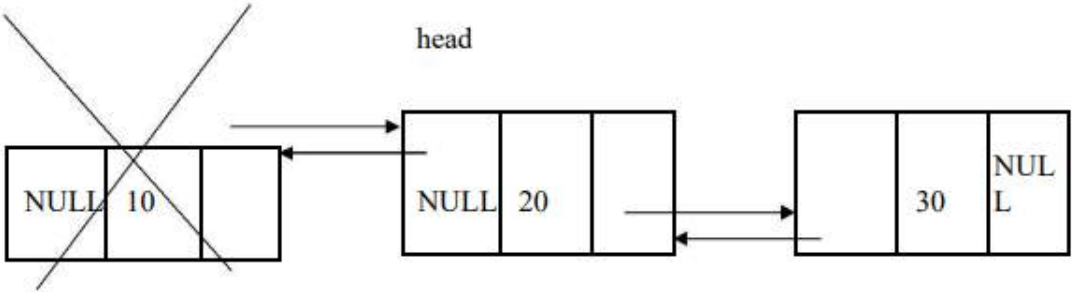
Case 1: Delete a node at beginning of the list



head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
delete(t);
```



code for deleting a node at front

```
template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {t=head;
                head=head->next;
                head->prev=NULL;
                cout<<"dnode "<<t->data<<" Deletion is sucess";
                delete(t);
        }
}
```

A program to perform the following operations on doubly linked list:

i) Creation ii) Insertion iii) Deletion iv) Traversal v) Print the list in reverse

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
/*

* Node Declaration

*/

using namespace std;

struct node
{

int info;

struct node *next;
struct node *prev;
}*start;
```

```
/*
Class Declaration
*/
class double_llist
{
public:
void create_list(int value); void
add_begin(int value);
void add_after(int value, int position);
void delete_element(int value);
void display_dlist();
void reverse(); double_llist()
{
start = NULL;
}
};
/*
* Main: Conatins Menu
*/
int main()
{
int choice, element, position;double_llist dl;
while (1)
{
cout<<endl<<"---------------------------- "<<endl;
cout<<endl<<"Operations on Doubly linked list"<<endl;
cout<<endl<<"---------------------------- "<<endl;
cout<<"1.Create Node"<<endl; cout<<"2.Add at
begining"<<endl; cout<<"3.Add after
position"<<endl; cout<<"4.Delete"<<endl;
cout<<"5.Traverse"<<endl;
cout<<"6.Reverse"<<endl; cout<<"7.Quit"<<endl;
cout<<"Enter your choice : "; cin>>choice;
switch ( choice )
{
case 1:
cout<<"Enter the element: "; cin>>element;
dl.create_list(element);
```

```
cout<<endl;
break;
case 2:
cout<<"Enter the element: ";
cin>>element; dl.add_begin(element);
cout<<endl;
break;
case 3:
cout<<"Enter the element: ";cin>>element;
cout<<"Insert Element after postion: ";cin>>position;
dl.add_after(element, position);cout<<endl;
break;
case 4:
if (start == NULL)
{
cout<<"List empty,nothing to delete"<<endl;break;
}
cout<<"Enter the element for deletion: "; cin>>element;
dl.delete_element(element);
cout<<endl;
break;
case 5:
dl.display_dlist();cout<<endl;
break;
case 6:
if (start == NULL)
{
cout<<"List empty,nothing to reverse"<<endl;
break;
}
dl.reverse();cout<<endl;
break;
case 7:
exit(1);
default:
cout<<"Wrong choice"<<endl;
}
}
return 0;
}
/*
```

```
* Create Double Link List
*/
void double_llist::create_list(int value)
{
   struct node *s, *temp;
   temp = new(struct node);
   temp->info = value;
   temp->next = NULL;
if (start == NULL)
{
temp->prev = NULL;
start = temp;
}
else
{
s = start;
while (s->next != NULL)
s = s->next;
s->next = temp;
temp->prev = s;
}
}
/*
* Insertion at the beginning
*/
void double_llist::add_begin(int value)
{
if (start == NULL)
{
cout<<"First Create the list."<<endl;return;
}
struct node *temp;
temp = new(struct node); temp->prev
= NULL;
temp->info = value;
```

```
 temp->next = start;
start->prev = temp;
start = temp;
cout<<"Element Inserted"<<endl;

}
/*

* Insertion of element at a particular position

*/

void double_llist::add_after(int value, int pos)

{

if (start == NULL)

{

cout<<"First Create the list."<<endl;return;
}

struct node *tmp, *q;int i;
q = start;

for (i = 0;i<pos - 1;i++)

{

q = q->next;

if (q == NULL)

{

cout<<"There are less than "; cout<<pos<<"
elements."<<endl;

return;
}

}

tmp = new(struct node); tmp->info =
value;
if (q->next == NULL)

{

q->next = tmp;

tmp->next = NULL;

tmp->prev = q;


}

else
```

```
{
tmp->next  =  q->next;  tmp->next-
>prev = tmp;q->next = tmp;
tmp->prev = q;
}
cout<<"Element Inserted"<<endl;
}
/*
* Deletion of element from the list
*/
void double_llist::delete_element(int value)
{
struct node *tmp, *q;
/*first element deletion*/
if (start->info == value)
{
tmp = start;
start = start->next;
start->prev = NULL;
cout<<"Element Deleted"<<endl;
free(tmp);
return;
}
q = start;
while (q->next->next != NULL)
{
/*Element deleted in between*/
if (q->next->info == value)
{
tmp = q->next;
q->next  =  tmp->next;  tmp->next-
>prev = q;
cout<<"Element Deleted"<<endl;free(tmp);
return;
}
q = q->next;
```

```
}
/*last element deleted*/
if (q->next->info == value)
{

tmp = q->next;
free(tmp);
q->next = NULL;
cout<<"Element Deleted"<<endl;
return;
}
cout<<"Element "<<value<<" not found"<<endl;
}
/*
* Display elements of Doubly Link List
*/
void double_llist::display_dlist()
{
struct node *q;
if (start == NULL)
{
cout<<"List empty,nothing to display"<<endl;return;
}
q = start;
cout<<"The Doubly Link List is :"<<endl; while (q !=
NULL)
{
cout<<q->info<<" <-> ";
q = q->next;
}
cout<<"NULL"<<endl;
}

/*
* Reverse Doubly Link List
*/
```

```
void double_llist::reverse()
{
struct node *p1, *p2;
p1 = start;
p2 = p1->next;
p1->next = NULL;
p1->prev = p2;
while (p2 != NULL)
{
p2->prev = p2->next;
p2->next = p1;
p1 = p2;
p2 = p2->prev;
}
start = p1;
cout<<"List Reversed"<<endl;
}
```

INPUT/OUTPUT

```
---------------------------------
Operations on Doubly linked list
---------------------------------
1.Create Node
2.Add at begining
3.Add after position
4.Delete
5.Traverse
6.Reverse
7.Quit
Enter your choice : 1
Enter the element: 10
---------------------------------

Operations on Doubly linked list
1.Create Node
---------------------------------
2.Add at begining
3.Add after position
4.Delete
5.Traverse
6.Reverse
7.Quit
```

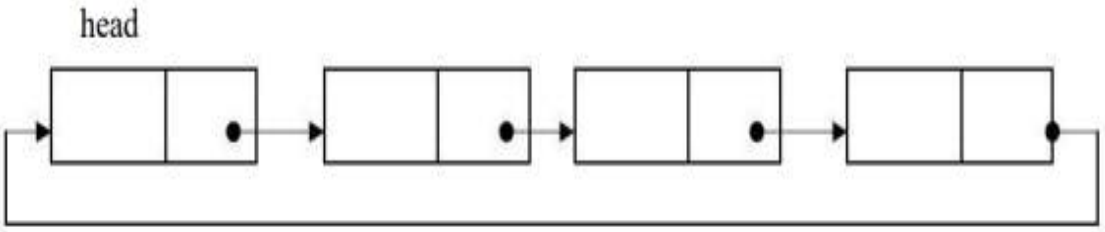
Enter your choice : 1
Enter the element: 20

CIRCULARLY LINKED LIST

A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

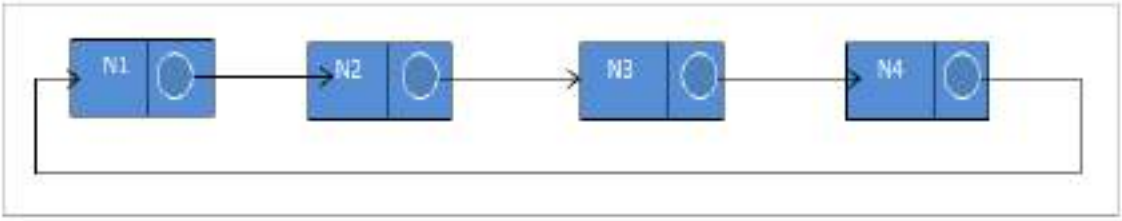
Advantages:

- Any node can be traversed starting from any other node in the list.
- There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
- In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.



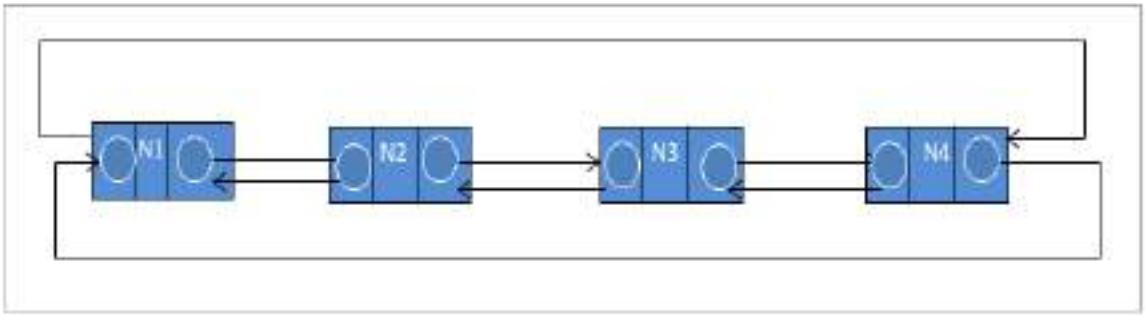
Circular Linked List In C++

The arrangement shown below is for a singly linked list.



A circular linked list can be a singly linked list or a doubly linked list. In a doubly circular linked list, the previous pointer of the first node is connected to the last node while the next pointer of the last node is connected to the first node.

Its representation is shown below.



Declaration

We can declare a node in a circular linked list as any other node as shown below:

```
struct Node
{
   int data;
   struct Node *next;
};
```

In order to implement the circular linked list, we maintain an external pointer "last" that points to the last node in the circular linked list. Hence last->next will point to the first node in the linked list.

By doing this we ensure that when we insert a new node at the beginning or at the end of the list, we need not traverse the entire list. This is because the last points to the last node while last->next points to the first node.

Declaration

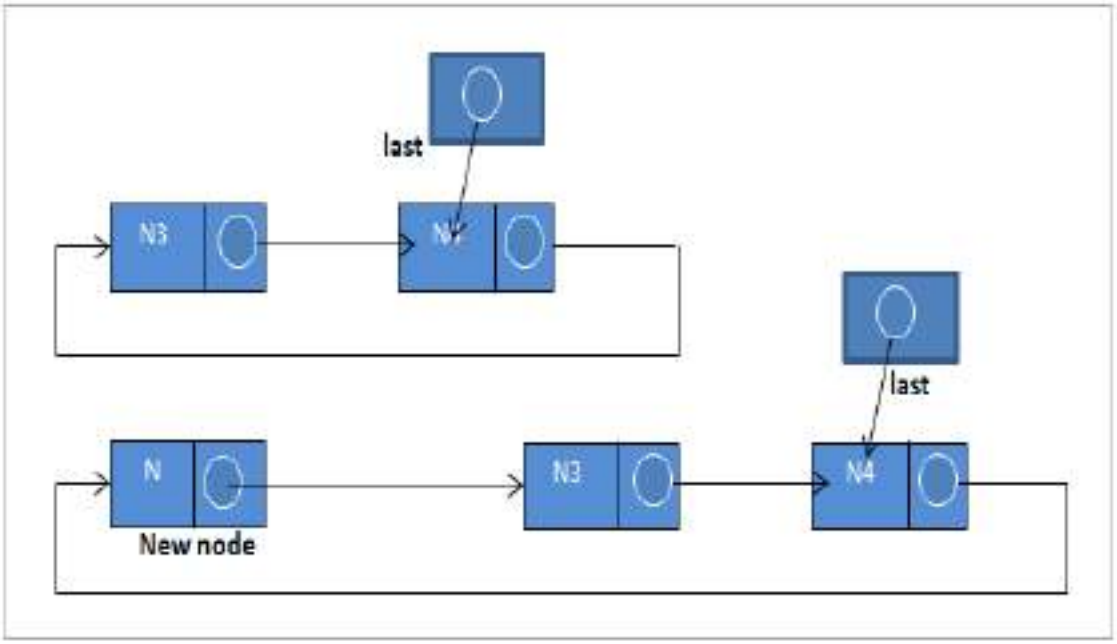
We can declare a node in a circular linked list as any other node as shown below:

```
struct Node
{
   int data;
   struct Node *next;
};
```

In order to implement the circular linked list, we maintain an external pointer "last" that points to the last node in the circular linked list. Hence last->next will point to the first node in the linked list.

By doing this we ensure that when we insert a new node at the beginning or at the end of the list, we need not traverse the entire list. This is because the last points to the last node while last->next points to the first node.

#2) Insert at the beginning of the list

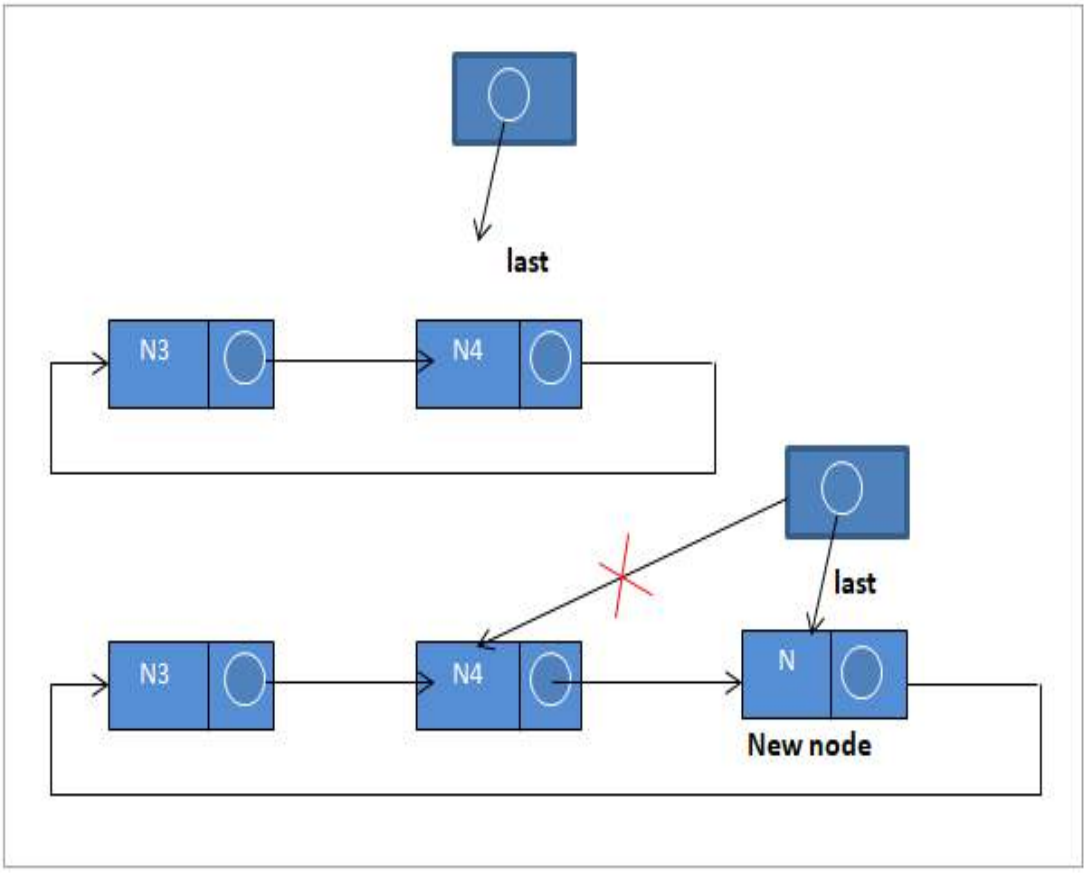


As shown in the above representation, when we add a node at the beginning of the list, the next pointer of the last node points to the new node N thereby making it a first node.

N->next = last->next

Last->next = N

#3) Insert at the end of the list



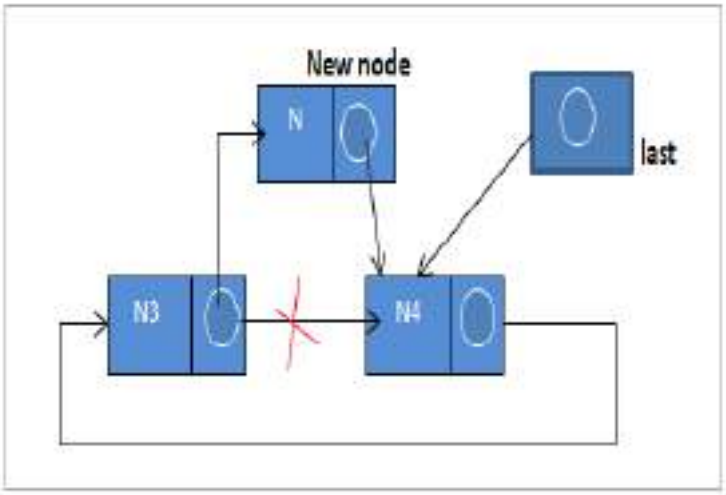
To insert a new node at the end of the list, we follow these steps:

N-> next = last ->next;

last ->next = N

last = N

#4) Insert in between the list



Suppose we need to insert a new node N between N3 and N4, we first need to traverse the list and locate the node after which the new node is to be inserted, in this case, its N3.

After the node is located, we perform the following steps.

N -> next = N3 -> next;

N3 -> next = N

This inserts a new node N after N3.

A program to perform the following operations on circular linkedlist.:

i) Creation ii) Insertion iii) Deletion iv) Traversal v) Count of nodes

```
#include<iostream>  #include<cstdio>
#include<cstdlib>

using namespace std;
/*

 * Node Declaration

 */


struct node

{

   int info;
```

```
        struct node *next;
}*last;


/*
     * Class Declaration
     */


    class circular_llist
    {
        public:
            void create_node(int value);
            void add_begin(int value);
            void add_after(int value, int position); void
            delete_element(int value);
             void display_list();
             int count();
circular_llist()
            {
                last = NULL;
            }
    };

    /*
     * Main :contains menu
     */
    int main()
    {
        int choice, element, position;circular_llist cl;
        while (1)
        {
cout<<endl<<"--------------------------"<<endl;
cout<<endl<<"Circular singly linked list"<<endl;cout<<endl<<"  "<<endl;
cout<<"1.Create Node"<<endl; cout<<"2.Add at
beginning"<<endl;cout<<"3.Add after"<<endl;
cout<<"4.Delete"<<endl; cout<<"5.Display"<<endl;
cout<<"6.Count"<<endl; cout<<"7.Quit"<<endl;
 cout<<"Enter your choice : ";
```

```
cin>>choice;
switch(choice)
        {
        case 1:
        cout<<"Enter the element: "; cin>>element;
        cl.create_node(element); cout<<endl;
        break;
        case 2:
        cout<<"Enter the element: "; cin>>element;
        cl.add_begin(element); cout<<endl;
           break;
           case 3:
        cout<<"Enter the element: ";cin>>element;
        cout<<"Insert element after position: ";
        cin>>position;
        cl.add_after(element, position);cout<<endl;
           break;
           case 4:
           if (last == NULL)
           {
                cout<<"List is empty, nothing to delete"<<endl;
                break;
           }
        cout<<"Enter the element for deletion: ";cin>>element;
         cl.delete_element(element);
        cout<<endl;
        break;
        case 5:
        cl.display_list();
        break;
        case 6:
        cout<<"Count of  Nodes= "<<cl.count()<<endl;
        break;
        case 7:
        exit(1);
           break;default:
        cout<<"Wrong choice"<<endl;
        }
    }
    return 0;
  }
```

```
    /*
     * Create Circular Link List
     */
    void circular_llist::create_node(int value)
    {
        struct node *temp;
        temp = new(struct node);
        temp->info = value;
        if (last == NULL)
        {
            last = temp;
            temp->next = last;
        }
        else
        {
            temp->next = last->next;last->next = temp;
            last = temp;
        }
    }

    /*
     * Insertion of element at beginning
     */
    void circular_llist::add_begin(int value)
    {
        if (last == NULL)
        {
cout<<"First Create the list."<<endl;
return;
        }
        struct node *temp;
        temp = new(struct node);
        temp->info = value;
        temp->next = last->next;
```

```
    last->next = temp;
  }
  /*
   * Insertion of element at a particular place
   */
  void circular_llist::add_after(int value, int pos)
  {
  if (last == NULL)
    {

cout<<"First Create the list."<<endl;
return;
    }
    struct node *temp, *s;
    s = last->next;
    for (int i = 0;i< pos-1;i++)
    {
      s = s->next;
      if (s == last->next)
      {
cout<<"There are less than ";
cout<<pos<<" in the list"<<endl;
        return;
      }
    }
    temp = new(struct node);
    temp->next = s->next;
    temp->info = value;
    s->next = temp;
    /*Element inserted at the end*/
    if (s == last)
    {
      last=temp;
    }
  }
```

```
/*
 * Deletion of element from the list
 */
void circular_llist::delete_element(int value)
{
    struct node *temp, *s;
    s = last->next;
     /* If List has only one element*/
    if (last->next == last && last->info == value)
    {
        temp = last;
         last = NULL;
        free(temp);
         return;
    }
    if (s->info == value)  /*First Element Deletion*/
    {
        temp = s;
        last->next = s->next;
        free(temp);
        return;
    }
    while (s->next != last)
    {
        /*Deletion of Element in between*/
        if (s->next->info == value)
        {
            temp = s->next;
            s->next = temp->next;free(temp);
cout<<"Element "<<value;
cout<<" deleted from the list"<<endl;
return;
        }

        s = s->next;
```

```
        }
        /*Deletion of last element*/
        if (s->next->info == value)
        {
            temp = s->next;
            s->next = last->next;
            free(temp);
            last = s;
            return;
        }
cout<<"Element "<<value<<" not found in the list"<<endl;
    }
    /*
     * Display Circular Link List
     */
    void circular_llist::display_list()
    {
        struct node *s;
        if (last == NULL)
        {
cout<<"List is empty, nothing to display"<<endl;
return;
        }
        s = last->next;
cout<<"Circular Link List: "<<endl;
do
{
cout<<s->info<<"->";
s = s->next;
} while (s != last->next);
        if(s==last->next)
cout<<s->info<<endl;
    }
 int circular_llist::count()
```

```
    {
        int       count=0;
        struct node *s;
        if (last == NULL)
        {
cout<<"List is empty, nothing to display"<<endl;return 0;
        }
        s = last->next;
        while (s != last)
        {
            count++;
            s = s->next;
        }
        count++;
        return count;
    }
```

Input/Output:

```
---------------------------------
Circular singly linked list
---------------------------------
1.Create Node
2.Add at beginning
3.Add after
4.Delete
5.Display6.Count 7.Quit
Enter your choice : 1
Enter the element: 10
Circular singly linked list
---------------------------------
1.Create Node
2.Add at beginning
3.Add after
 4.Delete
5.Display
6.Count
7.Quit
Enter your choice : 1
Enter the element: 20
```

STACK

STACK ADT

- A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack.
- **A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO)-an element inserted last will be removed first.**
- Stack uses a single index or pointer to keep track of the information in the stack.
- The basic operations associated with the stack are: a) push(insert) an item onto the stack. b) pop(remove) an item from the stack.
- **The general terminology associated with the stack is as follows:**
- A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack.
- When an object is removed from the stack, it is said to be popped off the stack.
- Two additional terms almost always used with stacks are **overflow-** which occurs when we try to push more information on a stack that it can hold, and **underflow-**which occurs when we try to pop an item off a stack which is empty.

Pushing items onto the stack: Assume that the array elements begin at 0
(because the array subscript starts from 0) and the maximum elements that can be placed in stack is max.
The stack pointer, top, is considered to be pointing to the top element of the stack. A push operation thus involves adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack. Initially the top is initialized to -1.

Code to push an element on to stack:

```
void stack::push()
{
 if(top==max-1)
 cout<<"Stack Overflow...\n";
else
{
cout<<"Enter an element to be pushed:";
top++;
cin>>data;
stk[top]=data;
 cout<<"Pushed Successfully....\n";
}
 }
```

Code to pop an element from a stack:

```
void stack::pop()
{
if(top==-1)
cout<<“stack underflow”;
else
{
cout<<“the elelment popped is:”<<stk[top];
top--;
}
}
```

Popping an element from stack:

To remove an item, **first extract the data from top position in the stack** and then decrement the stack pointer, top.

Program to demonstrate stack using array implementation:

```
#include <iostream>

using namespace std;
int stack[100], n=100, top=-1;
void push(int val)
{
  if(top>=n-1)

cout<<"Stack Overflow"<<endl;

else {
     top++;
      stack[top]=val;
  }

}

void pop()

{

 if(top<=-1)
cout<<"Stack Underflow"<<endl;
else
{
cout<<"The popped element is "<< stack[top] <<endl;
top--;
  }

}

void display()

{

if(top>=0)

{
cout<<"Stack elements are:";
for(int i=top; i>=0; i--) cout<<stack[i]<<" ";
cout<<endl;
  }
  else

cout<<"Stack is empty";

}

int main()

{
  int ch, val;
```

```
cout<<"1) Push in stack"<<endl; cout<<"2) Pop from stack"<<endl; cout<<"3) Display stack"<<endl; cout<<"4) Exit"<<endl;
  do
  {
cout<<"Enter choice: "<<endl;cin>>ch;
    switch(ch)
     {
    case 1:
cout<<"Enter value to be pushed:"<<endl;cin>>val;
        push(val);break;

      case 2:   pop();
        break;

      case 3: display();
        break;


case 4:

cout<<"Exit"<<endl;break;


      default:

cout<<"Invalid Choice"<<endl;


    }

}while(ch!=4);

return 0;
}
```

Input/Output:

1) Push in stack

2) Pop from stack

3) Display stack

4) Exit

Enter choice:

1

Enter value to be pushed:
10
Enter choice:

1

Enter value to be pushed:
20
Enter choice:

3

Stack elements are:20 10

A program to implement stack (its operations) using Linkedlists

```
#include <iostream> #include<stdlib.h>

using namespace std;

struct Node {
   int data;

   struct Node *next;

};

struct Node* top = NULL;

void push(int val)

 {
   struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
newnode->data = val;
newnode->next = top;
top = newnode;
}

void pop()

{

if(top==NULL)
cout<<"Stack Underflow"<<endl;
   else
   {

cout<<"The popped element is "<< top->data <<endl;

top = top->next;
   }

}

void display() {

struct Node* ptr;if(top==NULL)
cout<<"stack is empty";
else
{
ptr = top;

cout<<"Stack elements are: ";
```

```
while (ptr != NULL)
{
cout<<ptr->data <<" ";
ptr = ptr->next;
      }
   }
cout<<endl;
}
int main()
{
 int ch, val;
cout<<"1) Push in stack"<<endl; cout<<"2) Pop from stack"<<endl; cout<<"3) Display stack"<<endl; cout<<"4) Exit"<<endl;
   do {
cout<<"Enter choice: "<<endl;cin>>ch;
      switch(ch)
      {
      case 1:
cout<<"Enter value to be pushed:"<<endl;cin>>val;
            push(val);
            break;

         case 2:

          pop();
            break;

         case 3:

         display();
            break;
case 4:
 cout<<"Exit"<<endl;break;

         default:

 cout<<"Invalid Choice"<<endl;

 }while(ch!=4);

 return 0;
 }
```

Input/Output:

1) Push in stack

2) Pop from stack

3) Display stack

4) Exit

Enter choice:
1

Enter value to be pushed:
10
Enter choice:

1

Enter value to be pushed:
20
Enter choice:

3

Stack elements are: 20 10

Applications of Stack:

1. Stacks are used **in conversion of infix to postfix expression**.
2. Stacks are also used in **evaluation of postfix expression.**
3. Stacks are used to **implement recursive procedures.**
4. Stacks are used in **compilers.**
5. **Reverse String**

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are

1. Infix Notation

2. Prefix (Polish) Notation

3. Postfix (Reverse-Polish) Notation

INFIX TO POSTFIX CONVERSION USING STACKS

Basic:-

Infix Expression: The operator is in between the two operands

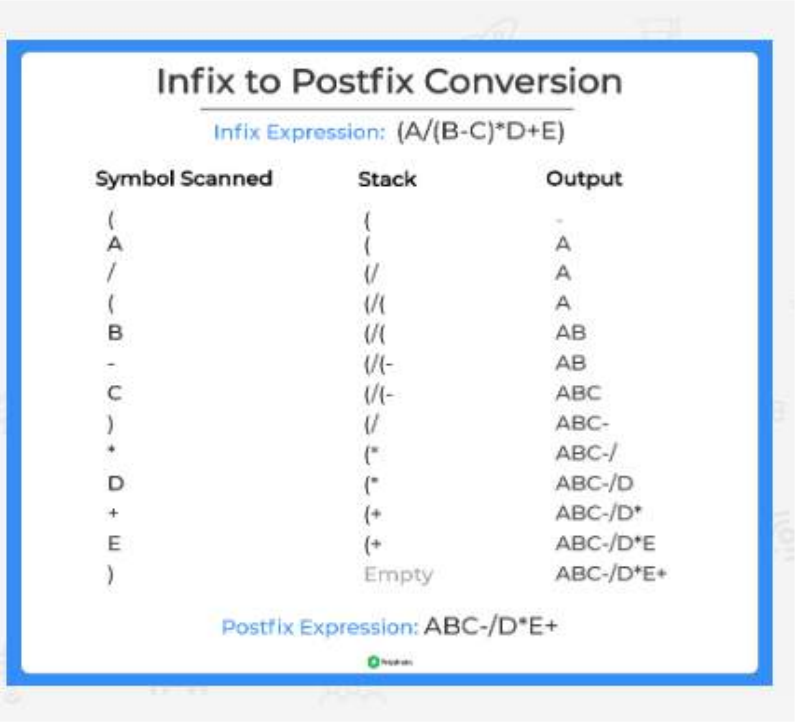
Example: A + B is known as infix expression.

Postfix Expression: The operator is after the two operands

Example: BD + is known as postfix expression.

Steps needed for infix to postfix conversion using stack in C++:-

1. First Start scanning the expression from left to right
2. If the scanned character is an operand, output it, i.e. print it
3. Else
    - If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'(‘), then push operator in the stack
    - Else, Pop all the operators, that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Now, we should repeat steps 2 – 6 until the whole infix i.e. whole characters are scanned.
7. Print output
8. Do the pop and output (print) until the stack is not empty



Benefits of Postfix expression over infix expression

- In postfix any formula can be expressed without parenthesis.
- It is very useful for evaluating formulas on computers with stacks.
- Infix operators have precedence

Example:A program to implement Infix to Postfix Conversion using Stacks

```
#include<iostream>

#include<string.h>

 #include<ctype.h>

using namespace std;

const int MAX = 50 ;

class infix
{

private :

char target[MAX], stack[MAX] ;

char *s, *t ;
int top ;
public :
infix( ) ;

void setexpr( char *str ) ;

void push ( char c ) ;

char pop( ) ;
void convert( ) ;

int priority ( char c ) ;

void show( ) ;
} ;

infix :: infix( )

{

top = -1 ;

strcpy( target, "" ) ;

strcpy( stack, "" ) ;

t = target ;
s = "" ;

}

void infix :: setexpr ( char *str )
{

s = str ;

}

void infix :: push ( char c )

{
```

```
if ( top == MAX )
cout<< "\nStack is full\n" ;
else
{
top++ ;
 stack[top] = c ;
}
}
char infix :: pop( )
{
if ( top == -1 )
{
cout<< "\nStack is empty\n" ;return -1 ;
}
else
{
char item = stack[top] ;
top-- ;
return item ;
}
}
void infix :: convert( )
{
while ( *s )
{
if ( *s == ' ' || *s == '\t' )
{
s++ ;
continue ;
}
if ( isdigit ( *s ) || isalpha ( *s ) )
{
while ( isdigit ( *s ) || isalpha ( *s ) )
{
*t = *s ;s++ ;
```

```
t++ ;
}
}
if ( *s == '(' )
{
push ( *s ) ;
s++ ;
}
char opr ;
if ( *s == '*' || *s == '+' || *s == '/' || *s == '%' || *s == '-' || *s == '$' )
{
if ( top != -1 )
{
opr = pop( ) ;
while ( priority ( opr ) >= priority ( *s ) )
{
*t = opr ;t++ ;
opr = pop( ) ;
}
push ( opr ) ;
push ( *s ) ;
}
else
push ( *s ) ;
s++ ;
}
if ( *s == ')' )
{
opr = pop( ) ;
while ( (opr ) != '(' )
{
*t = opr ;
t++ ;
opr = pop( ) ;
```

```
}
s++ ;
}

}

while ( top != -1 )

{

char opr = pop( ) ;

*t = opr ;

t++ ;
}

*t = '\0' ;

}

int infix :: priority ( char c )
{

if ( c == '$' )

return 3 ;
if ( c == '*' || c == '/' || c == '%' )
return 2 ;
else

{

if ( c == '+' || c == '-' )

return 1 ;
else
return 0 ;
}

}

void infix :: show( )

{

cout<<target ;

}

int main( )

{

char expr[MAX] ;
infix q ;
cout<< "\nEnter an expression in infix form: " ;
cin.getline ( expr, MAX ) ;
q.setexpr ( expr ) ;
q.convert( ) ;
```

```
cout<< "\nThe postfix expression is: " ;q.show( ) ;
return 0;
}
```

Input/Output:

Enter an expression in infix form: a+b*c

The postfix expression is: abc*+

EVALUATION OF POSTFIX EXPRESSION

1. Start reading the expression from left to right.
2. If the element is an operand then, push it in the stack.
3. If the element is an operator, then pop two elements from the stack and use the operator on them.
4. Push the result of the operation back into the stack after calculation.
5. Keep repeating the above steps until the end of the expression is reached.
6. The final result will be now left in the stack, display the same.

Program to demonstrate evaluation of postfix:

```
#include<iostream>
#include<stack>
#include<math.h>
using namespace std;
// The function calculate_Postfix returns the final answer of the expression after calculation
int calculate_Postfix(string post_exp)
{
stack <int> stack;
int len = post_exp.length();
// loop to iterate through the expression
for (int i = 0; i < len; i++)
{
// if the character is an operand we push it in the stack
// we have considered single digits only here
if ( post_exp[i] >= '0' && post_exp[i] <= '9')
{
stack.push( post_exp[i] - '0');
```

```
}
// if the character is an operator we enter else block
else
{
// we pop the top two elements from the stack and save them in two integers
int a = stack.top();
stack.pop();
int b = stack.top();
stack.pop();
//performing the operation on the operands
switch (post_exp[i])
{
case '+': // addition
stack.push(b + a);
break;
case '-': // subtraction
stack.push(b - a);
break;
case '*': // multiplication
stack.push(b * a);
break;
case '/': // division
stack.push(b / a);
break;
case '^': // exponent
stack.push(pow(b,a));
break;
}
}
}
//returning the calculated result
return stack.top();
```

```
}
//main function/ driver function
int main()
{
//we save the postfix expression to calculate in postfix_expression string
string postfix_expression = "59+33^4*6/-";
cout<<"The answer after calculating the postfix expression is : ";
cout<<calculate_Postfix(postfix_expression);
return 0;
}
```

Output

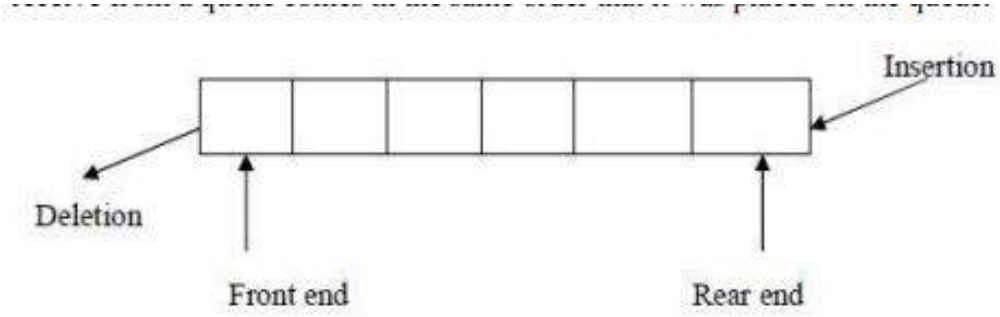
The answer after calculating the postfix expression is: -4

QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end.

The key difference when compared stacks is that in a queue the information stored is processed First-in First-out or **FIFO-an element inserted first, will be removed first.**

In other words the information receive from a queue comes in the same order that it was placed on the queue.



Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to define an array Queue, and two additional variables front and rear.

The rules for manipulating these variables are simple:

- **Each time information is added to the queue, increment rear.**
- **Each time information is taken from the queue, increment front.**

- **Whenever front >rear or front=rear=-1 the queue is empty.**

Array implementation of a Queue do have drawbacks:

- The maximum queue size has to be set at compile time, rather than at run time.
- Space can be wasted, if we do not use the full capacity of the array.
- **Operations on Queue:**
- A queue have two basic operations: a)adding new item to the queue
- b) removing items from queue.
- **The operation of adding new item on the queue occurs only at one end of the queue called the rear or back.**
- **The operation of removing items of the queue occurs at the other end called the front**.
- For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize.
- **The variable front will hold the index of the item that is considered the front of the queue**, **while the rear variable will hold the index of the last item in the queue.**
- **Assume that initially the front and rear variables are initialized to -1.** Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

-

Applications of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Program to implement Queue using Array:

```
#include<stdlib.h> #include<iostream>

 using namespace std;

#define max 5

 template <class T>

class queue
{

private:

T q[max],item;

int front,rear;
public: queue();
void insert_q();
void delete_q();
void display_q();
};

template <class T>
queue<T>::queue()
{

front=rear=-1;

}

//code to insert an item into queue;

template <class T>
void queue<T> ::insert_q()

{

if(rear>=max-1)

cout<<"queue Overflow...\n";

else
{

if(front>rear)

front=rear=-1;
else

{

if(front==-1)

front=0;

rear++;
cout<<"Enter an item to be inserted:";cin>>item;
q[rear]=item;
```

```
cout<<"inserted Sucesfully..into queue..\n";
}
}
}
template <class T>
void queue<T>::delete_q()
{
if(front==-1||front>rear)
{
front=rear=-1;
cout<<"queue is Empty....\n";
}
else
{
item=q[front];
front++;
cout<<item<<" is deleted Sucesfully... \n";
}
}
template <class T>
void queue<T>::display_q()
{
if(front==-1||front>rear)

{
front=rear=-1;
cout<<"queue is Empty....\n";
}
else
{
for(int i=front;i<=rear;i++)
cout<<"|"<<q[i]<<"|<--";
}
```

```
}

int main()

{

int choice; queue<int>
q;while(1)
{

cout<<"\n\n*****Menu for QUEUE operations*****\n\n";
cout<<"1.INSERT\n2.DELETE\n3.DISPLAY\n4.EXIT\n";
cout<<"Enter Choice:";cin>>choice;
switch(choice)
{

case 1: q.insert_q();
break;
case 2: q.delete_q();
break;
case 3:
 cout<<"Elements in the queue are ..... \n";

q.display_q();break;
case 4:
exit(0);
default: cout<<"Invalid choice...Try again...\n";

}

}

return 0;

}
```

Input/Output:

*****Menu for QUEUE operations*****

1.INSERT
2.DELETE

3.DISPLAY

4.EXIT

Enter Choice:3

Elements in the queue are....

queue is Empty....

*****Menu for QUEUE operations*****1.INSERT
2.DELETE

3.DISPLAY

4.EXIT

Enter Choice:1

Enter an item to be inserted:10

A program that implement Queue (its operations) using LinkedLists

```
#include<stdlib.h>
#include<iostream>

using namespace std; template <class
T>

class node
{

public:

T data; node<T>*next;
};

template <class T>

class queue
{

private:

T item;

friend class node<T>;node<T>
*front,*rear;

public:

queue();
void insert_q();
void delete_q();
void display_q();
};

template <class T>
queue<T>::queue()
{

front=rear=NULL;

}

//code to push an item into queue;

template <class T>
void queue<T>::insert_q()

{

node<T>*p;

cout<<"Enter an element to be inserted:";
```

```
cin>>item;
p=new node<T>;
p->data=item;
p->next=NULL;
if(front==NULL)
{

rear=front=p;

}

else

{

rear->next=p;

rear=p;
}

cout<<"\nInserted into Queue Succesfully . \n";

}

//code to delete an element

template <class T>
void queue<T>::delete_q()

{

node<T>*t; if(front==NULL)
cout<<"\nqueue is Underflow";
else
{

item=front->data;

t=front;
front=front->next;

cout<<"\n"<<item<<" is deleted Sucesfully from queue....\n";

}

delete(t);

}

//code to display elements in queuetemplate <class T>
void queue<T>::display_q()

{

node<T>*t;
if(front==NULL)
cout<<"\nqueue Under Flow";
else
{
```

```
cout<<"\nElements in the queue are ... \n";

t=front;

while(t!=NULL)
{

cout<<"|"<<t->data<<"|<-";t=t->next;
}

}

}

int main()

{

int choice;

queue<int>q1;

while(1)
{

cout<<"\n\n***Menu for Queue operations***\n\n";
cout<<"1.Insert\n2.Delete\n3.DISPLAY\n4.EXIT\n";        cout<<"Enter
Choice:";
cin>>choice; switch(choice)
{

case 1:
q1.insert_q();
break;
case 2: q1.delete_q();
break;
case 3: q1.display_q();
break;
case 4:
exit(0);

default:cout<<"Invalid choice...Try again...\n";

}

}

return 0;

}
```

Input/Output:

```
***Menu for Queue operations***
1.Insert
2.Delete
3.DISPLAY
4.EXIT
```

Enter Choice:1

Enter an element to be inserted:10

Inserted into Queue Succesfully....

Menu for Queue operations

1.Insert
2.Delete 3.DISPLAY
4.EXIT

Enter Choice:1

Enter an element to be inserted:20 Inserted into Queue Sucesfully....

Menu for Queue operations

1.Insert
2.Delete 3.DISPLAY
4.EXIT

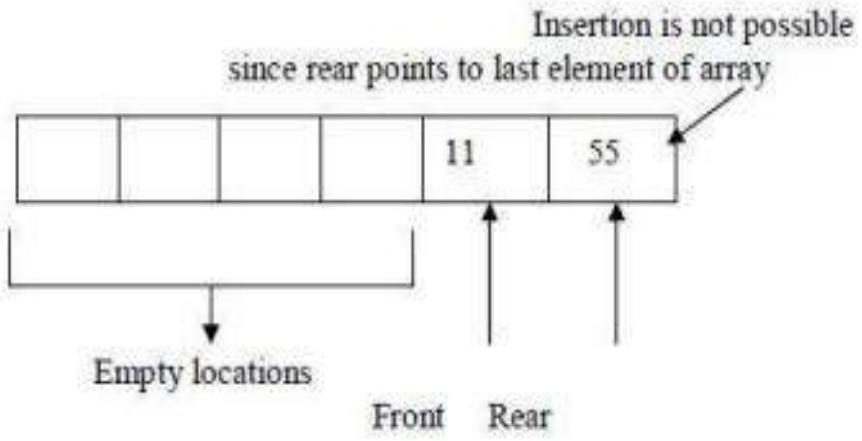
Enter Choice:3

Elements in the queue are....

|10|<-|20|<-

CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.



When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted.

Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as Circular Queues.

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1].

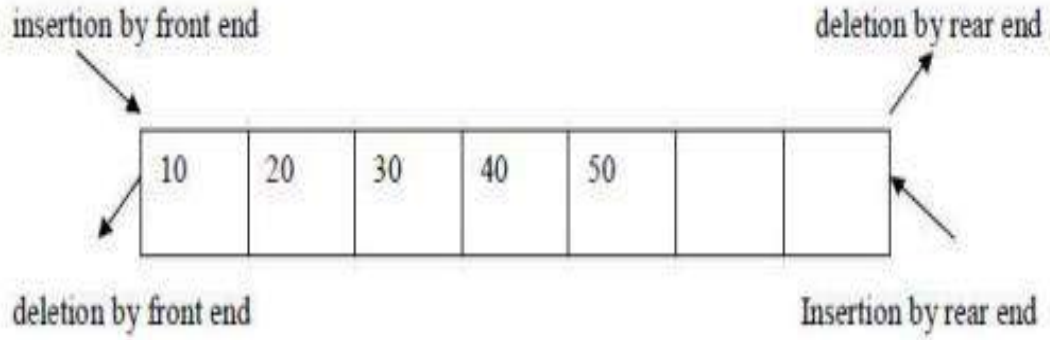
Circular Queue empty or underflow condition is

```
if(front==-1)
        cout<<"Queue is empty";
```

Circular Queue Full or overflow condition is

```
if(front==(rear+1)%max)
{
    cout<<"Circular Queue is full\n";
}
```

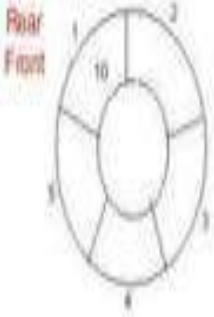
Normally insertion of elements is done at rear end and delete the elements from front end. For example elements 10,20,30 are inserted at rear end. To insert any element from front end then first shift all the elements to the right.



Example: Consider the following circular queue with N = 5.

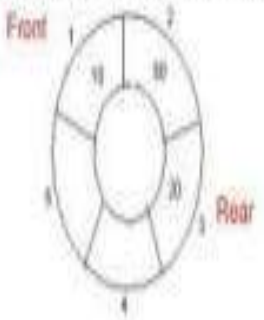
1. Initially, Rear = 0, Front = 0.

2. Insert 10, Rear = 1, Front = 1.

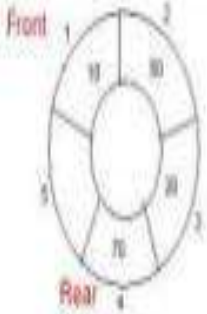


3. Insert 50, Rear = 2, Front = 1.

4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.

Insertion into a Circular Queue:

```
Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)
Step 1: If Rear = maxSize-1 then
                Rear = 0
        else
                Rear=Rear+1
Step 2: If Front = Rear then
        print "Queue Overflow"
        Return
Step 3: Q[Rear] = item

Step 4: If Front = 0 then
        Front = 1
Step 5: Return
```

Deletion from Circular Queue:

```
Algorithm CQueueDeletion(Q,maxSize,Front,Rear,item)
Step 1: If Front = 0 then
        print "Queue Underflow"
        Return
Step 2: K=Q[Front]
Step 3: If Front = Rear then
        begin
                Front = -1
                Rear = -1
        end
        else
        If Front = maxSize-1 then
                Front = 0
        else
                Front = Front + 1
Step 4: Return K
```

DS Question Bank - UNIT – II

Explain Representation of Stack using Arrays ? (L-2)

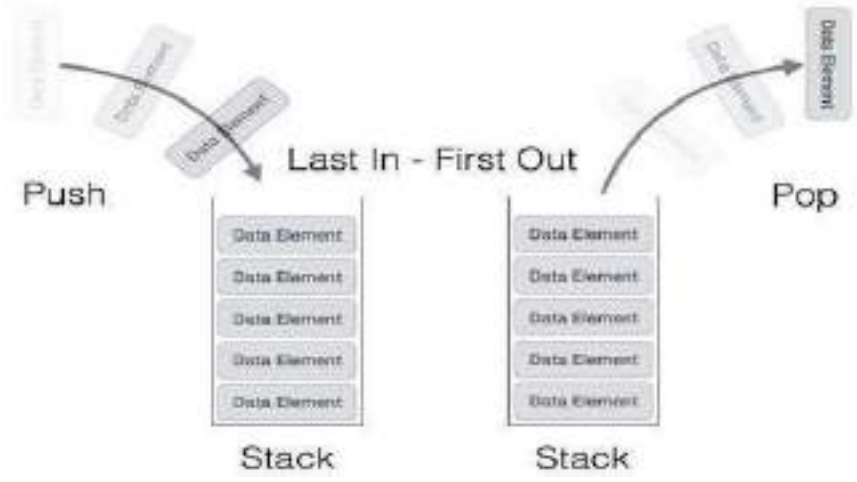
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.

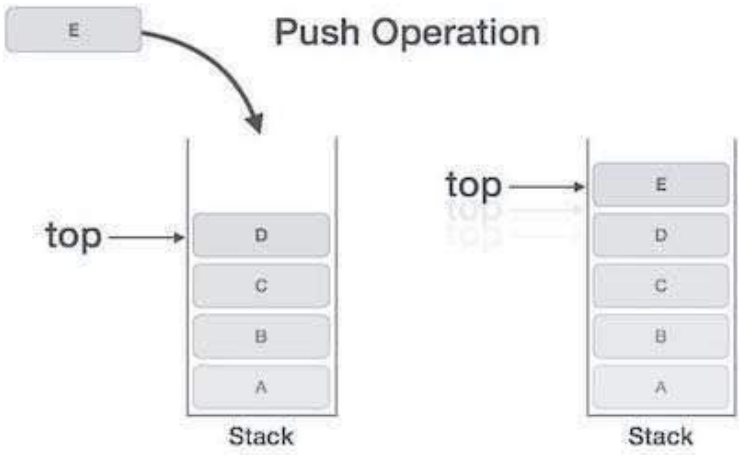
- **pop()** − Removing (accessing) an element from the stack.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

PUSH Operation code:

```
void push(int data){
        if(!isFull()) {
                top = top + 1; stack[top] = data;
        }
```

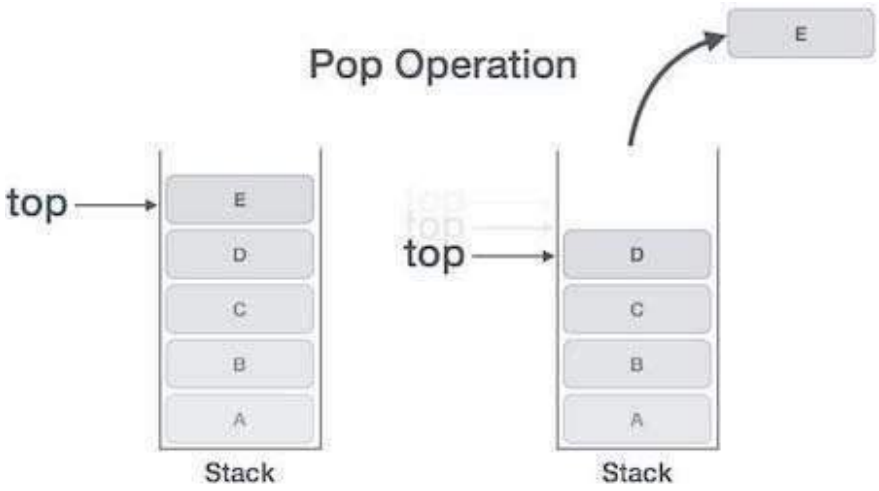
```
        else {
            cout<<"Could not insert data, Stack is full.\n";
        }
    }
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses data element at which **top** is
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



POP Operation code:

```
int pop(int data){
```

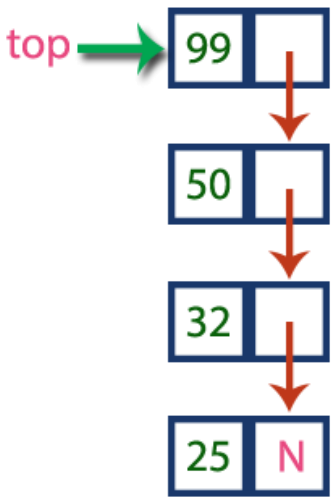
```
    If(!isempty()){
        Data=stack[top];
        Top=top-1;
        Return data;
    }
    Else{
      Cout<<"stack empty";
    }
}
```

1. Explain Linked list representation of stack ? (L-2)

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set newNode → next = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set newNode → next = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = top → next'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display 'temp → data --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (temp → next != **NULL**).
- **Step 5 -** Finally! Display 'temp → data ---> **NULL**'.

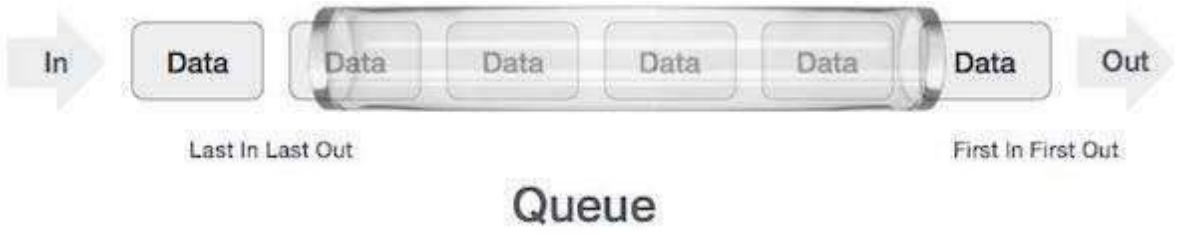
2. Explain the operations of Queue with Example ? (L-2)

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

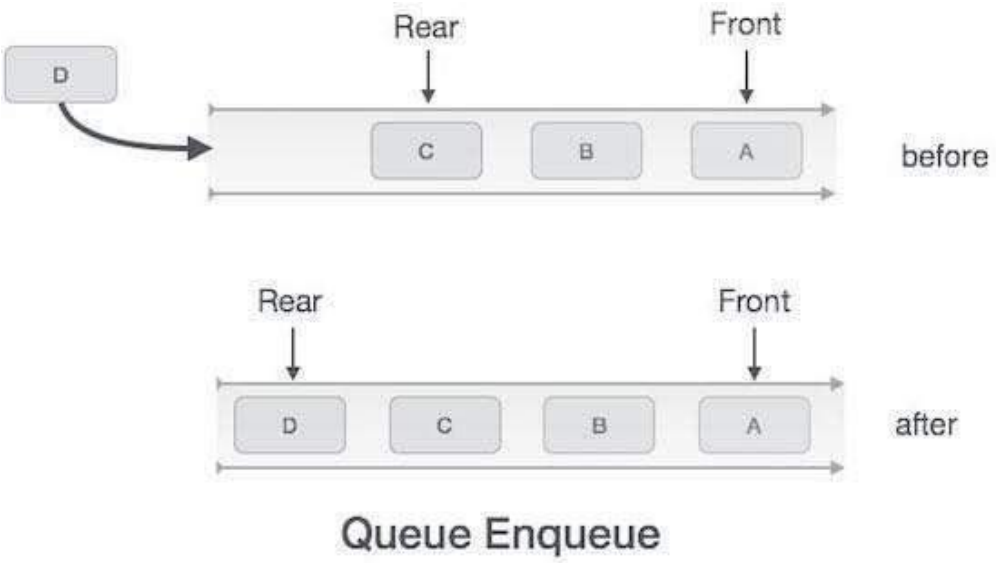
- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

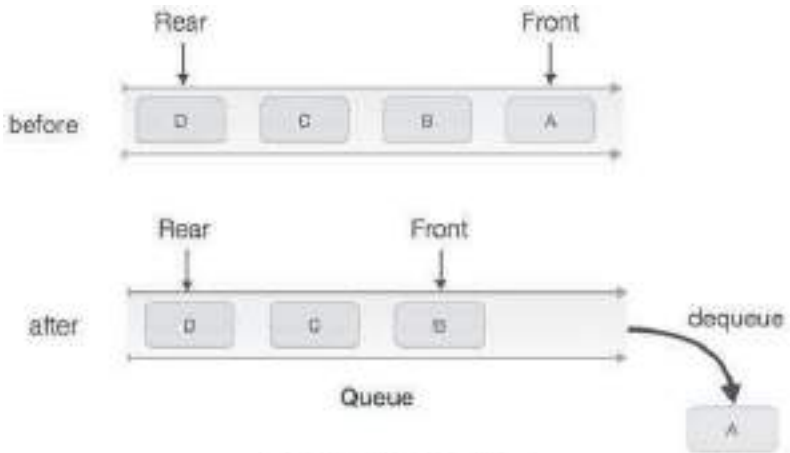
Algorithm for enqueue Operation

```
Procedure enqueue(data)
    If queue is full
        Return
    Rear ← rear+1
    Queue[rear] ← data
End procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue Operation

```
Procedure dequeue
        If queue isempty
                Return
        Data=queue[front]
        Front  front+1
        Return true
End procedure
```

3. Write a program to Queue implementation using two stacks. (L-4)

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
        stack<int> s1, s2;
        void enQueue(int x)
        {
                // Move all elements from s1 to s2
                while (!s1.empty()) {
                        s2.push(s1.top());
                        s1.pop();
         }

                // Push item into s1
                s1.push(x);

                // Push everything back to s1
                while (!s2.empty()) {
```

```
            s1.push(s2.top());
            s2.pop();
        }
        }

        // Dequeue an item from the queue
        int deQueue()
        {
        // if first stack is empty
        if (s1.empty()) {
            cout << "Q is Empty";
            exit(0);
        }

        // Return top of s1
        int x = s1.top();
        s1.pop();
        return x;
        }
};

// Driver code
int main()
{
        Queue q;
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);
```

```
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';

        return 0;
}
```

4. Write a program to Check if a queue can be sorted into another queue using a stack (L-4)

```
#include <bits/stdc++.h>
using namespace std;

// Function to check if given queue element
// can be sorted into another queue using a
// stack.
bool checkSorted(int n, queue<int>& q)
{
    stack<int> st;
    int expected = 1;
    int fnt;
    // while given Queue is not empty.
    while (!q.empty()) {
        fnt = q.front();
    q.pop();
    // if front element is the expected element
    if (fnt == expected)
```

```
                expected++;
        else
        {
                // if stack is empty, push the element
                if (st.empty())
                {
                        st.push(fnt);
                }

                // if top element is less than element which
                // need to be pushed, then return false.
                else if (!st.empty() && st.top() < fnt) {
                        return false;
                }

                // else push into the stack.
                else
                        st.push(fnt);
        }

        // while expected element are coming from
        // stack, pop them out.
        while (!st.empty() && st.top() == expected) {
                st.pop();
                expected++;
        }
}

// if the final expected element value is equal
```

```
        // to initial Queue size and the stack is empty.
        if (expected - 1 == n && st.empty())
                return true;

        return false;
}

// Driven Program
int main()
{
        queue<int> q;
        q.push(5);
        q.push(1);
        q.push(2);
        q.push(3);
        q.push(4);

        int n = q.size();

        (checkSorted(n, q) ? (cout << "Yes") :
                                        (cout << "No"));

        return 0;
}
```

5. Write a program to implement string palindrome using stack. (L-4)

```
    // C++ implementation of the approach
```

```
#include <bits/stdc++.h>
using namespace std;

// Function that returns true
// if string is a palindrome
bool isPalindrome(string s)
{
                int length = s.size();

                // Creating a Stack
                stack<char> st;

                // Finding the mid
                int i, mid = length / 2;

                for (i = 0; i < mid; i++) {
                st.push(s[i]);
                }

                // Checking if the length of the string
                // is odd, if odd then neglect the
                // middle character
                if (length % 2 != 0) {
                i++;
                }

                char ele;
                // While not the end of the string
                while (s[i] != '\0')
```

```
                {
                ele = st.top();
                st.pop();

                // If the characters differ then the
                // given string is not a palindrome
                if (ele != s[i])
                return false;
                i++;
                }

return true;
}

// Driver code
int main()
{
                string s = "madam";

                if (isPalindrome(s)) {
                cout << "Yes";
                }
                else {
                cout << "No";
                }

                return 0;
}
```

6. Show the detailed contents of stack to evaluate the given postfix expression.
{1 2 3 + * 3 2 1 - + *} **(L-4)**

Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

Evaluate the postfix expression: 1 2 3 + * 3 2 1 - + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
1				1	
2				1, 2	
3				1, 2, 3	
+	2	3	5	1, 5	+ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
*	1	5	5	5	* is read, so 1 and 5 are popped from the stack and their product 5, is pushed
3				5, 3	
2				5, 3, 2	
1				5, 3, 2, 1	
-	2	1	1	5, 3, 1	- is read pops 2 and 1 and pushes 2-1 = 1 is pushed
+	3	1	4	5, 4	+ is read pops 3 and 1 and pushes 3 + 1 = 4 is pushed
*	5	4	20	**20**	Finally, * is read pops 5 and 4 and pushes 5 * 4 = 20 is pushed

8. Write a C program to implement multiple stacks using single array. (L-4)

```
/*Implementation of multiple stacks in a single array


#include <iostream>
#include <stdlib >
int s[50],top[50],min[50],max[50];
int ns,size;
void init(void)
{
 int i;
```

```
 for(i=0;i<50;++i)
 {
 s[i]=min[i]=max[i] = 0;
 top[i]=-1;
 }
}
void createstack()
{
 int i ;
 //min and top of 0th stack will be -1
min[0]= -1;
 max[0] = size -1;
 top[0]=-1;

 //min and top of 1,2,3,….th stacks
 for(i=1;i<ns;++i)
 {
 min[i]= min[i-1] + size;
 top[i] = min [i];
}
for(i=1;i<ns;++i)
 {
```

```
 max[i]= min[i+1];
void push(int ele,int k)
{
 if(top[k-1]==max[k-1])
 {
cout<<"Stack no %d is full i.e overflow\n"<<k;
 return;
 }

 ++top[k-1];
 s[top[k-1]] = ele;
}
void pop(int k)
{

 if(top[k-1]==min[k-1])
 {
 cout<<"\nStack no %d is empty i.e underflow\n"<<k;
 return;
}

 cout"%d from stack %d is deleted:\n"<<s[top[k-1]],k;
- -top[k-1];
}
void display(int k)
{
 int j;
 if(top[k-1]==min[k-1])
```

```
 {
 cout<<"\nStack no %d is empty\n"<<k;
 return;
 }
 cout<<"\nStack %d -> "<<k;

 for(j=min[k-1]+1;j<=top[k-1];++j)
 {
 cout<<"%d"<<s[j];
}
}
int main() {
int ele,ch,skn;
 init();
  cout<<"\nEnter the number of Stacks\n";
 cin>>ns;
 size = 50/ns;
 cout<<"\n1.Push\n2.Pop\n3.Display\n4.Exit\n";
 do{
cout<<"\nEnter your choice : \t";
 cin>>ch;
 switch(ch)
 {
 case 1: cout<<"\nEnter the stack no : \t";
cin>>skn;
 cout<<"\nEnter the element : \t";
 cin>>ele;
 push(ele,skn);
```

```
 break;
 case 2 : cout<<"\nEnter the stack no to pop : \t";
 cin>>skn;
 pop(skn);
 break;
 case 3: cout<<"\nEnter the stack no to display : \t";
 cin>>skn;
 display(skn);
 break;
 case 4 : cout<<"\nProgram Terminating";
 break;
 default : cout<<"\nInvalid Option\n";
 }//end switch
 }//end do-while loop
 while(ch!=4);
 return 0;
}
```

9. Convert the infix expression a / b – c + d * e – a * c into postfix expression and trace that postfix expression for given data a = 6, b = 3, c = 1, d = 2, e = 4. (L-3)

Infix expression:6/3-1+2*4-6*1

No of steps	Symbols reading	stack	Postfix expression
Step1	6	Empty	6
Step2	/	/	6
Step3	3	/	63
Step 4	-	-	63/
Step 5	1	-	63/1
Step 6	+	+	63/1-
Step 7	2	+	63/1-2
Step 8	*	+*	63/1-2
Step 9	4	+*	63/1-24
Step 10	-	-	63/1-24*+
Step 11	6	-	63/1-24*+6
Step 12	*	-*	63/1-24*+6
Step 13	1	-*	63/1-24*+61
Step 14		empty	63/1-24*+61*-

```
#include <iostream>
#include <string>
#include <ctype>
const int MAX = 50 ;
class infix
{
private :
```

```
char target[MAX], stack[MAX] ;
char *s, *t ;
int top ;
public :
infix( ) ;
void setexpr ( char *str ) ;
void push ( char c ) ;
char pop( ) ;
void convert( ) ;
int priority ( char c ) ;
void show( ) ;
} ;
infix :: infix( )
{
top = -1 ;
strcpy ( target, "" ) ;
strcpy ( stack, "" ) ;
t = target ;
s = "" ;
}
void infix :: setexpr ( char *str )
}
}
char infix :: pop( )
{
if ( top == -1 )
{
cout << "\nStack is empty\n" ;
```

```
return -1 ;
}
else
{
char item = stack[top] ;
top-- ;
return item ;
}
}
void infix :: convert( )
{
while ( *s )
{
if ( *s == ' ' || *s == '\t' )
{
s++ ;
continue ;
}
if ( isdigit ( *s ) || isalpha ( *s ) )
{
while ( isdigit ( *s ) || isalpha ( *s ) )
{
*t = *s ;
s++ ;
t++ ;
}
}
if ( *s == '(' )
```

```
{
push ( *s ) ;
s++ ;
}
char opr ;
if ( *s == '*' || *s == '+' || *s == '/' || *s == '%' || *s == '-' ||
*s == '$' )
{
if ( top != -1 )
{
opr = pop( ) ;
while ( priority ( opr ) >= priority ( *s ) )
{
*t = opr ;
t++ ;
opr = pop( ) ;
}
push ( opr ) ;
push ( *s ) ;
}
else
push ( *s ) ;
s++ ;
}
if ( *s == ')' )
{
opr = pop( ) ;
while ( ( opr ) != '(' )
{
```

```
*t = opr ;
t++ ;
opr = pop( ) ;
}
s++ ;
}
}
while ( top != -1 )
{
char opr = pop( ) ;
*t = opr ;
t++ ;
}
}
*t = '\0' ;
}
int infix :: priority ( char c )
{
if ( c == '$' )
return 3 ;
if ( c == '*' || c == '/' || c == '%' )
return 2 ;
else
{
if ( c == '+' || c == '-' )
return 1 ;
else
return 0 ;
```

```
}
}
void infix :: show( )
{
cout << target ;
}
void main( )
{
char expr[MAX] ;
infix q ;
cout << "\nEnter an expression in infix form: " ;
cin.getline ( expr, MAX ) ;
q.setexpr ( expr ) ;
q.convert( ) ;
cout << "\nThe postfix expression is: " ;
q.show( ) ;
}
```

10a)list the applications of Stack. (L – 2)

- Evaluation of Arithmetic Expressions:

To evaluate a postfix expression using Stack data structure we can use the following steps:

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Ex: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

- Backtracking:

It is a general algorithm to find solution to some computational problems. Let us consider a simple example of a maze. There are several points from starting to destination. If a random path is chosen, suppose its wrong there must be a way to reach the beginning of a point. With stacks, we remember the point where we reached by pushing that point to stack. In case wrong path, pop the last point from stack and thus return to the last point and continue to find the right path.

- Delimiter Checking:

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the

delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

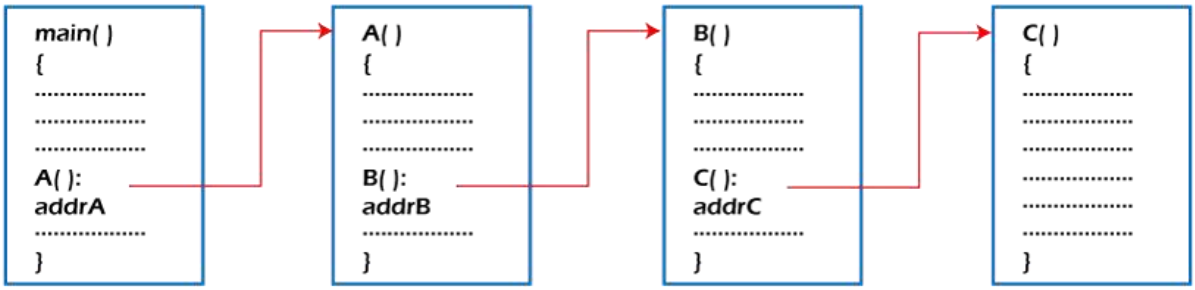
To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

- Memory management:

Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc. The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments. Each program that is running in a computer system has its own memory allocation containing

- Function Call(recursive functions.):

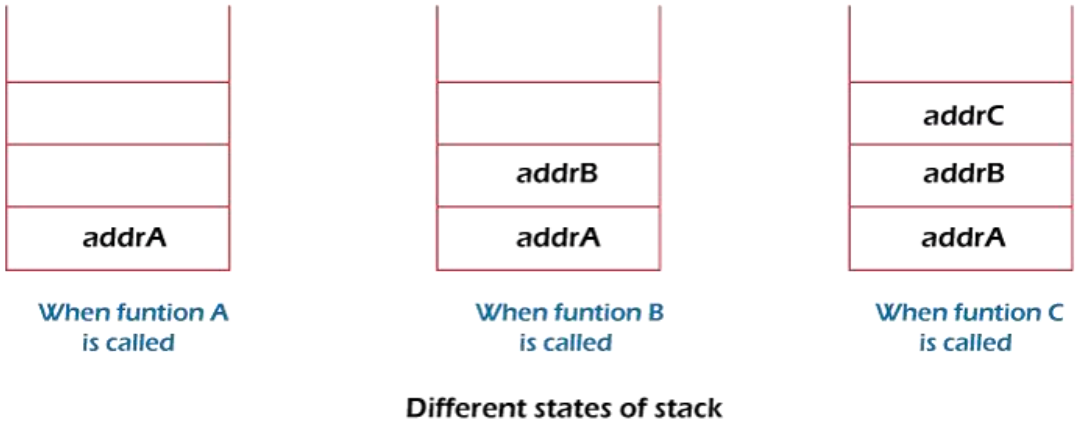
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

- Reversing a String:

  It is one of the simple applications of stack, where characters of string are pushed onto the stack one by one as the string is read from left to right. Once all characters are pushed, they are popped out. Since the last character pushed in is the first to be popped out. Subsequent pop operation results in reversal of strings.

- Expression Conversion or Expression Evaluation:

  Expressions are 3 types:

  Infix Expression:

  An infix expression is an expression in which operators (+, -, *, /) are written between the two operands.

  Ex:

  A+B

  Here we have written **'+'** operator between the operands A and B.

  Postfix Expression:

  In the postfix expression, the operator is written after the operand. It is also known as **Reverse Polish Notation**.

  Ex:

  AB+

  Here we have written **'+'** operator after the operands A and B.

  Prefix Expression:

  In the postfix expression, the operator is written before the operand.

  Ex:

  +AB

  Here we have written **'+'** operator before the operands A and B.

infix to postfix:

Algorithm:

1. STEP 1: Start
2. STEP 2: Initialize the stack.
3. Aurora’s Technological and Research Institute 7
4. Department of CSE Data Structures Lab Manual
5. STEP 3: While (INSTR!= NULL)
6. STEP 4: CH= get the character from INSTR.
7. STEP 5: If(CH= = operand) then
8. append CH int POSTSTR
9. else if(CH = = ‘(‘) then
10. push CH into stack
11. else if(CH = =’)’) then
12. pop the data from the stack and append the data into POSTSTR until we get
13. ‘(‘ from the stack
14. else
15. while(precedence (TOP) >= precedence (CH))
16. pop the data from stack and append the data into POSTSTR.
17. [End of while structure]
18. [End of if structure]
19. STEP 6: Push CH into the stack.
20. STEP 7: [End of second while structure]
21. STEP 8: pop all the data from stack and append data into POSTSTR.
22. STEP 9: Stop

EX:

Infix Expression : A+B*(C^D-E)				
Token	**Action**	**Result**	**Stack**	**Notes**
A	Add **A** to the result	A		
+	Push **+** to stack	A	+	
B	Add **B** to the result	AB	+	
*	Push ***** to stack	AB	* +	* has higher precedence than +
(	Push **(** to stack	AB	(* +	
C	Add **C** to the result	ABC	(* +	
^	Push **^** to stack	ABC	^ (* +	
D	Add **D** to the result	ABCD	^ (* +	
-	Pop **^** from stack and add to result	ABCD^	(* +	- has lower precedence than ^
	Push **-** to stack	ABCD^	- (* +	
E	Add **E** to the result	ABCD^E	- (* +	
)	Pop **-** from stack and add to result	ABCD^E-	(* +	Do process until (is popped from stack
	Pop **(** from stack	ABCD^E-	* +	
	Pop ***** from stack and add to result	ABCD^E-*	+	Given expression is iterated, do Process till stack is not Empty, it will give the final result
	Pop **+** from stack and add to result	**ABCD^E-*+**		
Postfix Expression : ABCD^E-*+				

10.2 (L-3)

```
#include <iostream>
#include <stack>
using namespace std;
bool isBalancedExp(string exp) {
   stack<char> stk;
   char x;
   for (int i=0; i<exp.length(); i++) {
      if (exp[i]=='('||exp[i]=='['||exp[i]=='{') {
         stk.push(exp[i]);
         continue;
      }
      if (stk.empty())
         return false;
      switch (exp[i]) {
      case ')':
         x = stk.top();
         stk.pop();
         if (x=='{' || x=='[')
            return false;
         break;
      case '}':
         x = stk.top();
         stk.pop();
         if (x=='(' || x=='[')
            return false;
         break;
      case ']':
         x = stk.top();
         stk.pop();
         if (x =='(' || x == '{')
```

```
            return false;
         break;
      }
   }
   return (stk.empty());
}
int main() {
   string expresion = "()[(){()}]";
   if (isBalancedExp(expresion))
      cout << "This is Balanced Expression";
   else
      cout << "This is Not Balanced Expression";
}
```