C++ PROGRAMMING LAB MANNUAL

FOR

3rd SEMESTER

(2023)

(For Private Circulation Only)

Gopal Narayan Singh University

FACULTY OF INFORMATION TECHNOLOGY

JAMUHAR, SASARAM, BIHAR-821305

1.Getting accustomed with C++ Compiler

1. Write a program to print a string-" I love my country"

```
#include <iostream>
using namespace std;
int main()
{
   cout<<"I love my country";
   return 0;
}
```

2. Take two variables and take user input and add them.

```
#include <iostream>
using namespace std;
int main()
{
   float n1,n2,sum,av;
   cout<< "Enter two numbers"<< "\n";
   cin>> n1;
   cin>> n2;
   sum= n1+n2;
   cout<< "The Sum is="<< sum<<"\n";
   return 0;
}
```

3. Write a Program to read two numbers and find their Average.

```
#include <iostream>
using namespace std;
int main()
{
   float n1,n2,sum,av;
   cout<< "Enter two numbers"<< "\n";
   cin>> n1;
   cin>> n2;
   sum= n1+n2;
   av= sum/2;
   cout<< "The Sum is="<< sum << ";" << "The Average is="<< av<<"\n";
   return 0;
}
```

4. Write a program to **c**heck if a number is even or odd.

```
#include <iostream>
using namespace std;
int main() {
int a ;
cin>>a;
if(a%2 == 0)
   cout<<"even";
else
```

```
 cout<<"odd";
return 0;
}
```

5. Write a program to check whether a number is a palindrome
6. Write a program to find Factorial of a number
7. Write a program to print Fibonacci Series till nth term
8. Write a program to sum the series: 1+1/2+1/3+1/4+..+1/n till the nth term
9. Write a program to sum the series: 1+1/2!+1/3!+1/4!+..+1/n! till nth term
10. Write a program to convert the given temperature in Fahrenheit to Celsius using the following formula: C=(F+32)/1.8

1. Write a program using Class to print a Name and Roll number of a student
2. Write a class to calculate the Area and Volume of a Room.
3. Write a program to find the Factorial and Fibonacci series using two class program.
4. Define a class student with the following specification

Private members of class student

admno	integer
sname	20 character
eng. math, science	float
total	float
ctotal()	a function to calculate eng + math + science with float return type.

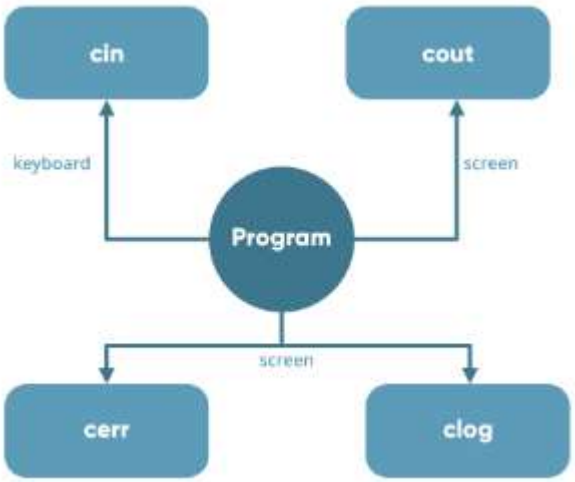
Public member function of class student

Takedata()	Function to accept values for admno, sname, eng, science and invoke ctotal() to calculate total.
Showdata()	Function to display all the data members on the screen.

Basic Input / Output in C++:

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



Header files available in C++ for Input/Output operations are:

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip**: iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.
4. **bits/stdc++:** This header file includes every standard library. In programming contests, using this file is a good idea, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive.
   In C++ after the header files, we often use '*using namespace std;*'. The reason behind it is that all of the standard library definitions are inside the namespace std. As the library functions are not defined at global scope, so in order to use them we use *namespace std*. So, that we don't need to write STD:: at every line (eg. STD::cout etc.).

The two instances **cout in C++** and **cin in C++** of iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.
This article mainly discusses the objects defined in the header file *iostream* like the cin and cout.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).
- **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard. The extraction operator(>>) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.
- **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream that is used to output the errors. This is also an instance of the iostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
- **The main difference** between cerr and cout comes when you would like to redirect output using "cout" that gets redirected to file if you use "cerr" the error doesn't get stored in file.(This is what un-buffered means ..It cant store the message)

```
#include <iostream>

using namespace std;

int main()
{
   cerr << "An error occurred";
   return 0;
}
```

- **buffered standard error stream (clog)**: This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.

```
#include <iostream>

using namespace std;

int main()
{
   clog << "An error occurred";

   return 0;
}
```

The scope resolution operator (: :) is used for several reasons. For example: If the global variable name is same as local variable name, the scope resolution operator will be used to call the global variable. It is also used to define a function outside the class and used to access the static variables of class.

Here an example of scope resolution operator in C++ language-

```
#include <iostream>
using namespace std;
char a = 'm';
static int b = 50;
int main() {
  char a = 's';
  cout << "The static variable : "<<::b;
  cout << "\nThe local variable : " << a;
  cout << "\nThe global variable : " << ::a;
  return 0;
}
```

The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them. The scope resolution operator can be used as both unary and binary. You can use the unary scope operator if a namespace scope or global scope name is hidden by a particular declaration of an equivalent name during a block or class. For example, if you have a global variable of name “a” and a local variable of name “a” , to access global “a” , we'll need to use the scope resolution operator.

```
#include <iostream>
using namespace std;

int a = 0;
int main(void) {
  int a = 0;
  ::a = 1;  // set global a to 1
  a = 2;    // set local a to 2
  cout << ::a << ", " << a;
  return 0;
}
```

We can also use the scope resolution operator to use class names or class member names. If a class member name is hidden, you can use it by prefixing it with its class name and the class scope operator. For example,

```
#include <iostream>
using namespace std;
class X {
  public:
  static int count;
};
int X::count = 10;  // define static data member

int main () {
  int X = 0;   // hides class type X
  cout << X::count << endl;   // use static member of class X
}
```

IMPORTANT –"this" pointer

If some code has some members say 'x', and we want to use another function that takes an argument with the same name 'x', then in that function, if we use 'x', it will hide the member variable, and local variable will be used. Let us check this in one code.

```
#include <iostream>
using namespace std;
class MyClass {
  private:
    int x;
  public:
    MyClass(int y) {
      x = y;
    }
  void myFunction(int x) {
    cout << "Value of x is: " << x;
  }
};
int main() {
  MyClass ob1(10);
  ob1.myFunction(40);
}
```

Value of x is: 40

To access the x member of the class, we have to use the 'this' pointer. 'this' is a special type of pointer that points to the current object. Let us see how **'this' pointer** helps to do this task.

```
#include <iostream>
using namespace std;
class MyClass {
  private:
    int x;
  public:
    MyClass(int y) {
      x = y;
    }
  void myFunction(int x) {
    cout << "Value of x is: " << this->x;
  }
};
int main() {
  MyClass ob1 (10);
  ob1.myFunction (40);
}
```

Value of x is: 10

Problems on Pointers-

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value.

Return by reference is very different from Call by reference. Functions behaves a very important **role** when variable or pointers are returned as reference.

1. Write a program to find factorial of a number using the following definition-
   a) int * fun1(int *);
2. Write a program to swap two numbers using a third variable-**Call by reference**-
   void swap(int &x, int &y)
3. Write a program to swap two numbers without a third variable-
   void swap(int *x,int *y)
   a) Address of the two numbers to be swapped in main function.
   b) Address of the same two numbers in the Function
   c) Output after swapping
4. Write a program to show **"Return by Reference"** by finding the maximum of two numbers
   int * max(int *x,int *y)
   a) Address of the two numbers to be swapped in main function.
   b) Address of the same two numbers in the Function
   c) Value of the same two numbers in the function.
   d) Print Maximum in main

**********Call **by reference************

```
#include <iostream>
using namespace std;
void swap(int &, int &);

int main()
{
   int a,b;
   cout<<"Input two numbers to Swap "<<"\n";
   cin>> a;
   cin>> b;
   cout<<"a="<<a <<"\n";
   cout<<"b="<<b <<"\n";
   swap(a,b);
   cout<< "After Swapping"<< "\n";
   cout<<"a="<<a <<"\n";
   cout<<"b="<<b <<"\n";
   return 0;
}
```

```
void swap(int &x, int &y)
{
 int temp;
 temp=x;
 x=y;
 y=temp;
}
```

1. Read a number and write a menu-driven program to do the following:
    a) To check whether the number is even or odd.
    b) To find the factorial of a number
    c) Press Exit if there is no more jobs to be done.
2. Write a program to input a 2-D matrix and print the upper and lower triangular elements.
3. Write a program to simulate a Calculator.
    - Add two numbers
    - Delete two numbers
    - Divide two numbers
    - Multiply two numbers
    - Exit

Constructors: Default, Parameterized & Copy

4. Write a program to print the names of students by creating a Student class. If no name is passed while creating an object of the Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating the object of the Student class.

5. Create a class named 'Rectangle' with two data members- length and breadth and a function to calculate the area which is 'length*breadth'. The class has three constructors which are :
   1 - having no parameter - values of both length and breadth are assigned zero.
   2 - having two numbers as parameters - the two numbers are assigned as length and breadth respectively.
   3 - having one number as parameter - both length and breadth are assigned that number.
   Now, create objects of the 'Rectangle' class having none, one and two parameters and print their areas.

6. Suppose you have a Piggie Bank with an initial amount of $50 and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of $50. Now make two constructors of this class as follows:
   1 - without any parameter - no amount will be added to the Piggie Bank
   2 - having a parameter which is the amount that will be added to the Piggie Bank
   Create an object of the 'AddAmount' class and display the final amount in the Piggie Bank

****

Copy Constructors:- A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.

Syntax**:** **ClassName(const ClassName &oldobj)**

Characteristics of Copy Constructor

1. The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
2. Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
{
    id=t.id;
```

}

3. The process of initializing members of an object through a copy constructor is known as *copy initialization.*

4. It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

5. The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Example:

```
#include <iostream>
using namespace std;
class Area {
  private:
    double length;
    double height;
  public:
    // initialize variables with parameterized constructor
    Area(double len, double hgt) {
      length = len;
      height = hgt;
    }
    // copy constructor with a Area object as parameter
    // copies data of the obj parameter
    Area(Area &obj) {
      length = obj.length;
      height = obj.height;
    }
    double calculateArea() {
      return length * height;
    }
};
int main() {
  // create an object of Area class
  Area a1(10.5, 8.6);
  // copy contents of a1 to a2
  Area a2 = a1;
  // print areas of a1 and a2
  cout << "Area of a1: " << a1.calculateArea() << endl;
  cout << "Area of a2: " << a2.calculateArea();
  return 0;
}
```

Types of Copy Constructors:

1. **Default Copy Constructor:**

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

```
#include <iostream>
using namespace std;

class Myclass {
    int id;
    string name;
public:
    void init(int x, string y)
    { id = x;
      name=y;
    }
    void display()
    { cout << endl << "ID=" << id;
      cout <<"\n" << "Name=" <<name;

    }
};

int main()
{
    Myclass obj1;
    obj1.init(10,"Arunava");
    obj1.display();
    // Implicit Copy Constructor Calling
    Myclass obj2(obj1); // or obj2=obj1;
    obj2.display();
    return 0;
}
```

Output:

```
ID = 10
Name = Arunava
ID = 10
Name = Arunava
```

2. User Defined Copy Constructor

A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written

********** **Parameterized constructor**********

In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.

```
#include <iostream>
using namespace std;
class Area {
  private:
```

```
    double length;
    double height;
  public:

// initialize variables with parameterized constructor
  Area(double len, double hgt) {
      length = len;
      height = hgt;
    }
  double calculateArea() {
      return length * height;
    }
};
int main() {
  // create an object of Area class
  Area a1(10.5, 8.6);
  Area a2(11,12);
  // print areas of a1 and a2
  cout << "Area of a1: " << a1.calculateArea() << endl;
  cout << "Area of a2: " << a2.calculateArea();
  return 0;
}
```

********** **Default Constructor**

```
#include <iostream>
using namespace std;
class Area {
  private:
    double length;
    double height;
  public:

// initialize variables with Default constructor
  Area() {
      length = 5;
      height = 10;
    }
  double calculateArea() {
   return length * height;
    }
};
int main() {
  // create an object of Area class
  Area a1;
   // print areas of a1
  cout << "Area of a1 with Default Values : " << a1.calculateArea() << endl;
   return 0;
}
```

**********Example of Menu Driven Program Ans-3**********

```
#include<iostream>
using namespace std;
class Myclass{
public:
      int x,y;
       Myclass(int x1,int y1)
       {
           x=x1;
           y=y1;
       }
       void add()
       {
         int z;
         z=x+y;
         cout<<"Add:"<<z<<endl;
       }
       void sub()
       {
         int z;
         z=y-x;
         cout<<"Subtract:"<<z<<endl;
       }
       void div()
       {
         double z=y/x;
         cout<<"Divide:"<<z<<endl;
       }
       void mult()
       {
         int z=x*y;
         cout<<"Multiply:"<<z<<endl;
       }
};
int main()
{

  int n=0,ch=0,m=0;
  cout<<"Enter 1st Number:";
  cin>>n;
  cout<<"Enter 2nd number:";
  cin>>m;
  Myclass M1(n,m);
  cout<<"Press"<<endl<<"1.To add two
numbers"<<endl;
  cout<<"2.To Subtract two numbers"<<endl;
  cout<<"3.Divide two numbers"<<endl;
  cout<<"4.Multiply two  numbers"<<endl;
  cout<<"5.Exit the program"<<endl;
  cout<<endl<<"Do you want to continue, then
press 1";
    cin>>n;
    while(1)
    {
       cout<<"Enter your choice:";
       cin>>ch;
       switch(ch)
       {
          case 1:
               M1.add();
                break;
          case 2:M1.sub();
                break;
          case 3:M1.div();
                break;
          case 4:M1.mult();
                break;
          case 5: exit(0);
                 break;
       }

    }
    return 0;
}
```

Constructor with Dynamic allocation, Destructors. **Friend Functions and Static members,**

Objects: memory considerations for objects, new and delete operators.

When allocation of memory is done dynamically using dynamic memory allocator **"new" inside a constructor**, it is known as **dynamic constructor**. By using this, we can dynamically initialize the objects. We can say that whenever allocation of memory is done dynamically using **new** inside a constructor, it is called dynamic constructor.

**********Example-1**********

```
#include <iostream>
using namespace std;

class Myclass {

   const char* p;

 public:
   // default constructor
   Myclass()
   {

      // allocating memory at run time
      p = new char[6];
      p = "Example of a Dynamic Constructor";
   }

   void display()
   {
      cout << p << endl;
   }
};

int main()
{
   Myclass obj;
   obj.display();
}
```

**********Example-2**********

```
#include<iostream>
using namespace std;

class Myclass
{
  int num1;
  int num2;
  int *ptr;

public:
  // Default constructor (Dynamic constructor also)
  Myclass()
  {
    num1 = 0;
    num2 = 0;
    ptr = new int;
    *ptr= 5;
  }

  //dynamic constructor with parameters
  Myclass(int x, int y, int z)
  {
    num1 = x;
    num2 = y;
     ptr = new int;
    *ptr = z;
  }

 void display()
 {
   cout << num1 << " " << num2 << " " << *ptr;
 }
};

int main()
{
  cout<<" This is Default & Dynamic Constructor";
  Myclass obj1;
  cout << endl;
  obj1.display();
  cout<<"\n"<<" This is Parametrized & Dynamic
Constructor";
  Myclass obj2(3, 5, 11);
  cout << endl;
  obj2.display();
}
```

Destructors in C++ are members functions in a class that delete an object. They are called when the class object goes out of scope such as when the function ends, the program ends, a delete variable is called etc.

Destructors are different from normal member functions as they don't take any argument and don't return anything. Also, destructors have the same name as their class and their name is preceded by a tilde (~).

A program on destructors in C++ is given below.

```
#include<iostream>
using namespace std;
class Myclass {
  private:
  int a,b;
  public:
  Myclass (int a1, int b1) {
    cout<<"Inside Constructor"<<endl;
    a = a1;
    b = b1;
  }
  void display() {
    cout<<"a = "<< a <<endl;
    cout<<"b = "<< b <<endl;
  }
  ~Myclass() {
    cout<<"Inside Destructor";
  }
};
int main() {
  Myclass obj1(10, 20);
  obj1.display();
  return 0;
}
```

Friend Functions & Friend Class

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

1. Write a program using friend function to find area and volume of a Cube.
2. Write a program which has a class that finds factorial of a number. Make this class a friendly into another class and print the factorial value in this class.
3. Write a program with the following implementation details:
    a) Class A to check whether the input is even or odd and a setInput(int)
    b) Class B is friendly to A and checks the **input of Class A** whether the input is a prime number.
    c) A Friendly function Display which will output even/odd or prime/not prime.

A 'global friend function' allows access to all the private and protected members of the global class declaration.

******************************Friendly Function***************************

```
#include <iostream>

using namespace std;

class MyBox {
   double w;
   double l;
   double h;
   double area;
   double volume;
   public:
     friend void myDisplay( MyBox );
     void setInput( double a, double b,double c );
     void findVol();
     void findArea();
};

// Member function definition
void MyBox::setInput(double a, double b,double c ) {

   l=a;
   w = b;
   h=c;
}
void MyBox::findArea()
{
    area= l*w;
}
void MyBox::findVol()
{
    volume= l*w*h;
}

// Note: myDisplay() is not a member function of any class.
void myDisplay(MyBox box) {
   /* Because myDisplay() is a friend of Box, it can
   directly access any member of this class */
   cout << "Area of box : " << box.area <<endl;
   cout << "Volume of Box:"<< box.volume<<endl;
}

// Main function for the program
int main() {
   MyBox box;
   // set box width without member function
   box.setInput(10.0,20, 30);
```

```
   box.findArea();
   box.findVol();
   // Use friend function to print the wdith.
   myDisplay( box );
   return 0;
}
```

****Friendly Class**

```
#include <iostream>
using namespace std;
class A
   {
     int i,n,fact=1;
     public:
     void setInput(int );
     void factorial();
     friend class B;
   };
void A:: setInput(int x)
     {
       n=x;
     }
void A:: factorial()
       {
          for(i=n;i>1;i--)
              {
                 fact=fact*i;
              }
       }
  class B
 {
    public:
       void displayFact(A &a)
       {
          cout<< endl<<"The Factorial is";
```

```
        cout<< endl<< a.fact;
      }
};
  int main()
  {
     A A1;
     A1.setInput(6);
     A1.factorial();
     B B1;
     B1.displayFact(A1);
  }
```

**********Function is friendly for two classes**********.

```
#include <iostream>
using namespace std;
class B; //Declaration
class A
  {

    public:
     int i,n,flag1=0;
    void setInput(int );
    void evenodd();
    friend class B;
    friend void myOutput(A &a, B &b);

  };

void A:: setInput(int x)
    {
      n=x;
    }

void A:: evenodd()
      {
        cout<< endl<<"Class A Number chk for Even/Odd is=";
        cout<< n;
        if (n%2==0)
           flag1=1;
          // cout<< fact;
```

```
        }

 class B
 {
    int flag2=0;
    public:
    friend void myOutput(A &a, B &b);
      void chkPrime(A &a)
      {   int i;
         cout<< endl<<"Friendly Class B Number chk for Prime is=";
         cout<< a.n;
         for(i=2;i<=(a.n/2);i++)
            {
              if(a.n%i==0)
               {
                  flag2=1;
                  break;
               }
            }
      }
 };
void myOutput(A &a, B &b)
{

   cout<<endl<<"---The output is---";
   if (a.flag1==1)
       cout<<endl<<"The Number is Even";
    else
       cout<<endl<<"The Number is Odd";
    if (b.flag2==1)
       cout<<endl<<"The Number is Not Prime";
    else
       cout<<endl<<"The Number is Prime";
}
int main()
   {
       A A1;
       A1.setInput(11);
       A1.evenodd();
       B B1;
       B1.chkPrime(A1);
       myOutput(A1,B1);    }
```

Static members, Objects.

Static data members are class members that are declared using static keywords. A static member has certain special characteristics. These are:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is being created, even before main starts.
- It is visible only within the class, but its lifetime is the entire program

1.Static Variables in a Function: In this type, we will declare a variable as statically defined inside the function and we will call this function again and again. Even if we call a function multiple times, its value will be initialized only once rather than again and again and we can store the previous state of this variable till the next call of the function.

```
#include<iostream>
#include<string>
using namespace std;
void demo()
{
        // static variable
        static int count = 0;
        cout << count << "";
        // value is updated and  will be carried to next  function calls
        count++;
}
int main()
{
        for (int i=0; i<5; i++)
                demo();
        return 0;
}
```

Output:

01234

In this example, the count variable has been declared as static that means, memory for this count variable will be allocated for the lifetime of the program. Therefore, whenever we call a function its previous value will be stored and carried throughout the lifetime of the program.

Even if the function is called multiple times, space for the static variable is allocated only once**. This is useful in applications where we have to store the previous state of the function.**

2.Static variables in a class:

Now, as we know, the static variable is initialized only once. So, even if we create multiple objects of the same class, all objects will share the same static variable rather than having multiple copies of the same static variable. Also because of this reason, **static variables cannot be initialized using constructors**.
** We can see that we are trying to create multiple copies of the same static variable in obj1 & obj2, **but that gave an error as shown below.**

```
#include<iostream>
using namespace std;
class Myclass
{
public:
        static int i;
        Myclass()
        {
                // Do nothing
        };
};
int main()
{
Myclass obj1;
Myclass obj2;
obj1.i =2;
obj2.i = 3;
// prints value of i
cout << obj1.i<<" "<<obj2.i;
}
```

We can define the static variable **using the class name and scope resolution operator** with the **variable name outside of the class** as shown below:

Correct Way:

```
#include<iostream>
using namespace std;

class Myclass
{
public:
        static int i;

        Myclass()
        {
                // Do nothing
        }
};

int Myclass::i = 999;

int main()
{
        Myclass obj;
        // prints value of i
        cout << obj.i;
}
```

3.Static functions in a class:

- Static member functions are allowed to **access only the static data members** or other **static member functions**, they **cannot access** the **non-static data members** or member functions of the class.
- We are allowed to invoke a static member function using the object and the '.' operator but it is **recommended** to invoke the **static members** using the **class name and the scope resolution operator**.

- By declaring a functioning member as **static**, you make it independent of any particular object of the class. A **static member** function can be **called even if no objects** of the class exist and the static functions are accessed using only the **class name and the scope resolution operator**.

```
#include<iostream>
using namespace std;
class Myclass
{
public:
  // static member function
    static void Mymsg()
    {
            cout<<"This is a Static Method & No object is required to invoke it";
    }
};
// main function
int main()
{
    // invoking a static member function
    Myclass::Mymsg();
}
```

4. Class objects as static: Objects can also be declared static

Case-1:

We created an object obj of class **Myclass** inside the **if block** as shown. So, the **scope of the variable "i" will be limited to only if block**. So, the first constructor is called and at the time when the **control of the program gets out of the if block**, the **destructor will be called because the scope of "i" or object obj is limited to if block only**, and then the last '**cout**' will be called.

```
#include<iostream>
using namespace std;
class Myclass
{
        int i;
        public:
```

```
        Myclass()
        {
                i = 0;
                cout << "Inside Constructor\n";
        }
        ~Myclass()
        {
                cout << "Inside Destructor\n";
        }
};

int main()
{
        int x = 0;
        if (x==0)
        {
                Myclass obj;
        }
        cout << "End of main\n";
}
```

Output:

```
Inside Constructor
Inside Destructor
End of main
```

Case-2: Here, we define the **object obj as static**. Now what happens is that the **scope** of this object **obj is now outside of that if block** too rather than limited to if block as seen previously. So, **even if the control flow of the program** will get **out of the if block**, the **destructor will be called at the end of the program** execution as seen in the output:

```
#include<iostream>
using namespace std;

class Myclass
{
        int i = 0;
        public:
        Myclass()
        {
```

```
                i = 0;
                cout << "Inside Constructor\n";
        }

        ~Myclass()
        {
                cout << "Inside Destructor\n";
        }
};

int main()
{
        int x = 0;
        if (x==0)
        {
                static Myclass obj;
        }
        cout << "End of main\n";
}
```

Output:

```
Inside Constructor
End of main
Inside Destructor
```

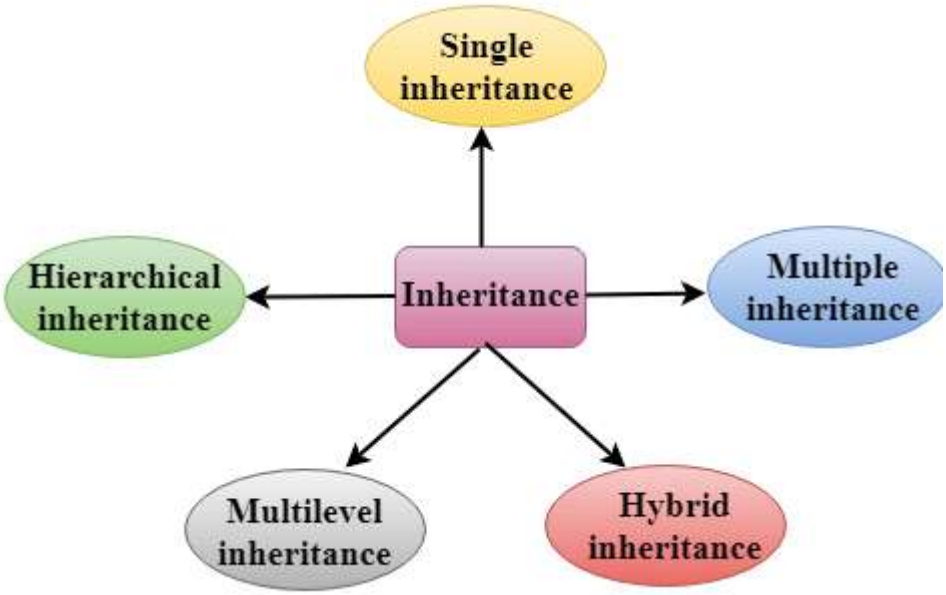
5. Write a program in C++ to overload concatenation (+) operator.

6. Write a program in C++ to overload Binary (+) operator

Inheritance-1 :

C++ supports five types of inheritance:

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



class Child_class_name :: visibility-mode Parent_class_name

{

}

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

Parent_class_name: It is the name of the base class.

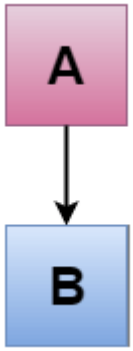
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

- 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 base class.

Note:

- In C++, the **default mode of visibility is private.**
- **The private members of the base class are never inherited.**

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where **'A'** is the base class**,** and **'B'** is the derived class

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance **which inherits the fields only**.

```
#include <iostream>

using namespace std;

class Student {

  public:

  string name =" Anil Chowbey";

};

class Programmer: public Student {

  public:

  string domain= "C programming";

  };

int main(void) {

    Programmer p1;

    cout<<"Name: "<<p1.name<<endl;

    cout<<"Domain of Programming:
"<<p1.domain<<endl;

   return 0;

}
```

C++ Single Level Inheritance Example: Inheriting functions

```
#include <iostream>
using namespace std;
 class Bird {
   public:
   void hasFeathers() {
cout<<"Birds have feathers"<<endl;
 }
 };
 class flyingBird: public Bird
 {
   public:
    void canFly(){
    cout<<"Birds can Fly";
     }
   };
int main(void) {
    flyingBird f1;
    f1.hasFeathers();
    f1.canFly();
    return 0;
}
```

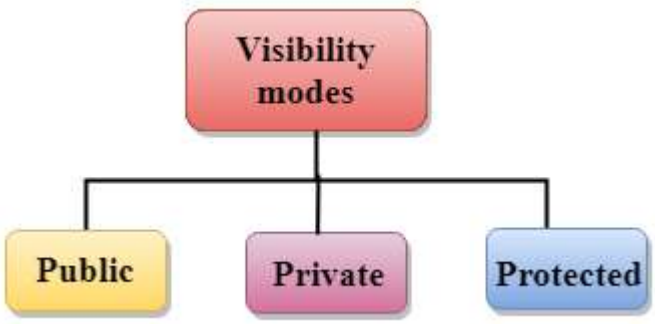
Private Inheritance: class A is privately inherited. Therefore, the **div() function** of class 'A' **cannot be accessed by the object of class B**. It can only be accessed by the **member function of class B.**

```
#include <iostream>
using namespace std;
class A
{
   int a = 10;
   int b = 5;
   public:
   float div()
   {
      int c = a/b;
      return c;
   }
};
 class B : private A
{
   public:
   void display()
   {
      int result = div();
      std::cout <<"Division of a by b is :
"<<result<< std::endl;
   }
};
int main()
{
   B b;
   b.display();
    return 0;
}
```

How to make a Private Member Inheritable:

The private member is not inheritable. If we modify the visibility mode by **making it public, but this takes away the advantage of data hiding.**

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the **member functions within the class** as well as the **class immediately derived from it**.



- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

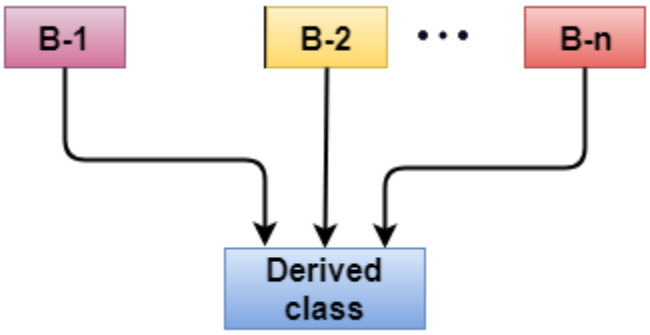
C++ Multilevel Inheritance:

Multilevel inheritance is a process of deriving a class from another derived class.
When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

```
#include <iostream>
using namespace std;
 class Bird {
   public:
   void hasFeathers() {
cout<<"Birds have feathers"<<endl;
 }
 };
class flyingBird: public Bird
 {
   public:
    void canFly(){
    cout<<"Birds can Fly";
     }
   };
 class crow: public flyingBird
 {
    public:
    void color(){
        cout<<"\n"<< "The color of Crow-a flying
Bird is Black";
    }
 };

int main(void) {
   crow f1;
   f1.hasFeathers();
   f1.canFly();
   f1.color();
   return 0;
}
```

C++ Multiple Inheritance:

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



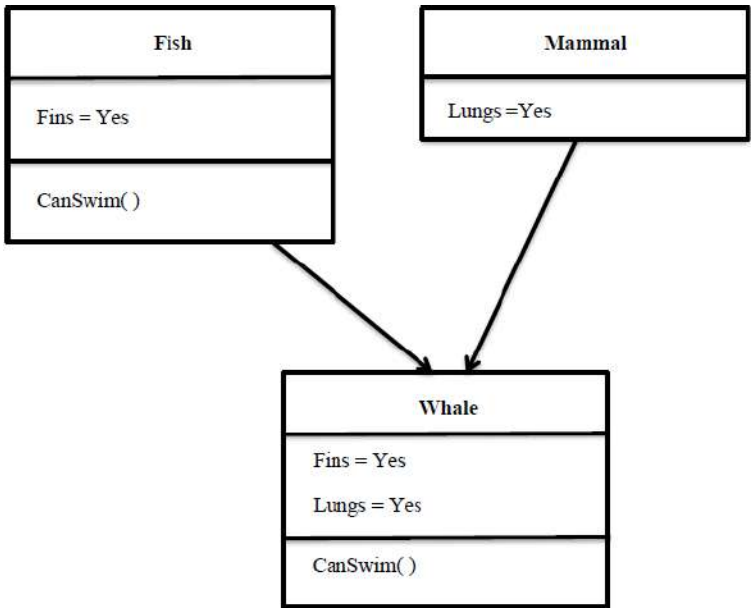
Ambiguity Resolution in Inheritance:

Ambiguity can occur in using the multiple inheritance when a function with the same name occurs in more than one base class.

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      cout << "Class A" << endl;
   }
};
class B
{
   public:
   void display()
   {
      cout << "Class B" << endl;
   }
};
class C : public A, public B
{
  // Error
  /*void view()
   {
      display();  //Cannot Resolve which display()-
Ambiguous
   }
 */
  // The above issue can be resolved by using the
class resolution operator with the function.
  //In the above example, the derived class code can
be rewritten as:
  public:
   void view()
   {
A :: display();        // Calling the display()
function of class A.
 B :: display();       // Calling the display()
function of class B.
   }

};
int main()
{
   C c;
   c.view();
   return 0;
}
```

Inheritance:

1. Write a program which will be having the following details:
   a) A Base class – Inputs the Name and Age of a Person using a getData function , and uses a Display function which prints it.
   b) A Derived class- It inherits the Base class and uses a getData function to add Salary and Address of the Person. It uses a Display function to print the attributes.
2. Write a program using the concept of inheritance as per the following descriptions: In class A – two variables are declared. Class A is the parent class. Class B is the derived class which takes the variables „a" and „b" from parent class and multiplies with a third variable of its own and displays the result.
3. **Implement the following class diagram:**



4. Create a class for "Employee" with attributes name, age and address. Create a class "Manager" and "Worker" which will inherit the Employee class. The manager class has extra attributes –department and salary. The worker class has attributes number of days worked, daily wage and total salary. Add necessary member functions to calculate total salary of worker and display the operation of all classes.

```
#include  <iostream>
using namespace std;
class baseA
{
  int age;
  char name[50];
  public:
    void getData();
    void Display();

};
void baseA :: getData()
   {
    cout<<"Enter the Name=";
```

```
    cin>> name;
    cout<<endl<< "Enter the Age=";
    cin>> age;
   }
void baseA :: Display()
   {
      cout<<endl<<"The Output for the Base Class are ";
      cout<< "Name="<<name<<" ";
      cout<< "Age="<< age<< " ";
   }
class derivedB : public baseA //derivedB is derived from baseA
{
   int sal;
   char add[30];
   public:
      void getData();
      void Display();
};
void derivedB :: getData()
   {
      baseA::getData(); //Calling baseclass getdata
      cout<< "Input Salary=";
      cin>> sal;
      cout<< "Input Address=";
      cin>> add;
   }
void derivedB :: Display()
{
   baseA::Display(); //Calling baseclass Display
   cout<<endl<< "The Output for the Derived (child) Class are ";
   cout<< "Salary="<<sal<<"";
   cout<< "Address="<<add<< endl;

}
int main()
{
   baseA A;
   derivedB B;
   B.getData();
   B.Display();
   return 0;
}
```

```
#include <iostream>
using namespace std;
```

```
class A
{
  public:
    int a,b;
    void getData();
};

void A :: getData()
    {
     cout<<"Enter the 1st Variable=";
     cin>> a;
     cout<<endl<< "Enter the 2nd Variable=";
     cin>> b;
    }

class B : public A // B is derived from A
{
    int z;
    public:
    void Display();
};

void B :: Display()
{
    A:: getData(); //Calling baseclass getdata
    cout<<endl<< "The Output for the Derived (child) Class are ";
    z= a*b;
    cout<< "Output="<<z<<endl;
 }
int main()
{
    A A1;
    B B1;
    B1.Display();
    return 0;
}
```

**********Overriding Example**********

```
#include <iostream>
using namespace std;
class Fish
{
  public:
    char fins;
    void getData1(char x);
    void canSwim();
```

```
};

void Fish::getData1(char x)
   {
      fins=x;
   }
void Fish::canSwim()
   {
      if (fins=='Y')
      cout<< "It is a Fish, it can Swim because it has Fins"<<endl;
   }
class Mammal
{
   public:
      char lungs;
      void getData2(char x);

};

void Mammal::getData2(char x)
   {
      lungs=x;

   }

class Whale:public Fish,public Mammal
{
   public:
   void canSwim();

};

void Whale:: canSwim() // Over-riding the canSwim function
{
  cout<<"\n"<<"----------Output ----------"<<"\n";
  cout<<"Fins="<<fins<<"\n";
  cout<<"Lungs="<< lungs<<"\n";
  if((fins=='Y') && (lungs=='Y'))
      cout<< "Whale is a Mammal since it has Lungs and Fins " <<endl;
   else
      cout<< "It is not a Mammal"<<endl;

}


int main()
```

```
{
  char x,y;
  Whale W1;
  cout<<"Enter Fins Y/N"<<endl;
  cin>>x;
   cout<<"Enter Lungs Y/N"<<endl;
  cin>>y;
  W1.getData1(x);
  W1.getData2(y);
  W1.canSwim();
 }
```

**********Over-riding Example **********

```
#include <iostream>
using namespace std;
class Employee
{
 public:
 string name,address;
 int age;
 void getEmployee(string,string,int x);
 void display();
};

void Employee::getEmployee(string n1, string add,int x)
{
   name = n1;
   address = add;
   age= x;
}

void Employee::display()
{
   cout<<"Name="<<name<<endl<<"Address="<<address<< endl<<"Age="<<age<<endl;

}

class Manager: public Employee
{
   public:
   string dept;
   int salary;
   void getEmployee(string,string,int x,string,int);
   void display();//Over-riding
};
```

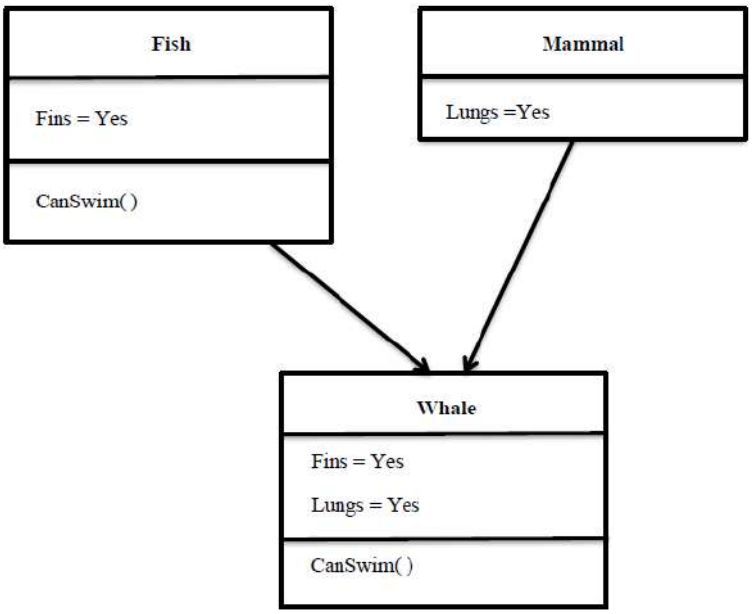
```
void Manager ::getEmployee(string n1,string add,int x,string d, int sal)
{
   name = n1;
   address = add;
   age= x;
   dept=d;
   salary= sal;

}
void Manager ::display()
{
   cout<< "Name="<<name<<endl;
   cout<<"Address="<<address<<endl;
   cout<<"Age="<<age<<endl;
   cout<<"Department="<<dept<<endl;
   cout<<"Salary="<<salary<<endl;
}
class Worker: public Employee
{
  public:
  int daysworked ;
  int dailywage;
  void getEmployee(string,string,int ,int,int);
  void display();//Over-riding
};

void Worker ::getEmployee(string n1,string add,int x,int d, int w)
{
   name = n1;
   address = add;
   age= x;
   daysworked=d;
   dailywage=w;

}

void Worker ::display()
{
   int t= daysworked* dailywage;
   cout<< "Name="<<name<<endl;
   cout<<"Address="<<address<<endl;
   cout<<"Age="<<age<<endl;
   cout<<"Daysworked="<<daysworked<<endl;
   cout<<"Daily wage="<<dailywage<<endl;
   cout<<"Total Salary="<<t<<endl;
}
```

```
int main()
{
   //Employee E1;// Calling the Parent Class and its functions
   //E1.getEmployee("Arun","Asansol", 47);
  // E1.display();
   //Manager m1;// Calling Manager Subclass- Overriding display function
   //m1.getEmployee("Arun","Asansol", 47,"ECE",93500);
   //m1.display();
   Worker W1;// Calling Worker Subclass- Overriding display function
   W1.getEmployee("Arun","Asansol", 47,30,500);
   W1.display();
}
```

Inheritance-2 :

Type your text

1. Write a program which will be having the following details:
   a) A Base class – Inputs the Name and Age of a Person using a getData function , and uses a Display function which prints it.
   b) A Derived class- It inherits the Base class and uses a getData function to add Salary and Address of the Person. It uses a Display function to print the attributes.
2. Write a program using the concept of inheritance as per the following descriptions: In class A – two variables are declared. Class A is the parent class. Class B is the derived class which takes the variables 'a' and 'b' from parent class and multiplies with a third variable of its own and displays the result.
3. **Implement the following class diagram:**



4. Create a class for "Employee" with attributes name, age and address. Create a class "Manager" and "Worker" which will inherit the Employee class. The manager class has extra attributes –department and salary. The worker class has attributes number of days worked, daily wage and total salary. Add necessary member functions to calculate total salary of worker and display the operation of all classes.

```
#include <iostream>
using namespace std;
class baseA
{
  int age;
  char name[50];
  public:
    void getData();
    void Display();

};
void baseA :: getData()
   {
    cout<<"Enter the Name=";
```

```
    cin>> name;
    cout<<endl<< "Enter the Age=";
    cin>> age;
   }
void baseA :: Display()
   {
      cout<<endl<<"The Output for the Base Class are ";
      cout<< "Name="<<name<<" ";
      cout<< "Age="<< age<< " ";
   }
class derivedB : public baseA  //derivedB is derived from baseA
{
   int sal;
   char add[30];
   public:
      void getData();
      void Display();
};
void derivedB :: getData()
   {
      baseA::getData(); //Calling baseclass getdata
      cout<< "Input Salary=";
      cin>> sal;
      cout<< "Input Address=";
      cin>> add;
   }
void derivedB :: Display()
{
   baseA::Display(); //Calling baseclass Display
   cout<<endl<< "The Output for the Derived (child) Class are ";
   cout<< "Salary="<<sal<<"";
   cout<< "Address="<<add<< endl;

}
int main()
{
   baseA A;
   derivedB B;
   B.getData();
   B.Display();
   return 0;
}
```

```
#include <iostream>
using namespace std;
```

```
class A
{
  public:
    int a,b;
    void getData();
};

void A :: getData()
    {
     cout<<"Enter the 1st Variable=";
     cin>> a;
     cout<<endl<< "Enter the 2nd Variable=";
     cin>> b;
    }

class B : public A  // B is derived from A
{
    int z;
    public:
    void Display();
};

void B :: Display()
{
    A:: getData(); //Calling baseclass getdata
    cout<<endl<< "The Output for the Derived (child) Class are ";
    z= a*b;
    cout<< "Output="<<z<<endl;
 }
int main()
{
    A A1;
    B B1;
    B1.Display();
    return 0;
}
```

**********Overriding Example**********

```
#include <iostream>
using namespace std;
class Fish
{
  public:
    char fins;
    void getData1(char x);
    void canSwim();
```

```
};

void Fish::getData1(char x)
   {
      fins=x;
   }
void Fish::canSwim()
   {
      if (fins=='Y')
      cout<< "It is a Fish, it can Swim because it has Fins"<<endl;
   }
class Mammal
{
   public:
      char lungs;
      void getData2(char x);

};

void Mammal::getData2(char x)
   {
      lungs=x;

   }

class Whale:public Fish,public Mammal
{
   public:
   void canSwim();

};

void Whale:: canSwim() // Over-riding the canSwim function
{
  cout<<"\n"<<"----------Output----------"<<"\n";
  cout<<"Fins="<<fins<<"\n";
  cout<<"Lungs="<< lungs<<"\n";
  if((fins=='Y') && (lungs=='Y'))
      cout<< "Whale is a Mammal since it has Lungs and Fins " <<endl;
   else
      cout<< "It is not a Mammal"<<endl;

}


int main()
```

```
{
  char x,y;
  Whale W1;
  cout<<"Enter Fins Y/N"<<endl;
  cin>>x;
   cout<<"Enter Lungs Y/N"<<endl;
  cin>>y;
  W1.getData1(x);
  W1.getData2(y);
  W1.canSwim();
 }
```

**********Over-riding Example **********

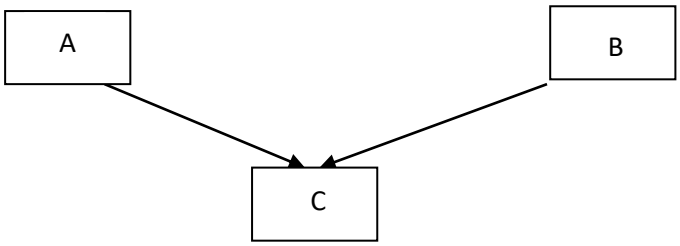
```
#include <iostream>
using namespace std;
class Employee
{
 public:
 string name,address;
 int age;
 void getEmployee(string,string,int x);
 void display();
};

void Employee::getEmployee(string n1, string add,int x)
{
   name = n1;
   address = add;
   age= x;
}

void Employee::display()
{
   cout<<"Name="<<name<<endl<<"Address="<<address<< endl<<"Age="<<age<<endl;

}

class Manager: public Employee
{
   public:
   string dept;
   int salary;
   void getEmployee(string,string,int x,string,int);
   void display();//Over-riding
};
```

```
void Manager ::getEmployee(string n1,string add,int x,string d, int sal)
{
   name = n1;
   address = add;
   age= x;
   dept=d;
   salary= sal;

}
void Manager ::display()
{
   cout<< "Name="<<name<<endl;
   cout<<"Address="<<address<<endl;
   cout<<"Age="<<age<<endl;
   cout<<"Department="<<dept<<endl;
   cout<<"Salary="<<salary<<endl;
}
class Worker: public Employee
{
  public:
  int daysworked  ;
  int dailywage;
  void getEmployee(string,string,int ,int,int);
  void display();//Over-riding
};

void Worker ::getEmployee(string n1,string add,int x,int d, int w)
{
   name = n1;
   address = add;
   age= x;
   daysworked=d;
   dailywage=w;

}

void Worker ::display()
{
   int t= daysworked* dailywage;
   cout<< "Name="<<name<<endl;
   cout<<"Address="<<address<<endl;
   cout<<"Age="<<age<<endl;
   cout<<"Daysworked="<<daysworked<<endl;
    cout<<"Daily wage="<<dailywage<<endl;
   cout<<"Total Salary="<<t<<endl;
}
```

```
int main()
{
   //Employee E1;// Calling the Parent Class and its functions
   //E1.getEmployee("Arun","Asansol", 47);
  // E1.display();
   //Manager m1;// Calling Manager Subclass- Overriding display function
   //m1.getEmployee("Arun","Asansol", 47,"ECE",93500);
   //m1.display();
   Worker W1;// Calling Worker Subclass- Overriding display function
   W1.getEmployee("Arun","Asansol", 47,30,500);
   W1.display();
}
```

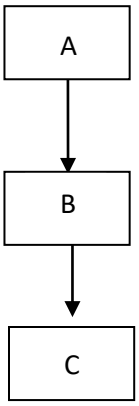
Constructors in Derived classes

Type-1 –Constructors in Multiple Inheritance



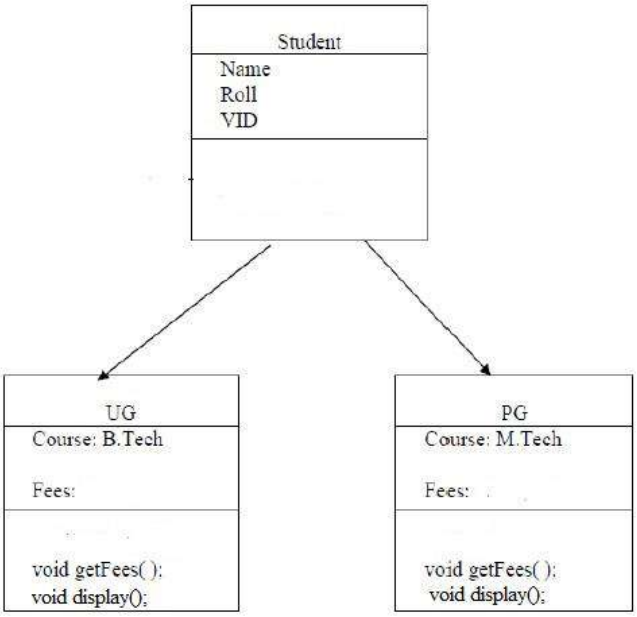

```
#include<iostream>
using namespace std;
class A{
   public:
   int x;
    A(int x1){
   x=x1;
   }
};
class B{
   public:
   int y;
   B(int y1)
   {
      y=y1;
   }
};
class C:public A,public B
{
   public:
   int z;
   C(int x1,int y1,int z1):A(x1),B(y1)
   {
      z=z1;
   }
   void display()
   {
      cout<<"x="<<x<<endl<<"y="<<y<<endl
       <<"z="<<z<<endl;
   }
};
int main()
{
   C C1(5,6,7);
   C1.display();
}
```

Type-2 –Constructors in Multi-level Inheritance

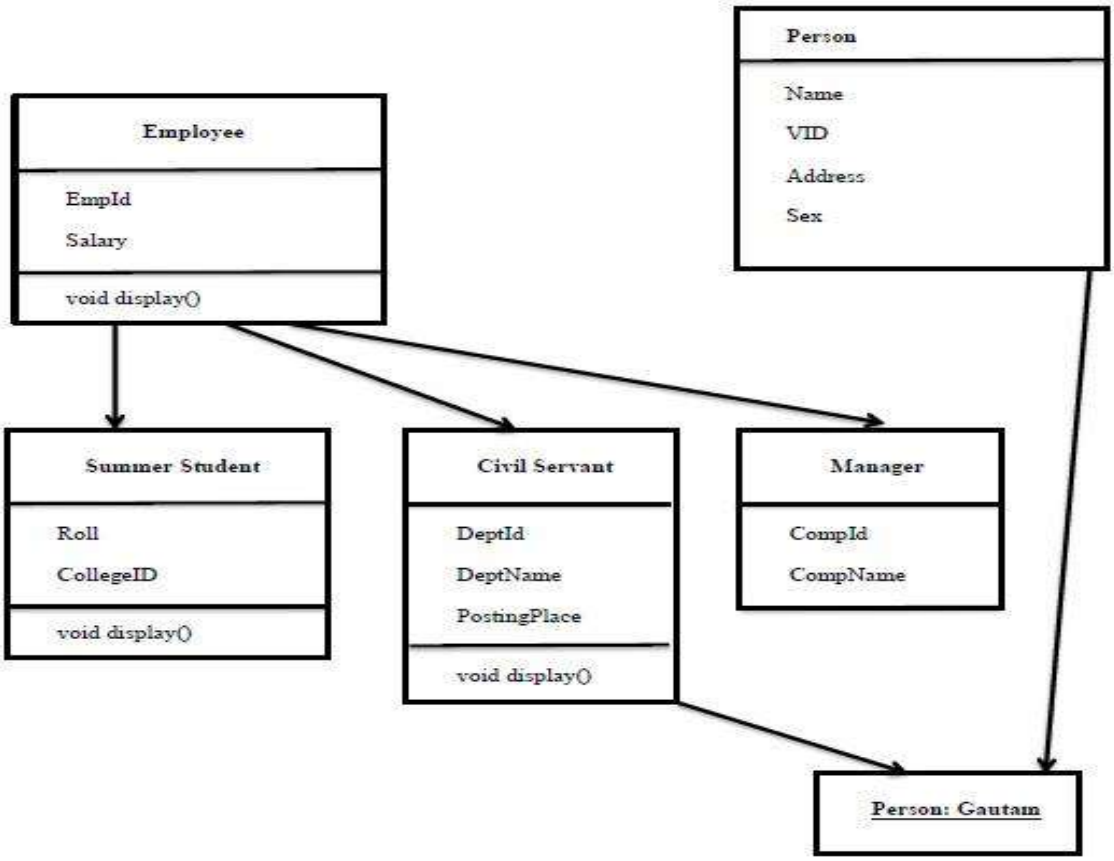



```
#include<iostream>
using namespace std;
class A{
   public:
   int x;
    A(int x1){
   x=x1;
   }
};

class B:public A
{
   public:
   int y;
   B(int x1,int y1):A(x1)
   {
      y=y1;
   }

};

class C:public B{
   public:
   int z;
   C(int x1,int y1,int z1):B(x1,y1)
   {
      z=z1;
   }
    void display()
   {

cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl;
   }
};

int main()
{
   C c1(5,6,7);
   c1.display();
}
```

1. Write a program using the concept of inheritance, constructors and over-riding to find
    a) Base Class-Area of a rectangular room
    b) Child Class- Volume of a rectangular room.
2. Use the concept of Over-riding to implement the class-diagram given below. Use constructors in all the classes.



3. Implement the class diagram given below:



Done program--

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

    public:
    int x,y;
    Room(int x1,int y1)
    {
       x=x1;
       y=y1;
    }
    void display()
    {
       int area;
       area=x*y;
       cout<<area;
    }
 };
 class Room1:public Room{
    public:
    int z;

    Room1(int x1,int y1,int z1):Room(x1,y1)
    {
       z=z1;
    }
  void display()
    {
       int area,vol;
       area=x*y;
       vol=x*y*z;
       cout<<"Area:"<<area<<endl;
       cout<<"Volume:"<<vol;
    }
 };
 int main()
 {
    Room1 R(5,6,7);
    R.display();
 }
```

Example of Nesting of classes-

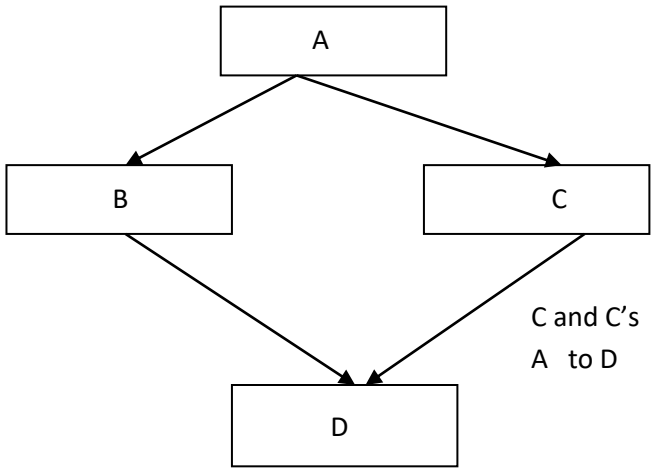
```
#include<iostream>
using namespace std;
class X
{
public :
   class B
   {
      int num;
   public:
      void getData(int n)
      {
         num = n;
      }
      void display()
      {
         cout<<"Nested class value :"<<num;
      }
   };
};
int main()
{
   X :: B B1;  //Gives the scope of the nested class B
   B1.getData(5);
   B1.display();
}
```

Virtual Classes: 1) The keyword **virtual** is ***not*** used in the derived functions

2) The number and type of parameters in the derived function **must** match the number and type in the initial declaration of the virtual function.

```
#include<iostream>
using namespace std;
class A
{
public:
   int x;
   void getData(int x1)
   {
      x = x1;
   }
};
class B : virtual public A
{
public:
   int y;
   void getData1(int x1,int y1)
   {
      x = x1;
      y = y1;
   }
};
class C : virtual public A
{
public:
   int z;
   void getData2(int x1,int z2)
   {
      x = x1;
      z = z2;
   }
};
class D : public B, public C
{
public:
   int f;
   void getData3(int x1,int x2,int x3,int x4)
   {
      x = x1;
      y = x2;
      z = x3;
      f = x4;
   }
   void display()
   {
      cout<<x<<endl<<y<<endl<<z<<endl<<f;
   }
};
int main()
{
   D d;
   d.getData3(5,10,15,20);
   d.display();
}
```

B and B's A comes to Class D



Class D has 2 A's and hence there is an ambiguity-
To remove ambiguity we need Virtual Class

Virtual functions key points:

- The keyword **virtual** is used ***only*** in the base class declaration.
- When a derived class redefines the virtual function in a base or predecessor class, this is referred to as over***riding*** the function. Don't confuse this with over***loading*** a function.

  When you call an over***riding*** function, you must supply the same number and type of parameters as the original. (You'll remember that the opposite is true for an over***loaded*** function call.)

- **Friend** functions cannot be virtual, nor can **constructor** functions, although **destructor** functions can be virtual.

```
#include <iostream>

using namespace std;


class Mybase {


// The keyword VIRTUAL must be used here!!

public:

  virtual void display() {

     cout << "I am in Base class\n";

     }

};

class Derived : public Mybase  {

public:

   void display() {

         cout << "I am in Derived class\n";

         }
};

int main() {

Mybase b1;     // object of the base class

b1.display();  // execute its virtual function

Derived d1;    //  object of the derived class

d1.display();  // execute its virtual function

}
```

- A derived class ***does not have to*** redefine (**override**) a virtual function. In other words, if a derived class **does not specify code for the virtual function**, then the **code in the predecessor function is used**.
- A derived function inherits the virtual functions of its predecessors which ***may*** or ***may not*** be the base class.

```
#include <iostream>

    using namespace std;

 class Mybase {

    // The keyword VIRTUAL must be used here!!

    public:

       virtual void display() {

           cout << endl << "Printing from base class" << endl;

          }

    };

 class Derived1 : public Mybase  {

    public:

       void display() {

           cout << endl << "Printing from first class" << endl;

           }
    };

 class Derived2 : public Mybase  {

     //derived class does not

    //specify code for the virtual function,

    //then the code in the predecessor function is used.

    };

int main() {

    Mybase base;          // object of the base class

    base.display();          // execute virtual function

    Derived1 d1;       // object of the derived class

    d1.display();    // execute virtual function

    Derived2 d2;     //  object for second

    d2.display();

}
```

Pure Virtual Functions:

However, in some cases, it does not make sense for the base class to define the virtual function at all. The actual method should really be implemented in each one of the descendent classes. The base function should simply provide a kind of ***placeholder*** for the virtual function and leave it up to the descendents to specify the individual methods.

A virtual function that is ***declared*** but not ***defined*** in a base class is referred to as a ***pure virtual function.***

virtual type function_name(parameter_list)=0;

Now when the base class uses a PURE virtual function, each descendent **must** override that function, or you will get a compile error. This makes sense, because the function has to be defined *somewhere*.

- Here's another definition:
  If a class contains a **pure virtual function**, that class is called an ***abstract class***.

- Because there is no definition for the function, you can***not*** create an object of that class.
  You ***can*** however, declare pointers to it.

Think of an **abstract class as a general class that lays the foundation for descendent classes** that define their own methods. This is the heart of polymorphism - let each class define its own operation.

Introduction to file handling, hierarchy of file stream classes, opening and closing of files, file modes, file pointers and their manipulators, sequential access, random access.

We have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

1) **ofstream -** This data type represents the output file stream and is used to create files and to write information to files.
2) **ifstream-** This data type represents the input file stream and is used to read information from files.
3) **fstream-** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

We can combine two or more of these values by **OR**ing them together. For example if we want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax –

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows –

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into our program using the stream extraction operator (>>) just as we use that operator to input information from the keyboard. The only difference is that we use an **ifstream** or **fstream** object instead of the **cin** object.

Example-1:

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
  char data[100];
  // open a file in write mode.
  ofstream outfile;
  outfile.open("myfile.dat");

  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);
  // write inputted data into the file.
  outfile << data << endl;
  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();
  // again write inputted data into the file.
  outfile << data << endl;
  // close the opened file.
  outfile.close();
  // open a file in read mode.
  ifstream infile;
  infile.open("myfile.dat");
  cout << "Reading from the file" << endl;
  infile >> data;
  // write the data at the screen.
  cout << data << endl;
 // again read the data from the file and display it.
  infile >> data;
  cout << data << endl;
  // close the opened file.
```

```
  infile.close();
  return 0;
}
```

Example-2:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main(){
  fstream newfile;
  newfile.open("myfile.txt",ios::out);  // open a file to perform write operation using file object
  if(newfile.is_open())      //checking whether the file is open
  {
    newfile<<"HELLO EVERYONE \n"; //inserting text
    newfile.close(); //close the file object
  }
  newfile.open("myfile.txt",ios::in); //open a file to perform read operation using file object
 if (newfile.is_open()){  //checking whether the file is open
    string tp;
while(getline(newfile, tp)){//read data from file object and put it into string.
 cout << tp << "\n";  //print the data of the string
    }
 newfile.close();  //close the file object.
  }
}
```

****ofstream** - This data type represents the output file stream and is used to create files and to write information to files.

****ifstream-** This data type represents the input file stream and is used to read information from files.

****fstream-** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files,

write information to files, and read information from files.

1. Opening a file----

Write a C++ program to create a file using file handling and check whether the file is created successfully or not.

If a file is created successfully then it should print "File Created Successfully" otherwise should print some error message.

```
#include <iostream>

#include<fstream>

using namespace std;

 int main()

{

    ofstream file;

   file.open("myfile.txt",);

    // If no file is created, then  error message.

 if(!file)
    {
    cout<<"Error in creating file!!!";
   }

  cout<<"File created successfully.";

 file.close();
 }
```

2. After the above file is created , open the file in write mode and insert "I am studying Files today" into it.

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
   fstream file;
   file.open("myfile.txt",ios::out);
   file<<" I am studying Files today ";
    if(!file)
  {
     cout<<"Error in creating file!!!";
  }
cout<<"I have successfully written into a file.";
 file.close();
 }
```

3. Create and Write a file using insertion operator

```
#include <iostream>
#include <fstream>
using namespace std;

int main()

 {
  // Another way Create and open a text file
   ofstream MyFile("a.txt");

// Write to the file

  MyFile << "Files can be tricky, but it is fun enough!";

// Close the file
```

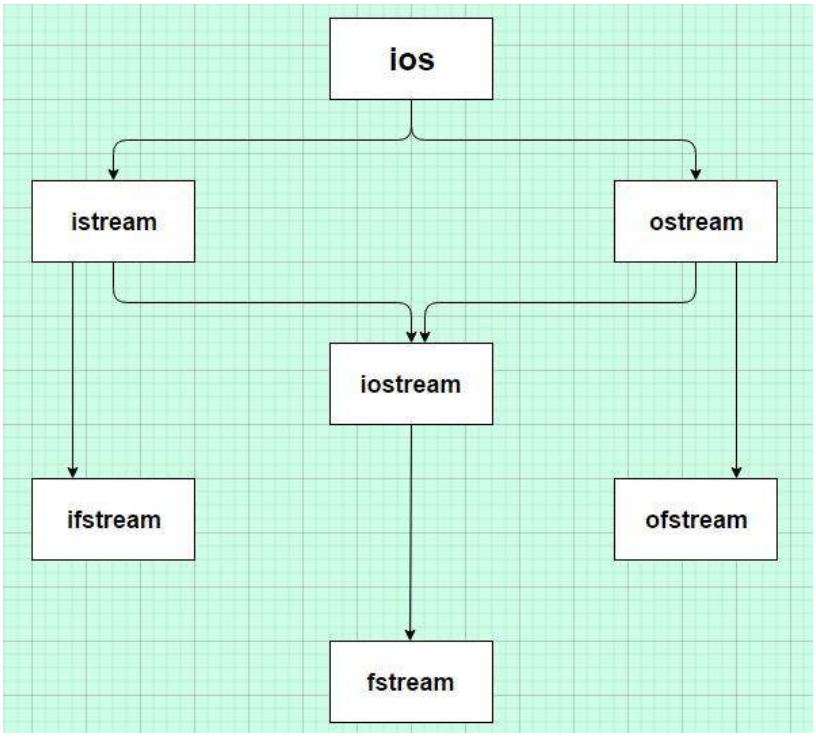
```
 MyFile.close();
}
```

4. Append Mode--ios::app

```
#include<iostream>
#include<string>
#include<fstream>
using namespace std;
int main()

 {

 ofstream fout;

ifstream fin;

 fin.open("a.txt");

fout.open ("a.txt",ios::app);

 if(fin.is_open())

 fout<<" I am appending  already present text file";

 cout<<"\n Data has been appended to file"<<endl;

 fin.close();

 fout.close();

  string word;

  fin.open("a1.txt");

  while (fin >> word)
        {
              cout << word << " ";

        }

}
```

5. Read /Write Example

```
#include <fstream>
#include <iostream>
using namespace std;

int main () {
  char data[100];

  // open a file in write mode.
  ofstream outfile;
  outfile.open("afile.dat");

  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);

  // write inputted data into the file.
  outfile << data << endl;

  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();

  // again write inputted data into the file.
  outfile << data << endl;

  // close the opened file.
  outfile.close();
  // open a file in read mode.
  ifstream infile;
  infile.open("afile.dat");

  cout << "Reading from the file" << endl;
  infile >> data;

  // write the data at the screen.
  cout << data << endl;

// again read the data from the file and display it.
  infile >> data;
  cout << data << endl;

  // close the opened file.
  infile.close();

}
```



C++ file handling provides a mechanism to store output of a program in a file and read from a file on the disk. So far, we have been using <iostream> header file which provide functions cin and cout to take input from console and write output to a console respectively. Now, we introduce one more header file <fstream> which provides data types or classes (ifstream , ofstream , fstream) to read from a file and write to a file.

A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.
Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

File Pointer & their Manipulation

The read operation from a file involves get pointer. It points to a specific location in the file and reading starts from that location. Then, the get pointer keeps moving forward which lets us read the entire file. Similarly, we can start writing to a location where put pointer is currently pointing. The get and put are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file. The functions which manipulate file pointers are as follows :

Function	Description
seekg( )	Moves the get pointer to a specific location in the file
seekp( )	Moves the **put** pointer to a specific location in the file
tellg( )	Returns the position of **get** pointer
tellp( )	Returns the position of **put** pointer

The function **seekg(n, ref_pos)** takes two arguments :

n denotes the number of bytes to move and **ref_pos** denotes the reference position relative to which the pointer moves. **ref_pos** can take one of the three constants :

- **ios :: beg** moves **the get pointer n bytes** from the **beginning of the file**,
- **ios :: end** moves **the get pointer n bytes** from the **end of the file** and
- **ios :: cur** moves **the get pointer n bytes** from the **current position**.

** If we don't specify the second argument, then **ios :: beg** is the **default reference position**.

The behaviour of **seekp(n, ref_pos)** is same as that of **seekg( ).**

Error Handling Functions:

Function	Return value and Meaning
eof()	Returns true (non zero) if end of file is encountered while reading; otherwise return false(zero)
fail()	Return true when an input or output operation has failed
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred.
good()	Returns true if no error has occurred.

We have a function named **seekg()** that we can use, specifically for **File Handling**, when we import **the fstream package.**

What does seekg() do?

Just like seekp(), seekg() moves the pointer to the desired location.

So what is the difference between seekp() and seekg()?

seekp() moves the put pointer to the desired location i.e. for write operation.

seekg() moves the get pointer to the desired location i.e. for read operation.

When we try reading a text file, the pointer is set to 0.

So we read from the 0th pointer all the way to the last pointer.

But if we want to read from a particular pointer, we use seekg().

Using **seekg()** we navigate to the **desired location in the text file** and **read from thereon**.

Note: Pointer value starts from 0

The syntax of seekg() function is given below:

<filehandling_obj>.seekg(<position_of_pointer>);

Example:

obj.seekg(6);

Here,

obj is our file handling object and **6 is the position** we would like to **set our pointer to**.

```
#include <fstream>
#include <iostream>
using namespace std;
int main ()
{
  fstream myobj;
  //obj.open("test.txt", ios::out);//Use this code if the file is not created..then write some lines on the *.txt file only after that execute the lines of code given below
  myobj.open("test.txt", ios::in);
  int pos=6;
  myobj.seekg(pos);
  while(!myobj.eof())
  {
    char ch;
    myobj>>ch;
    cout<<ch;
  }
  myobj.close();
}
```

The details of the code:

1. **fstream obj;**
   Here we are using the function fstream that enables read and write on a particular file.

2. **obj.open (“test.txt”, ios::in);**
   Here we are opening a text file called test with to

3. **int pos=6;**

   Here we are defining a variable pos of type int and storing the value 6 in it.

   We will use this to move to the 6th position.

4. **obj.seekg(pos);**

   We are moving to the 6th position in the test file.

5. **while(!obj.eof())**
   {
   char ch;
   obj>>ch;
   cout<<ch;
   }
   This is the basic code to read the text file. Reading from the position we set it to, all the way till the end of the file. Once we reach the end of the file, the while loop exits.

6. **obj.close();**

   And Finally, here we are closing the text file.

This function is useful especially when we know the position where we want to start overwriting from.

seekp()-We have a function called seekp which gets imported along with the other basic file handling operation in "fstream".

So what does seekp() do?
The seekp() function moves the pointer to the desired location.

When we create a text file or open a text file, our pointer is set to 0. So if we start writing in that text file, **we overwrite all previously written data**.

Using seekp() in C++ we can navigate the pointer to the desired location and write from thereon.

It takes 1 argument i.e. position you want to set the pointer to in the text file.
NOTE: pointer starts from 0 in the text file.

The syntax of seekp() function is given below:
<filehandling_obj>.seekp(<position_of_pointer>);

Example:
obj.seekp(5);

Here,
obj is our file handling object. On the other hand, **5 is the position we would like to set our pointer to.**

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
 fstream myobj;
 myobj.open ("mytest.txt", ios::out);
 myobj<<"Hello  World";
 int pos = 6;
 myobj.seekp(pos-1);
 myobj<<"..The text is changed";
 myobj.close();
 }
```

1. **fstream obj;**
   Here we are using the function fstream that enables read and write.
   obj is the object we are using to refer to fstream.
2. **obj.open ("test.txt", ios::out);**
   Here we are opening a text file called test with basic function to write hence the "ios::out"
3. **obj<<"Hello World";}**
   Here we are writing Hello World into the test.
   If you write only these 3 lines of codes along with the headers, test would have "Hello World" Printed in it.
4. **int pos = 6;**
   Here we are defining a variable named pos of type int and storing the value 6 in it.
   Why 6?
   Because we would like to start writing from the 6th position.
   That turns out to be "W" in Hello World.
5. **obj.seekp(pos-1);**
   Using the syntax, we wrote this line of code.
   Now we are moving to the sixth-1 position i.e. 5 in our text file.
   That turns out to be the space in Hello World.
6. **obj<<"…And here the text changed";**
   And now we are writing the following text into the text file.
   Since our pointer was moved to 5, we overwrite the space and "World".
7. **obj.close();**
   Here we are closing the test file we opened.

tellg() tells the **current pointer position** in the text file.
But instead of put pointer, it tells the **get pointer's location**

Say we have entered 20 characters in a text file, and you want to read it.
But along with reading **you also want to know the position of the last position in the text file.**
That is where tellg() comes into play.
It would give you the output 19 since counting starts from 0.

NOTE:
tellg = tell get pointer.
So it tells where your get pointer is.
I.E. for the read operation.

Syntax: pos = obj.**tellg();**

Where,
pos is a variable of type int.
obj is a file handling object.

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
 fstream obj;
 //obj.open ("test.txt", ios::out);
 obj.open ("test.txt", ios::in);
 char ch;
 int pos;
 while(!obj.eof())
 {
  obj>>ch;
  pos = obj.tellg();

  cout<<pos<<"."<<ch<<"\n";

 }
  obj.close();
}
```

1. **EOF is a predefined MACRO with the value of -1** that means EOF is not a character. So EOF is returned through the function which is going to read content from the file.
2. **obj.open ("test.txt", ios::in);**
   Here we are opening a text file called test with main purpose to read.
   I already have "Hello World" written in the test file.
3. **char ch;**
   int pos;
   We are defining 2 variables: "ch" of type char to read each character, and "pos" of type into to get the position of the get pointer.
4. Here we are using **pos=obj.tellg()** to get the pointer value of character being read.
   Finally, we are printing the position and the character each on a separate line.

```
while(!obj.eof())
{
obj>>ch;
pos = obj.tellg();
cout<<pos<<"."<<ch<<"\n";
}
```

4. **obj.close();**

And finally, we are closing the text file we opened.

tellp()-tellp() tells the current pointer position in the text file.

Say we have entered **20** characters in a text file. Your current pointer is at **19**. Since pointer value starts from 0 therefore **20-1=19**.
So if we use tellp() in this text file we would get an **output which is 19** of type int.

Using tellp() we can **edit the text file from the last position in the text file** or **wherever the current pointer is.**

NOTE:
tellp = tell put pointer.
So it tells where your put pointer is.

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
 fstream obj;
 obj.open ("test.txt", ios::out);
 obj<<"Hello World";
 int pos;
 pos = obj.tellp();
 cout<<pos;
 obj.close();
        }
```

1. **obj.open (“test.txt”, ios::out);**
   Here we are opening a text file called test with main purpose to write.
2. **obj<<“Hello World”;**
   Here we are writing Hello World into the text file. Notice there are 12 characters from H to d when we count from 1.
3. **int pos;**
   Here we defined a variable pos of type int to store the current put pointer position.
4. **pos = obj.tellp();**
   Using the tellp() syntax, we stored the current put pointer position in pos.
5. **cout<<pos;**
   Here we are printing the value of pos which has the put pointer’s current position.
6. **obj.close();**
   And finally, here we are closing the text file we opened.

Exercises:

1. Write a program to open a text file which has “Hello Everyone, open your laptops ” written on it. **Start to read the file from “’E’ of Everyone”.**

```
#include <fstream>
#include <iostream>
using namespace std;
int main ()
{
  fstream myobj;
  myobj.open("test.txt", ios::in);
  int pos=6;
  myobj.seekg(pos);
  while(!myobj.eof())
  {
     char ch;
     myobj>>ch;
     cout<<ch;
  }
  myobj.close();
}
```

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous,

b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.).

C++ provides the following specialized keywords for this purpose:
***try*:** Represents a block of code that can throw an exception.
***catch*:** Represents a block of code that is executed when a particular exception is thrown.
***throw*:** Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

Why Exception Handling?
The following are the main advantages of exception handling over traditional error handling:

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle only the exceptions they choose: A function can throw many exceptions, but may choose to handle some of them.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

C++ Exceptions:
When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:
Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable "age" is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that's why we are throwing an exception of type int in the try block (age), we can pass "int myNum" as the parameter to the catch statement, where the variable "myNum" is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

1. **Try/catch block**

```
#include <iostream>
using namespace std;

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

2. There is a special catch block called the **'catch all'** block, written as catch(…), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

```
#include <iostream>
using namespace std;

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
```

```
}
```

3. **Implicit type conversion doesn't happen for primitive types**. For example, in the following program, 'a' is not implicitly converted to int.

```
#include <iostream>
using namespace std;

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

4. **If an exception is thrown and not caught anywhere, the program terminates abnormally.** For example, in the following program, a char is thrown, but there is no catch block to catch the char.

```
#include <iostream>
using namespace std;

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```

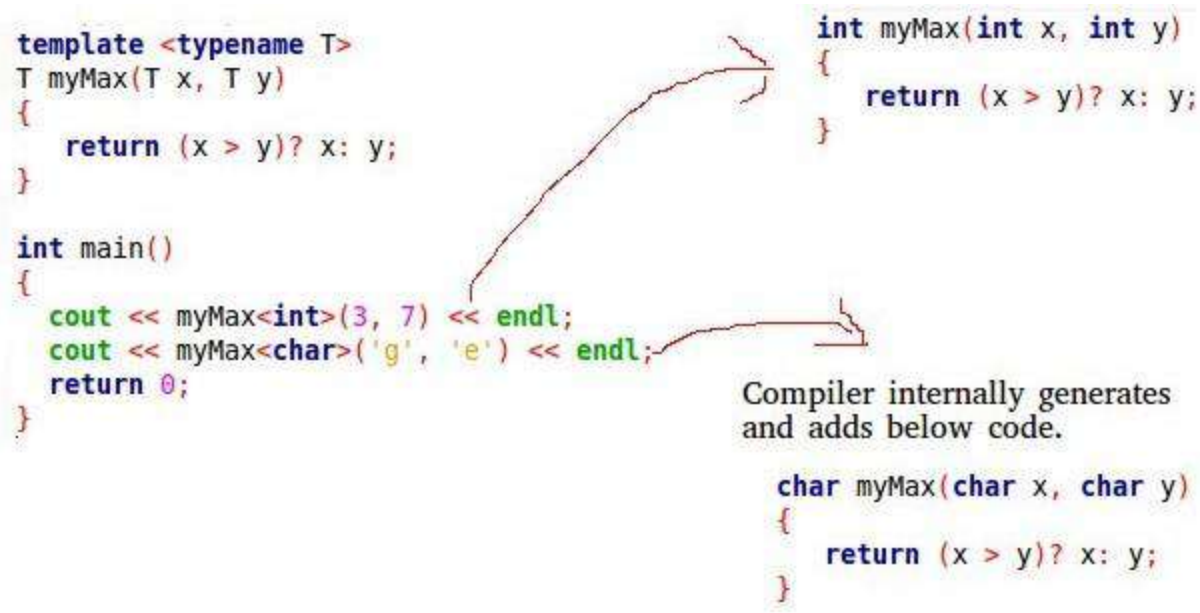
Templates in C++ :

A **template** is a simple yet very powerful tool in C++. The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

C++ adds two new keywords to support templates: ***'template'*** and ***'type name'***. The second keyword can always be replaced by the keyword **'class'**.

How Do Templates Work?

Templates are expanded at compiler time. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



Function Templates

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

```
#include <iostream>
using namespace std;

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T> T myMax(T x, T y)
{
   return (x > y) ? x : y;
}

int main()
{
   // Call myMax for int
```

```
    cout << myMax<int>(3, 7) << endl;
    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;
    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Output: 7

7

g