UNIT III – Syllabus

Operator Overloading and Type Conversion & Inheritance: The Keyword Operator, Overloading Unary Operator, Operator Return Type, Overloading Assignment Operator (=), Rules for Overloading Operators, Inheritance, Reusability, Types of Inheritance, Virtual Base Classes- Object as a Class Member, Abstract Classes, Advantages of Inheritance, Disadvantages of Inheritance.

What is meant by operator overloading? Explain with an example.

A symbol that is used to perform an operation is called an operator. It is used to perform operations on constants and variables. By using operator overloading, these operations are performed on objects. It is a type of polymorphism. The keyword **operator** is used to define a new operation for an operator.

Syntax:

```
return-type operator operator-symbol (list of arguments)
{
        // Set of statments
}
```

Steps:

1. Define a class which is to be used with overloading operators.
2. Declare the operator prototype function is in public section.
3. Define the definition of the operator.

Example

```
# include <iostream.h>
# include <conio.h>
class number
{
   public:
        int x,y;
        number()
        {
                x=0;
                y=0;
        }
        number(int a, int b)
        {
                x=a;
                y=b;
        }
        number operator + (number d)
        {
                number t;
                t.x=x+d.x;
                t.y=y+d.y;
                return t;
        }
        void show()
        {
                cout<<"\nX="<<x<<"\t Y="<<y;
        }
};

int main()
{
        number n1(10,20), n2(20,30), n3;
        n3=n1+n2;
        n3.show();
        return 0;
}
```

Output X=10 Y=20 X=20 Y=30 X=30 Y=40

Explain about overloading unary operators with an example.

- An operator whichtakes only one argument is called as unary operator.
  Ex: ++, --, -, +, !, ~, etc.
- A class member function will take zero arguments for overloading unary operator.
- A friend member function will take only one argument for overloading unary operator.

Constraints on increment /decrement operators

When ++ and − − operators are overloaded, there exists no difference between the postfix and prefix overloaded operator functions. To make the distinction between prefix and postfix

notations of operator, a new syntax is used to indicate postfix operator overloading funct ion. The syntaxes are as follows:

Operator ++(int); //postfix notation
Operator ++( ); //prefix notation

Ex: Program for overloading unary operator using normal member function.

```
# include <iostream.h>
# include <conio.h>
class num
{
    private:
        int a, b;
    public:
        num(int x, int y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<"\nA="<<a<<"\tB="<<b;
        }
        void operator ++( )             //prefix notation
        {
                ++a;
                ++b;
        }
        void operator --(int)           //postfix notation
        {
                a--;
                b--;
        }
};

int main()
{
        num x(4,10);
        x.show();
        ++x;
        cout<<"\n After increment";
        x.show();
        x--;
        cout<<"\n After decrement";
        x.show();

}
```

Output

After increment
A = 5 B = 11
After decrement
A = 4 B = 10

Ex: Program for overloading unary operator using friend function.

```
# include <iostream.h>
# include <conio.h>
class complex
{
    private:
        int real, imag;
    public:
        complex()
        {
                real=imag=0;
        }
        complex(int r, int i)
        {
                real=r;
                imag=i;
        }
        void show()
        {
                cout<<"\n real="<<real<<"\timaginary="<<imag;
        }
        friend complex operator -(complex &c)
        {
        =-c.r;
                c.i=-c.i;
                return c;
        }
};
void main()
{
        complex c(1,-2);
           how();
        cout<<"\nAfter sign change";
        -c;
        c.show();
}
```

Output

```
real= 1         imaginary=-2
After sign change
real= -1        imaginary=2
```

Explain about overloading binary operators with an example.

- An operator which takes two arguments is called as binary operator.
  Ex: +, *, /, etc.
- A class member function will take one argument for overloading binary operator.
- A friend function will takes two arguments for overloading binary operator.

Example: Program for overloading binary operator using class member function.

```
# include <conio.h>
# include <iostream.h>
class num
{
    private:
```

```
        int a , b;
    public:
        void input()
        {
                cout<<"\nEnter two numbers:";
                cin>>a>>b;
        }
        void show()
        {
                cout<<"\nA= "<<a<<"\tB=  "<<b;
        }
        num operator +(num n)
        {
                num t;
                t.a=a+n.a;
                t.b=b+n.b;
                return t;
        }
        num operator -(num n)
        {
                num t;
                t.a=a-n.a;
                t.b=b-n.b;
                return t;
        }
};
int main()
{
        num x,y,z;
        x.read();
        y.read();
        z=x+y;
        r.show();
        return 0;
}
```

Output

```
Enter two numbers: 1 2
Enter two numbers: 3 4
A=4     B=6
```

Ex: Program for overloading binary operator using friend function.

```
# include <conio.h>
# include <iostream.h>
class num
{
    private:
        int a, b;
    public:
        void input()
        {
                cout<<"\nEnter two numbers:";
                cin>>a>>b;
        }
        void show()
        {
                cout<<"\nA="<<a<<"\tB="<<b;
        }
        friend num operator + (num o1, num o2)
        {
                num t;
                t.a=o1.a+o2.a;
                t.b=o1.b+o2.b;
                return t;
        }
};
int main()
{
        num x,y,z;
        x.input();
        y.input();
        z=x+y;
        z.show();
        return 0;
}
```

Output

```
Enter two numbers: 5 8
Enter two numbers: 1 4
A=6     B=12
```

Explain about overloading assignment operator with an example.

Data members of one object are initialized with some values, and same values are assigned to another object with assignment operator. Assignment operators can be overloaded in two ways. They are:

1. Implicit overloading
2. Explicit overloading

Implicit overloading:

```
# include <iostream.h>
class num
{
    private:
        int x;
    public:
```

```
        num(int a)
        {
                x=a;
        }
        void show()
        {
                cout<<x<<"  ";
        }
};
int main()
{
        num n1(2), n2(3);
        cout<<"\nBefore assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        a2=a1;          //Implicit assigment
        cout<<"\nAfter assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
}
```

Output

Before assignment:
A = 2 B=3
After assignment:
A = 2 B=2

Explicit overloading:

```
# include <iostream.h>
class num
{
        private:
           int x;
        public:
        num(int a)
        {
                x=a;
        }
        void show()
        {
                cout<<X<<"  ";
        }
        void operator =(num b)
        {
                x=b.x;
        }
};
int main()
{
        num a1(2), a2(3);
```

```
        num n1(2), n2(3);
        cout<<"\nBefore assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        a1.operator=(a2);                    //Explicit assigment
        cout<<"\nAfter assignment:";
        cout<<"\n A=";
        a1.show();
        cout<<"\n B=";
        a2.show();
        return 0;
}
```

Output

Before assignment:
A = 2 B=3
After assignment:
A = 3 B=3

Explain about rules for overloading operators.

1. Operator overloading can't change the basic idea.
2. Operator overloading never changes its natural meaning. An overloaded operator "+" can be used for subtraction of two objects, but this type of code decreased the utility of the program.
3. Only existing operators can be overloaded.
4. The following operators can't be overloaded with class member functions

Operator	Description
.	Member operator
.*	Pointer to member operator
::	Scope resolution operator
sizeof()	Size of operator
# and ##	Preprocessor symbols

5. The following operators can't be overloaded using friend functions.

Operator	Description
( )	Function call operator
=	Assignment operator
[ ]	Subscripting operator
–>	Class member access operator

6. In case of unary operators normal member function requires no parameters and friend function requires one argument.
7. In case of binary operators normal member function requires one argument and friend function requires two arguments.
8. Operator overloading is applicable only within in the scope.
9. There is no limit for the number of overloading for any operation.
10. Overloaded operators have the same syntax as the original operator.

What is inheritance? What are the advantages and disadvantages of inheritance?

(OR)

Explain about reusability.

Inheritance is the most important and useful feature of OOP. Reusability can achieved with the help of inheritance. The mechanism of deriving new class from an old class is called as inheritance. The old class is known as *parent class* or *base class*. The new one is called as *child class* or *derived class*. In addition to that properties new features can also be added.

Advantages:

- Code can be reused.
- The derived class can also extend the properties of base class to generate more dominant objects.
- The same base class is used for more derived classes.
- When a class is derived from more than one class, the derived classes have similar properties to those of base classes.

Disadvantages:

- Complicated.
- Invoking member functions creates overhead to the compiler.
- In class hierarchy, various data elements remains unused, and the memory allocated to them is not utilized.

What are access specifiers? How class are inherited?

C++ provides three different access specifiers. They are:

1. Public
2. private and
3. protected.

- The public data members can be accessed directly outside ofthe class with the object.
- The private members are accessed by the public member functions of the class.
- The protected members are same as private but only the difference is protected members are inherited while private members are not inherited.

A new class can be derived from the old one is as follows:

```
class name_of_the_derived_class : access_specifier name_of_the_base_class
{
        ……
        Members of the derived class
        ......
};
```

Example:

```
class A : public B
{
        ..............
        ..............
};

class A : private B
{
..............
..............
```

};
class A : protected B
{
.....
.....
};

Note: If no access specifier is specified then by default it will takes as private.

Type of inheritance	Base class member	Derived class member
Private	Private	Not accessible
	Public	Private
	Protected	Private
Public	Private	Not accessible
	Public	public
	Protected	Protected
Protected	Private	Not accessible
	Public	Protected
	Protected	Protected

Public derivation:
In public derivation, all the public members of base class become public members of the derived class and protected members of the base class becomes protected members to the derived class. Private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};
class Derived : public Base
{
        public: int y;
};
int main()
{
        Derived d;
        d.x=10;
        d.y=20;
        cout<<"\n Member x="<<b.x;
        cout<<"\n Member y="<<b.y;
        return 0;

}
```

Output:
Member x=10
Member y=20

Private derivation

In private derivation, all the public and protected members of the base class become private members of the derived class and private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};

class Derived : private Base
{
        int y;
    public:
        Derived( )
        {
                x=10;
                y=20;
        }
        void show()
        {
                cout<<"\n Member x="<<x;
                cout<<"\n member y="<<y;
        }
};
int main()
{
        Derived d;
        d.show():
        return 0;
}
```

Output:

```
Member x=10
Member y=20
```

Protected derivation

In protected derivation, all the public and protected members of the base class become protected members of the derived class and private members of the base class will not be inherited.

Example:

```
# include <iostream.h>
class Base
{
        public: int x;
};

class Derived : protected Base
{
        int y;
    public:
```

```
        Derived( )
        {
                x=10;
                y=20;
        }
        void show( )
        {
                cout<<"\n Member x="<<x;
                cout<<"\n member y="<<y;
        }
};

int main()
{
        Derived d;
        d.show();
}
```

Output:
Member x=10
Member y=20

Explain about protected data with private members with an example.
The protected type is similar to private, but it allows the derived class to access the members of the protected.

Example

```
# include <iostream.h>
class Base
{
        protected: int x;
};

class Derived : private Base
{
    int y;
        public:
          Derived( )
          {
                x=10;
                y=20;
          }
          void show()
          {
                cout<<"\n Member x="<<x;
                cout<<"\n Member y="<<y;
          }
};
int main()
{
        Derived d;
        d.show();
}
```

Output:
Member x=10
Member y=20

Define inheritance. Explain about various types of inheritance with examples.
Deriving a new class from an existing one is called as inheritance.
Types:

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

Single Inheritance:
In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance..

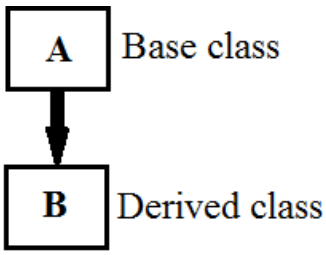


Fig. Single Inheritance

Here A is the base class and B is the derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Emp
{
  public:
    int eno;
    char ename[20],desig[20];
    void input()
    {
          cout<<"Enter employee no:";
          cin>>eno;
          cout<<"Enter employee name:";
          cin>>ename;
          cout<<"Enter designation:";
          cin>>desig;
    }
};

class Salary: public Emp
{
    float bp, hra, da, pf, np;
  public:
    void input1()
    {
          cout<<"Enter Basic pay:";
          cin>>bp;
          cout<<"Enter House Rent Allowance: ";
          cin>>hra;
          cout<<"Enter Dearness Allowance :";
          cin>>da;
          cout<<"Enter Provident Fund:";
          cin>>pf;
    }
    void calculate()
    {
          np=bp+hra+da-pf;
    }
    void display()
    {
          cout<<"\nEmp no: "<<eno
          cout<<"\nEmp name: "<<ename;
          cout<<"\nDesignation: "<<design;
          cout<<"\nBasic pay:"<<bp;
          cout<<"\nHRA:"<<hra;
          cout<<"\nDA:"<<da;
          cout<<"\nPF:"<<pf;
          cout<<"\nNet pay:"<<np;
    }
```

```
};

int main()
{
   clrscr();
   Salary s;
   s.input();
   s.input1();
   s.calculate();
   s.show();
   getch();
   return 0;
}
```

Output:

```
Enter employee number: 1001
Enter employee name: Vijayanand
Enter designation: Manager
Enter basic pay:25000
Enter House Rent Allowance:2500
Enter Dearness Allowance :5000
Enter Provident Fund:1200

Emp no: 1001
Emp name: Vijayanand
Designation: Manager
Basic pay:25000
HRA:2500
DA:5000
PF:1200
Net pay: 31300
```

Multiple Inheritance:

In this type of inheritance a class may derive from two or more base classes.

(or)

When a class is derived from more than one base class, is called as multiple inheritance.

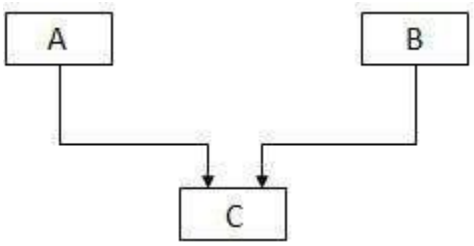


Fig. Multiple Inheritance

Where class *A* and *B* are Base classes and *C* is derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Student
{
      protected:
        int rno,m1,m2;
      public:
        void input()
        {
                cout<<"Enter the Roll no :";
                cin>>rno;
                cout<<"Enter the two subject marks :";
                cin>>m1>>m2;
        }
};

class Sports
{
     protected:
        int sm;                    // sm = Sports mark
     public:
```

```
        void getsm()
        {
            cout<<"\nEnter the sports mark :";
            cin>>sm;
        }
};
class Report : public student, public sports
{
        int tot, avg;
    public:
        void show()
        {
            tot=(m1+m2+sm);
            avg=tot/3;
            cout<<"\nRoll No : "<<rno<<"\nTotal : "<<tot;
            cout<<"\n\tAverage : "<<avg;
        }
};

int main()
{
        clrscr();
        Report r;
        r.input();
        r.getsm();
        r.show();
        return 0;
}
```

Output:
Enter the roll no: 10
Enter the two marks : 70 90
Enter the sports mark: 60
Roll no : 10
Total : 110
Average: 73

Hierarchical Inheritance
In this type of inheritance, multiple classes are derived from a single base class.

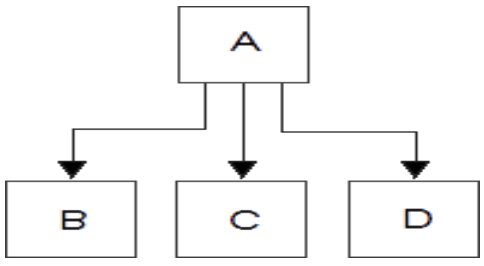


Fig. Hierachical Inheritance

Where class *A* is the base class and *B, C* and *D* are derived classes.
Example:

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

```
class Vehicle
{
    public:
        Vehicle()
        {
            cout<<"\nIt is motor vehicle";
        }
};
class TwoWheelers : public Vehicle
{
    public:
        TwoWheelers()
        {
            cout<<"\nIt has two wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 80 kmph";
        }
};
class ThreeWheelers : public Vehicle
{
    public:
        ThreeWheelers()
        {
            cout<<"\nIt has three wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 60 kmph";
        }
};
class FourWheelers : public Vehicle
{
    public:
        FourWheelers()
        {
            cout<<"\nIt has four wheels";
        }
        void speed()
        {
                cout<<"\nSpeed: 120 kmph";
        }
};

int main( )
{
        clrscr();
        TwoWheelers two;
        two.speed();
        cout<<"\n--------------------------";
        ThreeWheelers three;
        three.speed();
```

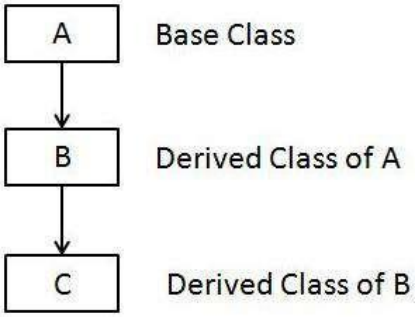
```
        cout<<"\n--------------------------";
        FourWheelers four;
        four.speed();
        getch();
        return 0;
}
```

Output

```
It is motor vehicle
It has two wheels
Speed: 80 kmph";
-----------------------------
It is motor vehicle
It has three wheels
Speed: 60 kmph";
-----------------------------
It is motor vehicle
It has four wheels
Speed: 120 kmph";
```

Multilevel Inheritance

In this type of inheritance, the derived class inherits from a class, which in turn inherits from some other class.



Where *class A* is the base class, *C* is derived class and *B* acted as base class as well as derived class.

Example:

```
#include<iostream.h>
#include<conio.h>
class Car
{
    public:
        Car()
        {
            cout<<"\nVehicle type: Car";
        }
};
class Maruti : public Car
{
    public:
        Maruti()
        {
           cout<<"\nComnay: Maruti";          }
        }
        void speed()
        {
```

```
                cout<<"\nSpeed: 90 kmph";
        }
};
class Maruti800 : public Maruti
{
    public Maruti800()
   {
       cout<<"\nModel: Maruti 800");
   }
   void speed()
   {
        cout<<"\nSpeed: 120 kmph";
   }
};
int main( )
{
        clrscr();
        Maruti800 m;
        m.speed();
        getch();
}
```

Output:
Vehicle type: Car
Company: Maruti
Model: Maruti 800
Speed: 120K mph

Hybrid (Virtual) Inheritance
Hybrid Inheritance is combination of one or more types of inheritance.

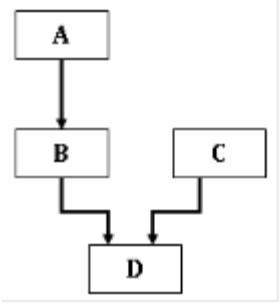


Fig. Hybrid Inheritance

Where class A to class B forms single inheritance, and class B,C to class D form Multiple inheritance.

Example:

```
#include<iostream.h>
#include<conio.h>
class Player
{
    protected:
        char name[20];
        char gender;
        int age;
};

class Physique : public Player
{
    protedted:
        float height,weight;
};
class Location
{
     protected:
        char city[15];
        long int pin;
};
class   Game    : public Physique, public
Location
```

```
{
        char game[15];
    public:
        void input()
         {
            cout<<"\nEnter Player Information";
            cout<<"Name: ";
            cin>>name;
            cout<<"Genger: ";
            cin>>gender;
            cout<<"Age: ";
            cin>>age;
            cout<<"Height and Weight: ";
            cin>>heigh>>weight;
            cout<<"City: ";
            cin>>city;
            cout<<"Pincode: ";
            cin>>pin;
            cout<<"Game played: ";
            cin>>game;

     }
     void input()
     {
            cout<<"\nPlayer Information";
            cout<<"\nName: "<<name;
            cout<<"\nGenger: "<<gender;
            cout<<"\nAge: "<<age;
            cout<<"\nHeight<<height;
            cout<<"\nWeight: "<<weight;
            cout<<"\nCity: "<<city;
            cout<<"\nPincode: "<<pin;
            cout<<"\nGame: "<<game;
      }
};
int main( )
{
        clrscr();
        Game  g;
        g.input();
        g.show();
        return 0;
}
```

Output

```
Enter Player Information
Name: Azar
Genger: M
Age: 38
Height and Weight: 5.8 70
City: Hyderabad
Pincode: 522183
Game played: Cricket

Player Information
Name: Azar
Genger: M
Age: 38
Height and Weight: 5.8 70
City: Hyderabad
Pincode: 522183
Game played: Cricket
```

Multi-path Inheritance

In this, one class is derived from two base classes and in turn these two classes are derived from a single base class in known as Multi-path Inheritance**.**

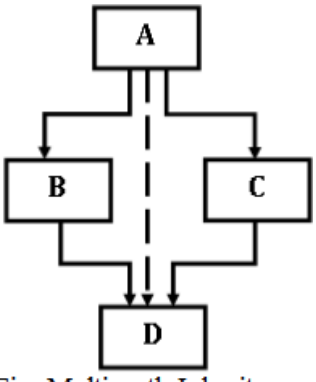


Fig. Multi-path Inheritance

Example

```
class A
{
        //class A definition
};
class B: public A
{
        //class B definition
};
class C: public A
{
        //class C definition
};
class D :public B, public  C
{
        //class D definition
```

```
};
```

Explain about virtual base classes with an example.

(OR)

How can you overcome the ambiguity occurring due to multipath inheritance? Explain with an example.

To overcome the ambiguity due to multipath inheritance the keyword virtual is used. When classes
are derived as virtual, the compiler takes essential caution to avoid the duplication of members.

Uses:

When two or more classes are derived from a common base class, we can prevent multiple copies of
the base class in the derived classes are done by using **virtual** keyword. This can be achieved by
preceding the keyword "**virtual"** to the base class.

Example

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

class A
{
        protected:
                int a1;
};
class B: public virtual A
{
        protected:
                int a2;

};
class C: public virtual A
{
        protected:
                int a3;

};
class D :public B, public C
{
        int a4;
    public:
        void input()
        {
            cout<<"Enter a1,a2,a3 and a4 values:";
            cin>> a1>>a2>>a3>>a4;
        }
        void show()
        {
            cout<<"a1="<<a1<<"\na2="<<a2;
            cout<<"\na3="<<a3<<"\na4="<<a4;
        }
```

```
};
int main()
{
    D d;
    d.input();
    d.show();
    return 0;
}
```

Output

```
Enter a1, a2, a3 and a4 values: 10 20 30 40
a1=10
a2=20
a3=30
a4=40
```

How constructors and destructors are executed in inherited class? Explain with an example.

The constructors are used to initialize the member variables and the destructors are used to destroy the object. The compiler automatically invokes the constructor and destructors.

Rules:

- The derived class does not require a constructor if the base class contains default constructor.
- If the base class is having a parameterized constructor, then it is necessary to declare a constructor in derived class also. The derived class constructor passes arguments to the base class constructor.

Example:

```
#include<iostream.h>
class A
{
    public:
        A()
        {
                cout<<"\n Class A constructor called";
        }
        ~A()
        {
                cout<<"\nClass A destructor called";
        }
};

class B : public A
{
    public:
        B()
        {
                cout<<"\n Class B constructor called";
```

```
        }
        ~B()
        {
                cout<<"\nClass B destructor called";
        }
};

class C : public B
{
    public:
        C()
        {
                cout<<"\n Class C constructor called";
        }
        ~C()
        {
                cout<<"\nClass C destructor called";
        }
};

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

Output

```
Class A constructor called
Class B constructor called
Class C constructor called
Class C destructor called
Class B destructor called
Class A destructor called
```

How can you pass an object as a class member? Explain.

(OR)

Explain about delegation with an example.

(OR)

Explain about container classes with an example.

Declaring the object as a class data member in another class is known as delegat ion. When a class has an object of another class as its member, such a class is known as a container class. This kind of relationship is known as has-a-relationship or containership.

Example

```
# include <iostream.h>
class A
{
     public:
        int x;
        A()
        {
                x=20;
                cout<<"\n In A constructor";
```

```
        }
};

class B
{
    public:
        int y;
        A a;
        B()
        {
                y=30;
                cout<<"\n In B constructor";
        }
        void show()
        {
                cout<<"\n X="<<a.x<<"\t Y="<<y;
        }
};
int main()
{
        B b;
        b.show();
        return 0;
}
```

Output

```
In A constructor
In B constructor
X=20 Y=30
```

Define abstract class. What is the use of abstract classes? Explain.

An abstract class is a class not used for creating objects. It is designed only to act as a base class. These classes are similar to a skeleton on which new classes are designed. These classes contain pure virtual functions.

Example

```
#include <iostream.h>
class Shape
{
   protected:
        int width;
        int height;
  public:
        virtual int getArea() = 0;
        void setWidth(int w)
        {
                width = w;
        }
        void setHeight(int h)
        {
                height = h;
        }
 };
```

```
class Rectangle: public Shape
{
  public:
        int getArea()
        {
                return (width * height);
        }
};
class Triangle: public Shape
{
    public:
        int getArea()
        {
                return (width * height)/2;
        }
};

void main()
{
        Rectangle r;
        Triangle t;
        r.setWidth(5);
        r.setHeight(7);
        cout << "\nRectangle area: " << r.getArea() << endl;
        t.setWidth(5);
        t.setHeight(7);
        cout << "\nTriangle area: " << t.getArea() << endl;
      return 0;
}
```

Output:
Rectangle area : 35
Triangle area : 17