CHAPTER 4

POLYMORPHISM

Er. Ganga Gautam

OUTLINES

1. Introduction to Polymorphism
2. Types of Polymorphism: Compile Time Polymorphism: Function Overloading, Operator Overloading Runtime Polymorphism: Virtual Function
3. Overloading Unary and Binary Operators
4. Function Overriding
5. this Pointer and Object Pointer
6. Pure Virtual Function, Abstract Class
7. Virtual Destructor
8. Type Conversion: Basic to User-Defined, User-Defined to Basic, User-defined to User-Defined

5.1 Polymorphism

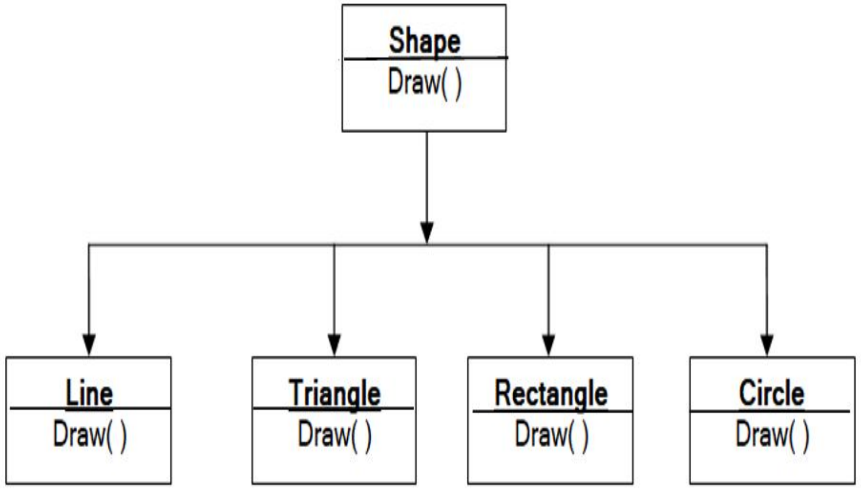
- Greek words: ***poly*** means *many* and ***morph*** means *forms.*
- It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios.

Parent
Traveler
Customer
Polymorphism
Officer
Husband
Parent

5.1 Polymorphism (contd.)

- Polymorphism is an important concept of object-oriented programming.
- The + operator in C++ is used to perform two specific functions.
- When it is used with numbers (integers and floating-point numbers), it performs addition.
- And when we use the + operator with strings, it performs string concatenation.
- Polymorphism is the concept with the help of which single action in different ways can be performed.

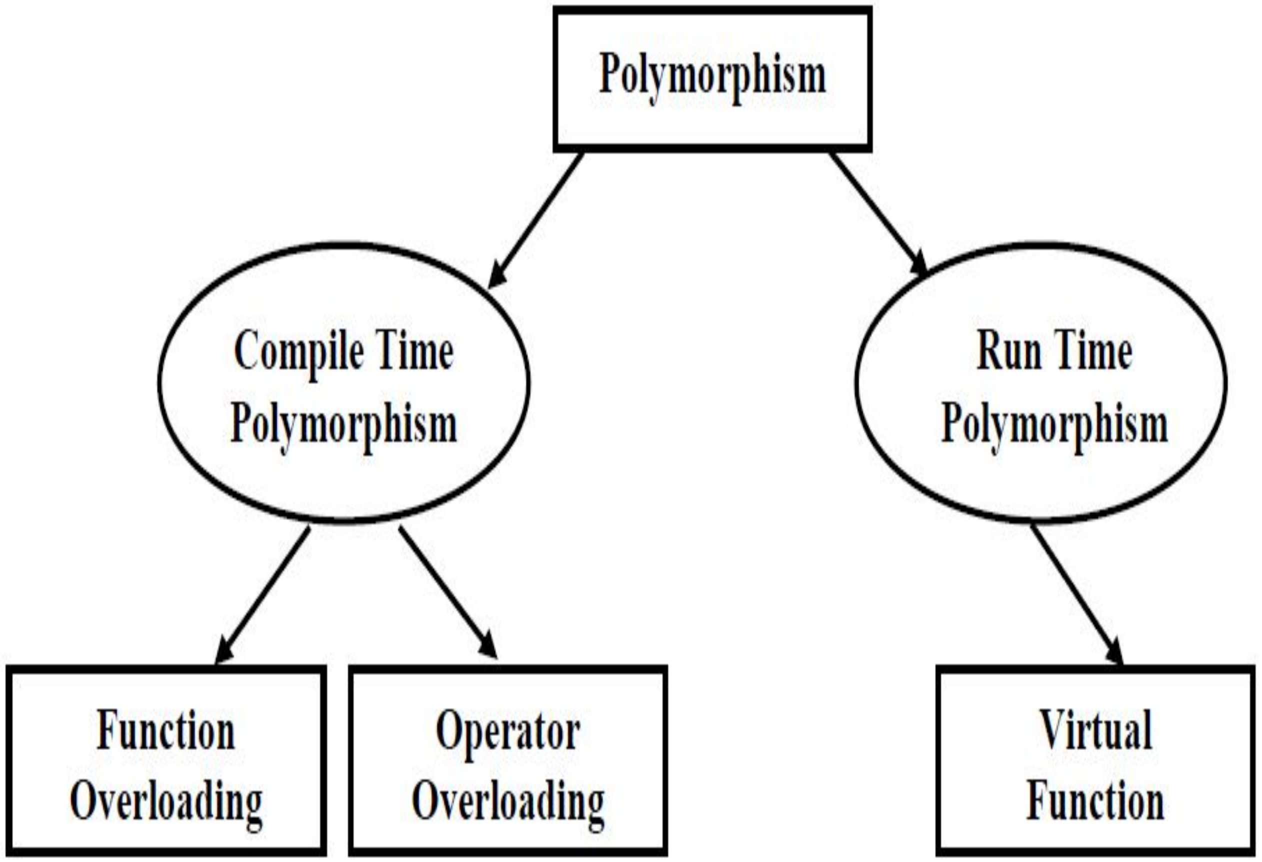
- In OOP, polymorphism refers to identically named function members that have different behavior depending on the type of object they have.
- Basically, we can define it as the ability for derived classes to redefine methods.



Ways of implementing polymorphism:

1. Function overloading
2. Operator overloading
3. Function overriding
4. Virtual functions

5.2. Types of polymorphism



5.2. Types of polymorphism (contd.)

1. Compile-Time Polymorphism:

- Choosing member function in a normal way during compilation time is called Compile-time polymorphism.
- implemented using the overloaded functions and operators.
- The overloaded member functions are 'selected' for invoking by matching arguments, both type and number.
- This information is known to the compiler at the compile time and, therefore compiler is able to select the appropriate function for a particular call at the compile time itself.
- This is called *early binding or static binding or static linking.*
- Early binding simply means that an object is bound to its function call at compile time.

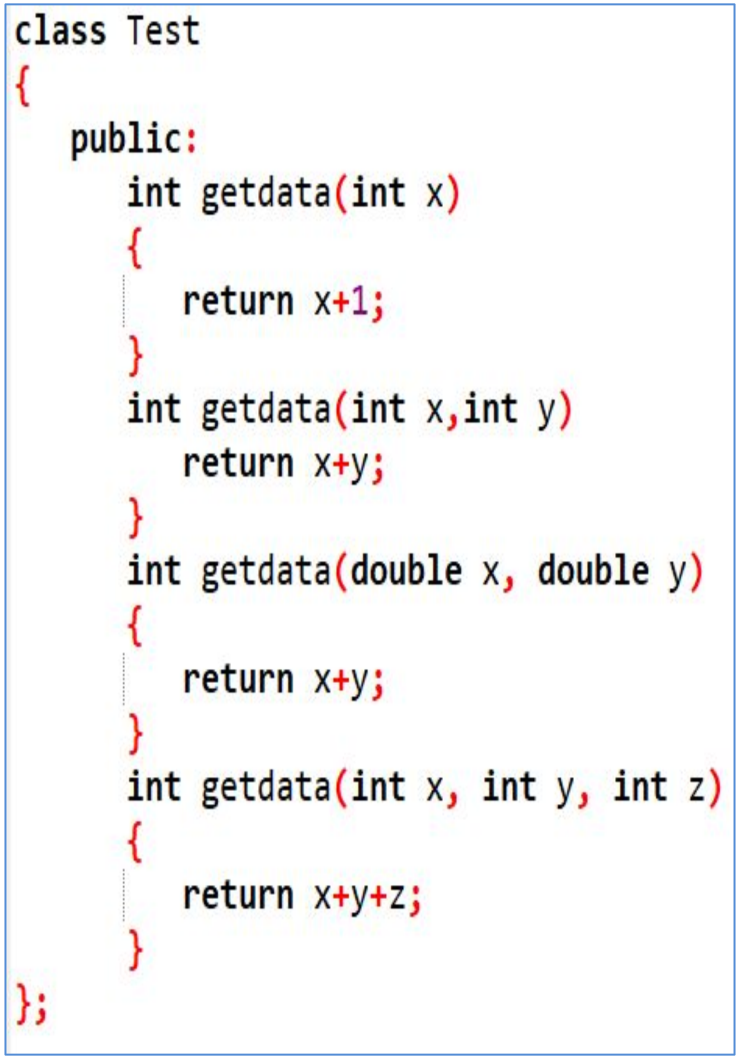
5.2. Types of polymorphism

2. Run-Time Polymorphism:

- Choosing member function during execution time is called Run-time polymorphism.
- The compiler does not know about the correct form of the function to be called.
- The correct form of the function is selected to call on the basis of content of calling object at run time.
- Also called Late-binding or Dynamic Binding or Dynamic Linkage.
- C++ supports a mechanism known as virtual function to achieve runtime polymorphism.

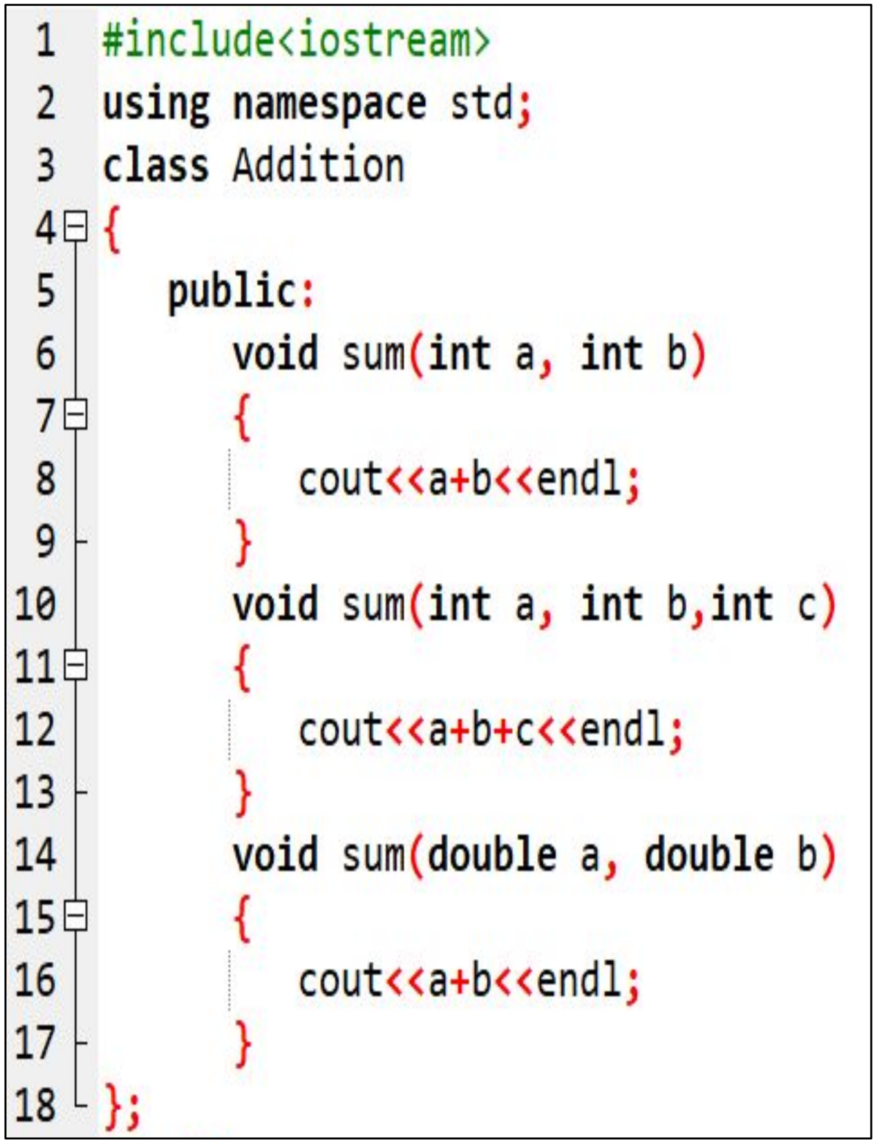
5.3 Function Overloading

- Function Overloading is defined as the process of having two or more functions within a class having same name, but different in parameters.
- Function overloading is done in two ways:
  - By changing number of arguments or parameter.
  - By changing the data type

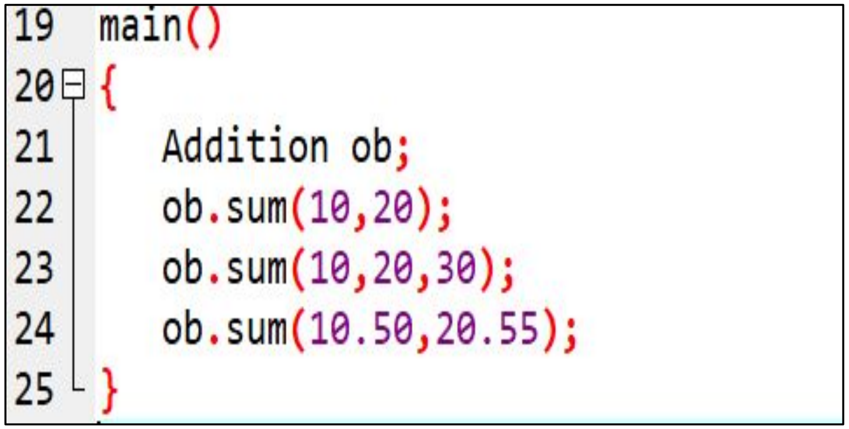


```
class Test
{
    public:
        int getdata(int x)
        {
            return x+1;
        }
        int getdata(int x,int y)
            return x+y;
        }
        int getdata(double x, double y)
        {
            return x+y;
        }
        int getdata(int x, int y, int z)
        {
            return x+y+z;
        }
};
```

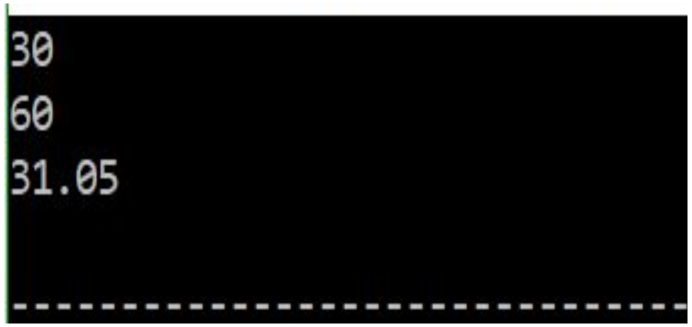
Function Overloading : Sample program



```cpp
#include<iostream>
using namespace std;
class Addition
{
    public:
        void sum(int a, int b)
        {
            cout<<a+b<<endl;
        }
        void sum(int a, int b,int c)
        {
            cout<<a+b+c<<endl;
        }
        void sum(double a, double b)
        {
            cout<<a+b<<endl;
        }
};
```



```cpp
main()
{
    Addition ob;
    ob.sum(10,20);
    ob.sum(10,20,30);
    ob.sum(10.50,20.55);
}
```



```
30
60
31.05

--------------------------------
```

Operator Overloading

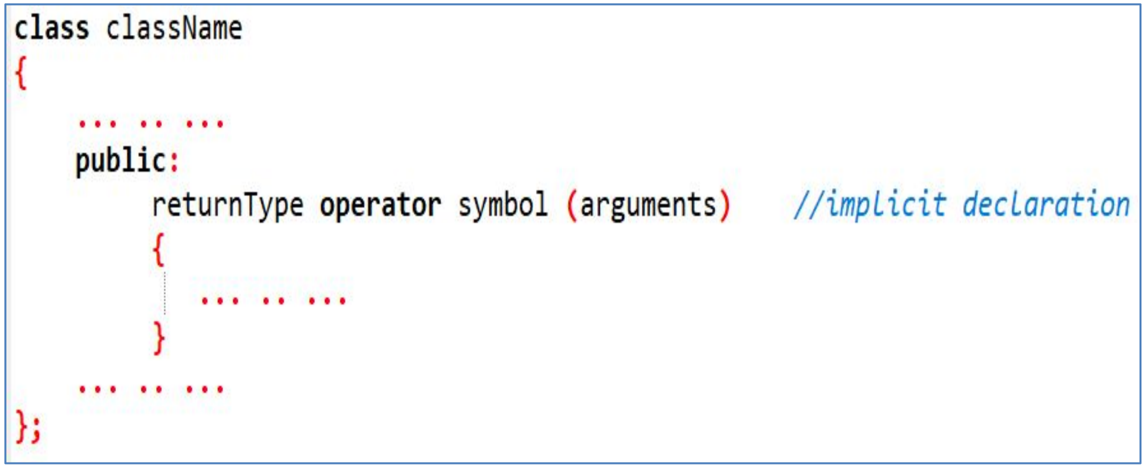
- Operator overloading refers to the use of an operator for different purposes in different data types.
- To add two integers, + operator is used.
- However, for user-defined types (like objects), we can redefine the way operator works.
- For example: If there are two objects of a class that contains string as its data members, we can redefine the meaning of + operator and use it to concatenate those strings.
- The operator to be overloaded is defined using the ***operator*** keyword.
- Below is the list of operators that are not overloaded:

Scope resolution operator	**::**
Size of operator	**sizeof**
Dot operator	**.**
Pointer selector	*****
Ternary operator	**?:**

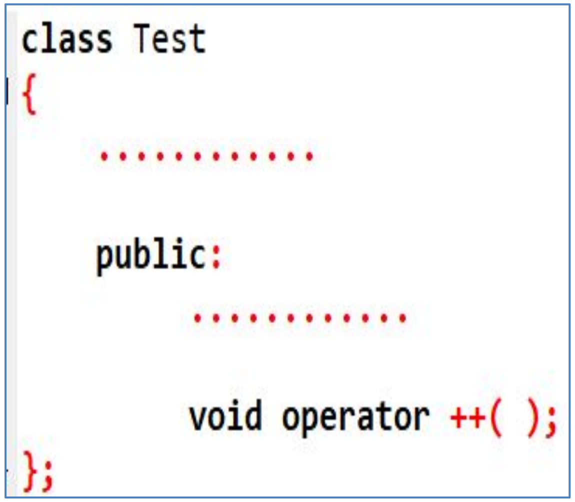
Operator Overloading (Contd.)

- To overload an operator, a special operator function is defined inside the class (Implicit declaration) as:

- Eg:



```
class className
{
    ... .. ...
    public:
        returnType operator symbol (arguments)    //implicit declaration
        {
            ... .. ...
        }
    ... .. ...
};
```

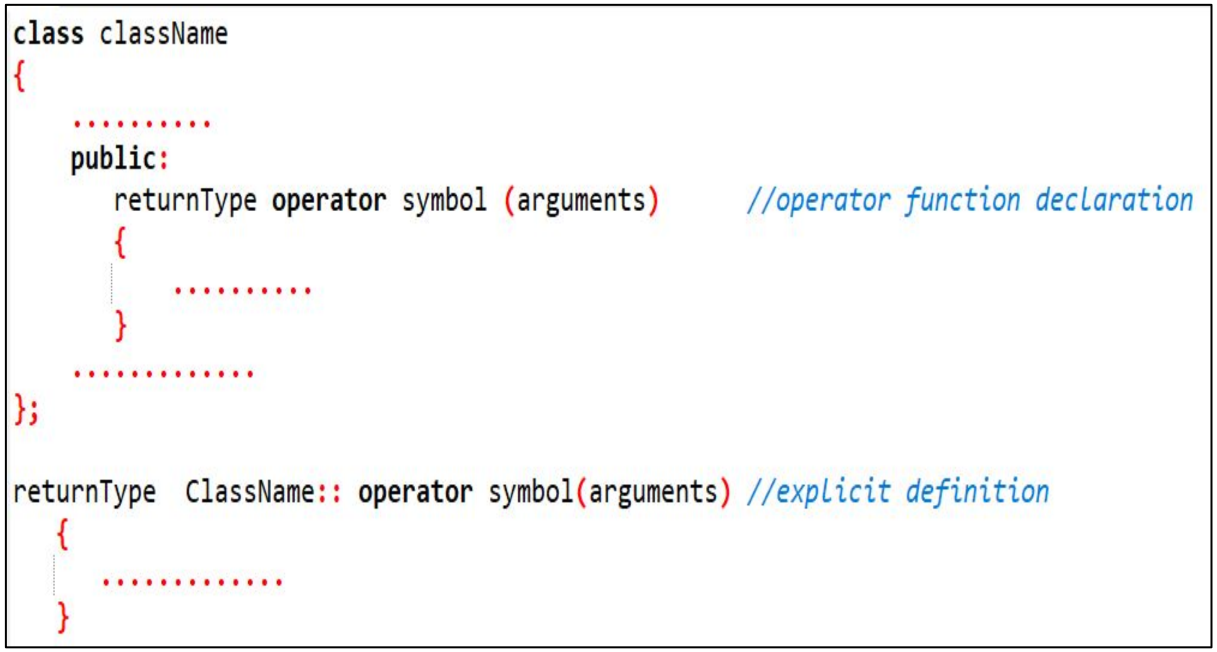


```
class Test
{
    ............

    public:
        ............

            void operator ++( );
};
```

-
  - ***Operator*** is the function name for operator overloading.
  - ***symbol*** is the operator that is being overloaded (i.e. + , ++, etc)
  - ***arguments list*** receives the values.

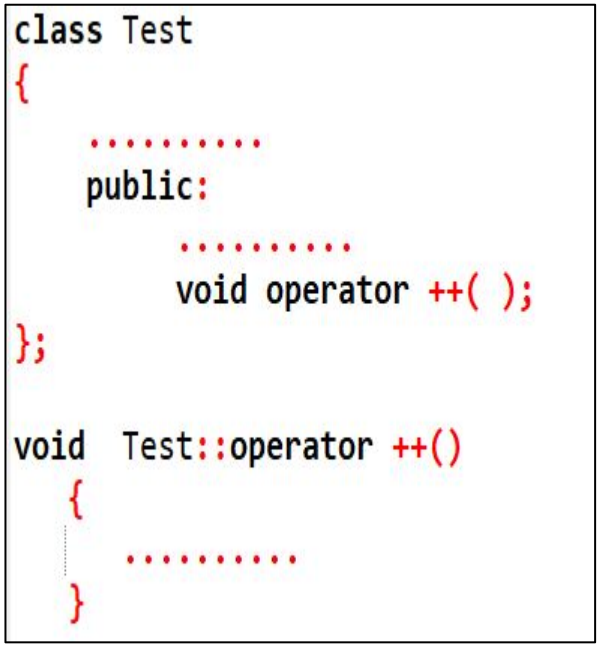
5.4 Operator Overloading (Contd.)

- If defined outside the class (Explicit declaration), the general form is as:



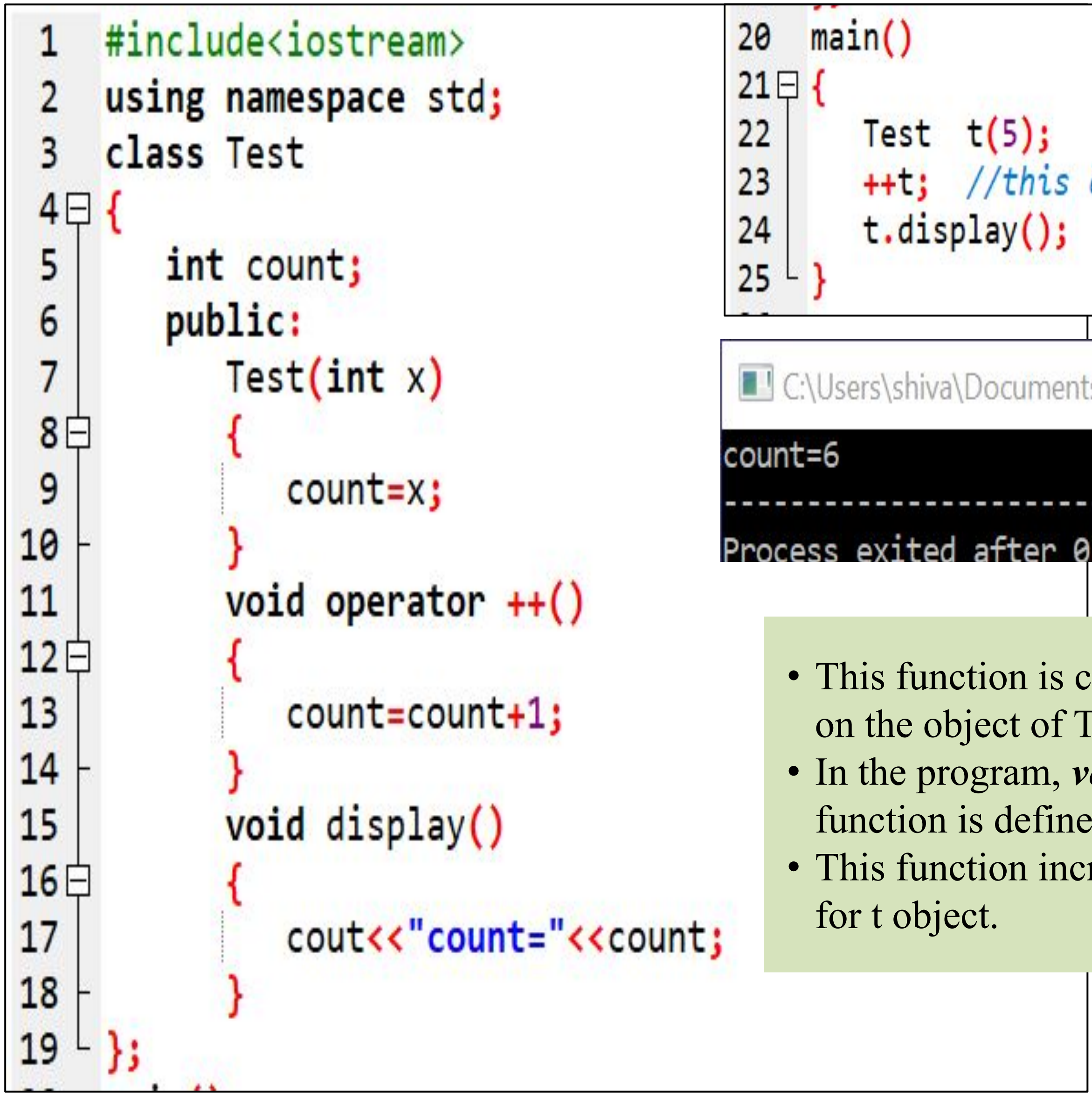
```
class className
{
    ..........
    public:
       returnType operator symbol (arguments)       //operator function declaration
       {
           ..........
       }
    .............
};

returnType  ClassName:: operator symbol(arguments) //explicit definition
   {
       .............
   }
```

- Eg:

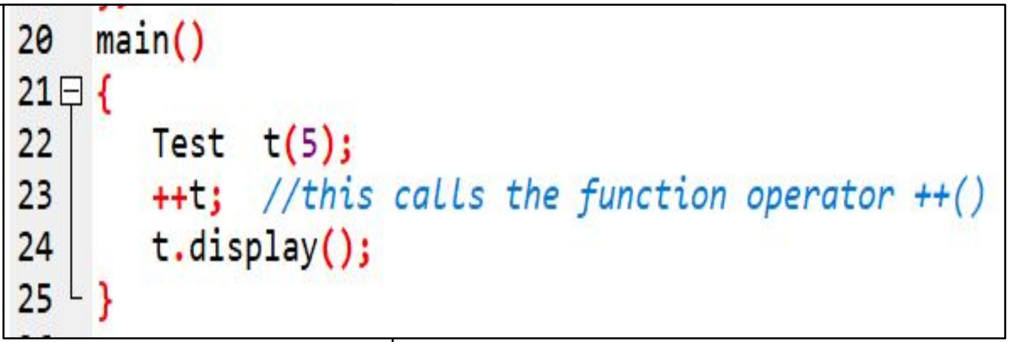


```
class Test
{
     ..........
     public:
          ..........
          void operator ++( );
};

void  Test::operator ++()
   {
       ..........
   }
```

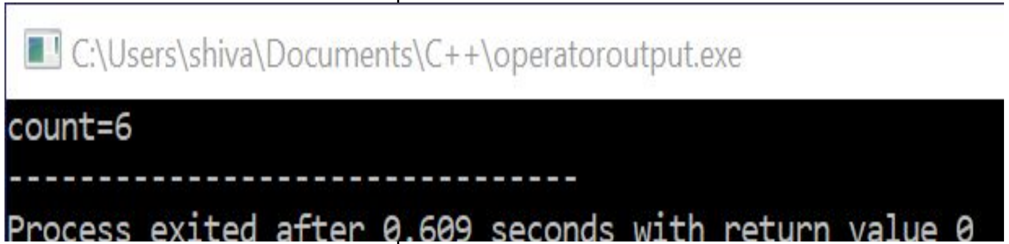
Sample program 1



```cpp
#include<iostream>
using namespace std;
class Test
{
    int count;
    public:
       Test(int x)
       {
          count=x;
       }
       void operator ++()
       {
          count=count+1;
       }
       void display()
       {
          cout<<"count="<<count;
       }
};
```



```cpp
main()
{
    Test  t(5);
    ++t;  //this calls the function operator ++()
    t.display();
}
```



C:\Users\shiva\Documents\C++\operatoroutput.exe

```
count=6
--------------------------------
Process exited after 0.609 seconds with return value 0
```

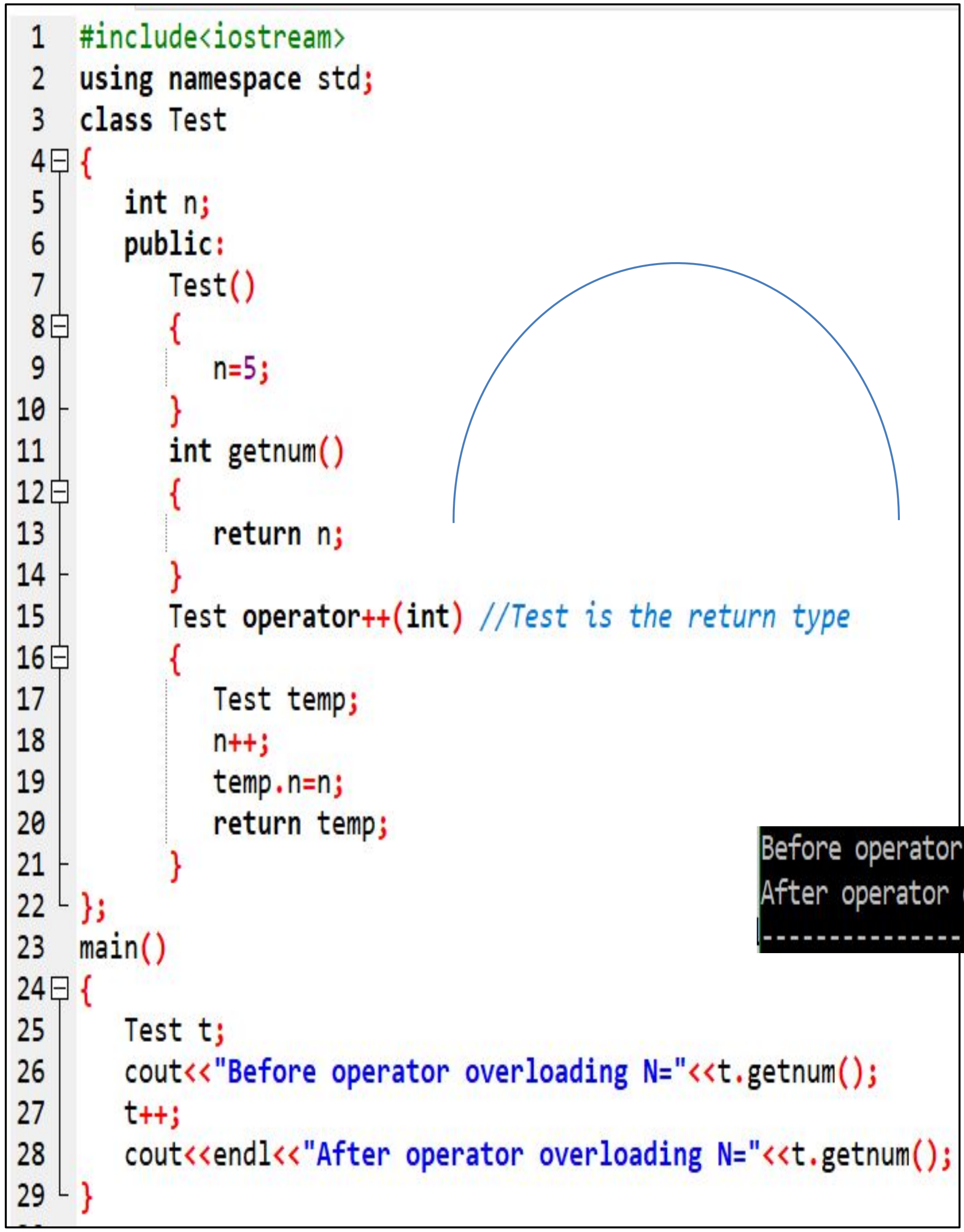
- This function is called when ++ operator operates on the object of Test class (object **t** in this case).
- In the program, ***void operator*** ++ ***()*** operator function is defined (inside Test class).
- This function increments the value of count by 1 for t object.

5.4.1 Operator arguments and ReturnType

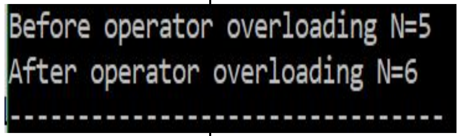
Like any other functions, an overloaded operator has a return type and a parameter list.

Return Type:

- The return type of operator function is always a class type,
- because the operator overloading is only for objects.



```cpp
#include<iostream>
using namespace std;
class Test
{
    int n;
    public:
      Test()
      {
          n=5;
      }
      int getnum()
      {
          return n;
      }
      Test operator++(int) //Test is the return type
      {
          Test temp;
          n++;
          temp.n=n;
          return temp;
      }
};
main()
{
    Test t;
    cout<<"Before operator overloading N="<<t.getnum();
    t++;
    cout<<endl<<"After operator overloading N="<<t.getnum();
}
```



```
Before operator overloading N=5
After operator overloading N=6
--------------------------------
```

5.4.1 Operator arguments

- Operator function should be:
  - either member function or
  - a friend function.

- An operator function, as a member function, requires:
  - one argument for binary operators and
  - no argument for unary operator.
- When operator function (as member function) is called,
  - the calling object is passed implicitly to the function and hence available for member function.

- An operator function, as a friend function, require:
  - two arguments for binary operators and
  - one argument for unary operator.
- When operator function (as friend function) is called,
  - it is essential to pass the objects by value or reference.

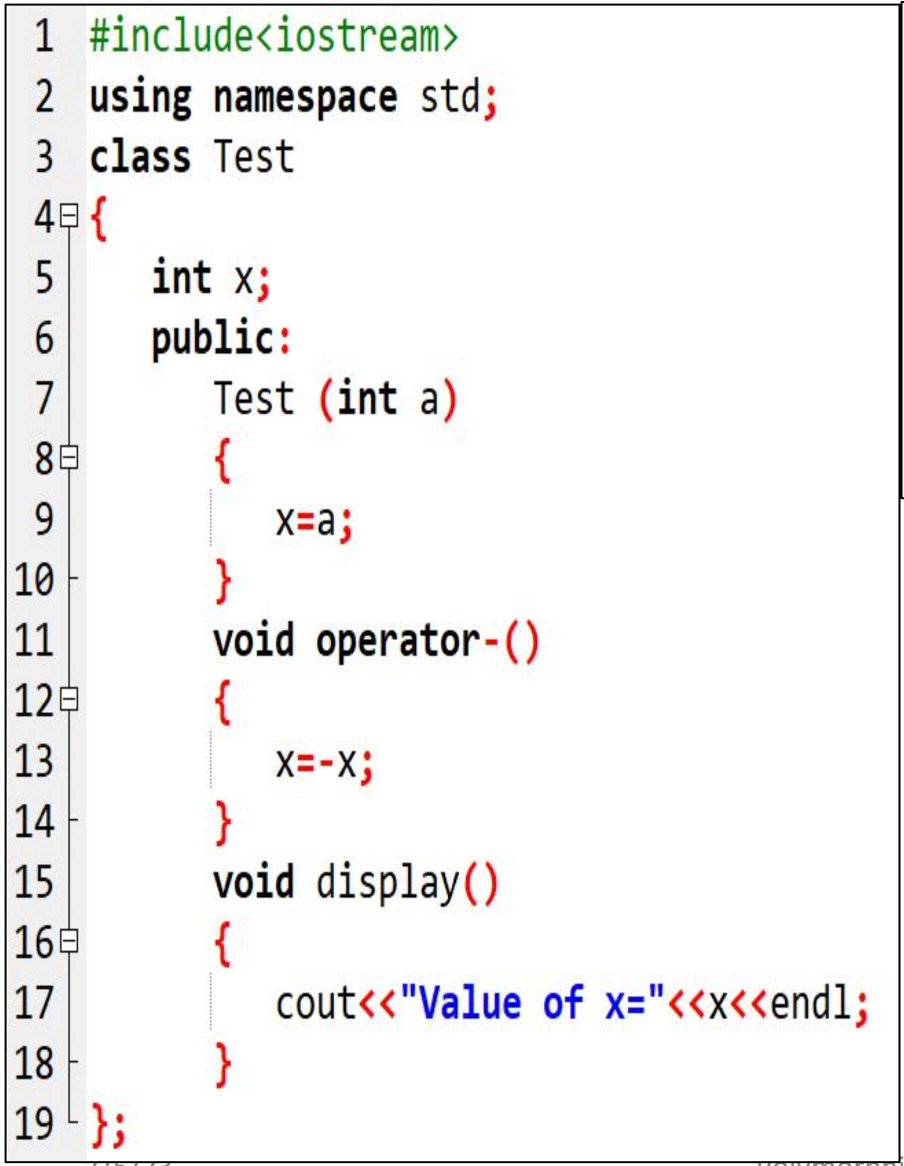
5.4.2 Types of Operator overloading

i. Unary operator overloading
ii. Binary operator overloading

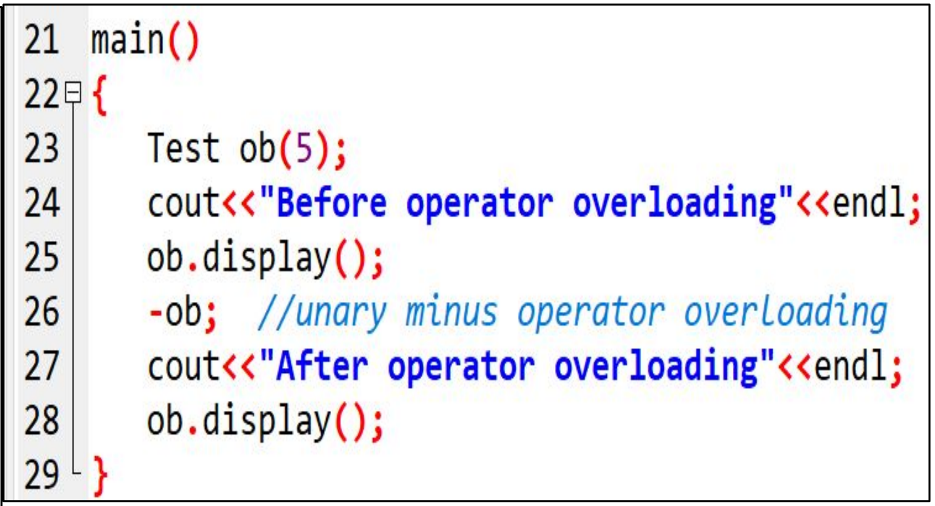
i) Unary Operator overloading

- The operator that requires only one operand to perform its operation are known as unary operators.
- Examples of Unary operators:
  - unary minus (-),
  - unary plus (+),
  - increment (++),
  - decrement(--),
  - logical not (!), etc.
- The operations in which unary operators are overloaded are known as Unary operator overloading.

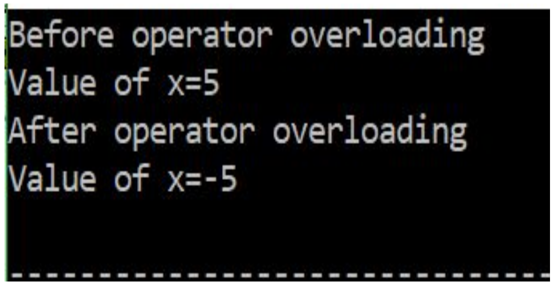
Sample program 1: WAP to overload unary minus (-).



```
#include<iostream>
using namespace std;
class Test
{
    int x;
    public:
        Test (int a)
        {
            x=a;
        }
        void operator-()
        {
            x=-x;
        }
        void display()
        {
            cout<<"Value of x="<<x<<endl;
        }
};
```



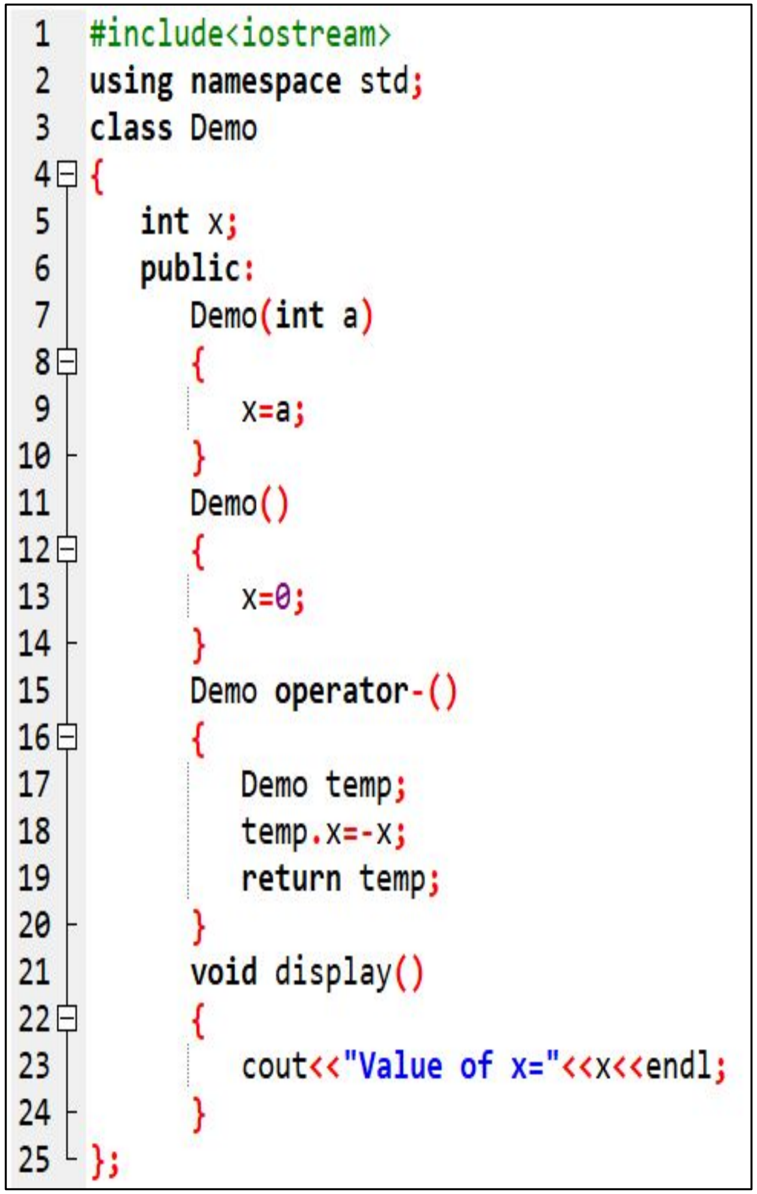
```
main()
{
    Test ob(5);
    cout<<"Before operator overloading"<<endl;
    ob.display();
    -ob;  //unary minus operator overloading
    cout<<"After operator overloading"<<endl;
    ob.display();
}
```



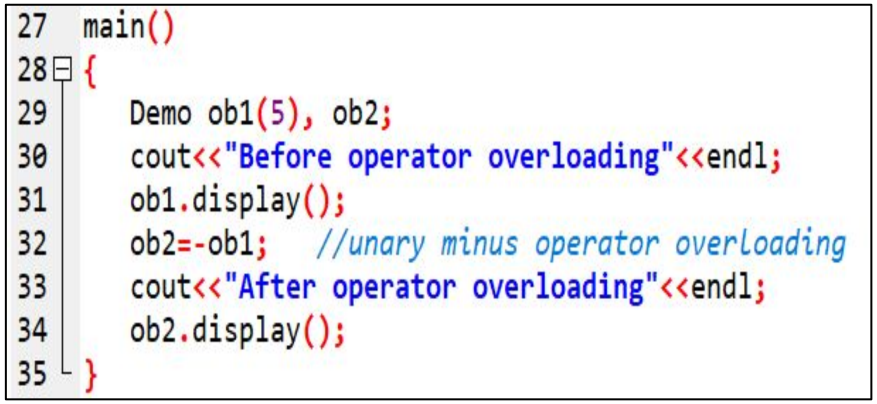
```
Before operator overloading
Value of x=5
After operator overloading
Value of x=-5
```

- In the above program, the minus (-) operator is overloaded.
- This operator is used to make negative value of built-in datatype *x*, and
- the same minus (-) operator is also applied to user-defined datatype *ob* which calls the function operator-().

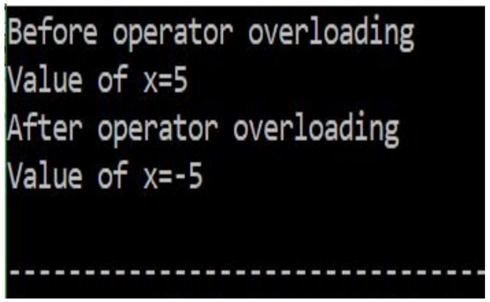
The same above program can be written as:



```
#include<iostream>
using namespace std;
class Demo
{
    int x;
    public:
        Demo(int a)
        {
            x=a;
        }
        Demo()
        {
            x=0;
        }
        Demo operator-()
        {
            Demo temp;
            temp.x=-x;
            return temp;
        }
        void display()
        {
            cout<<"Value of x="<<x<<endl;
        }
};
```

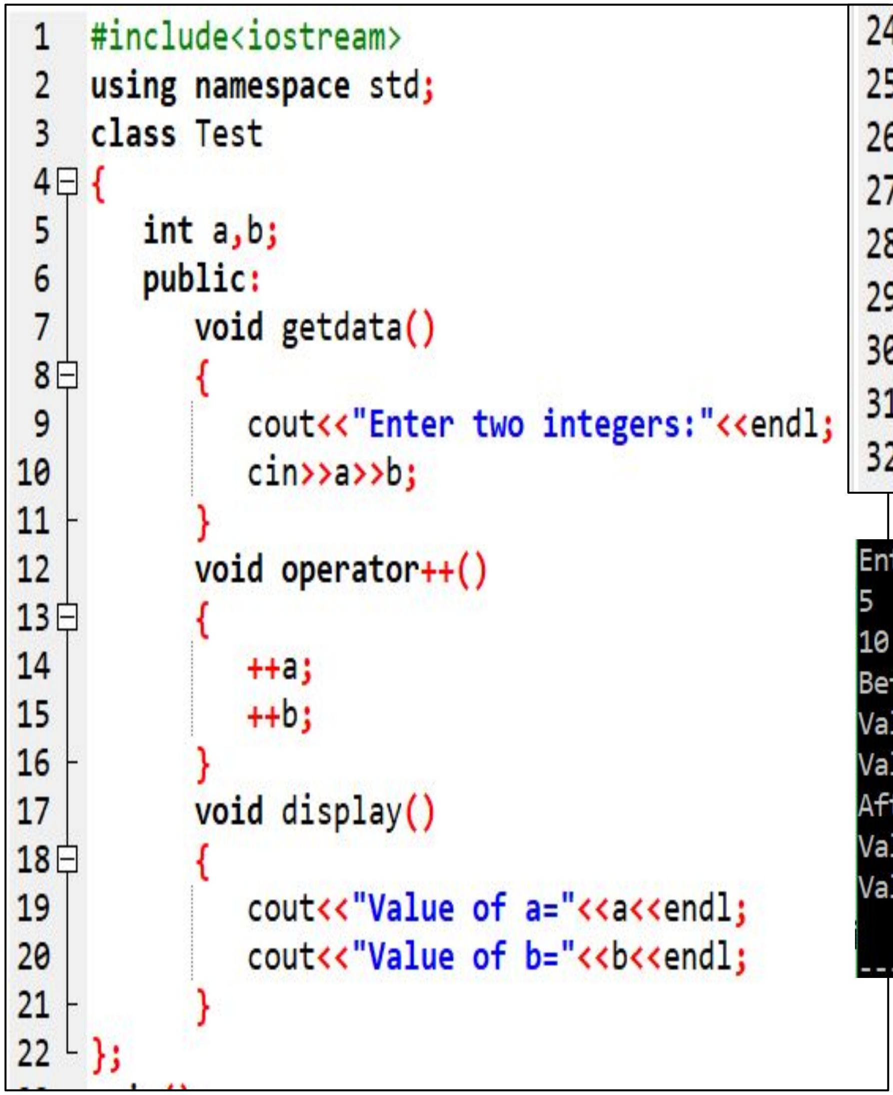


```
main()
{
    Demo ob1(5), ob2;
    cout<<"Before operator overloading"<<endl;
    ob1.display();
    ob2=-ob1;   //unary minus operator overloading
    cout<<"After operator overloading"<<endl;
    ob2.display();
}
```

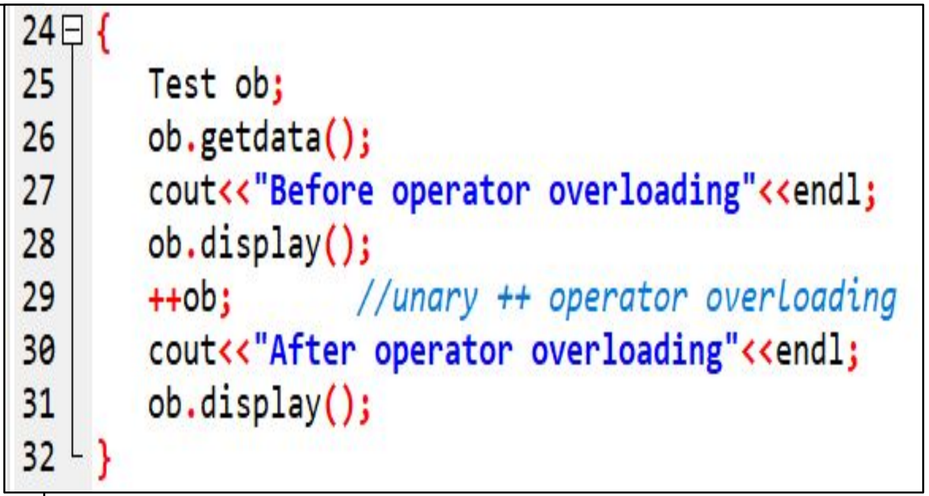


```
Before operator overloading
Value of x=5
After operator overloading
Value of x=-5

------------------------------
```

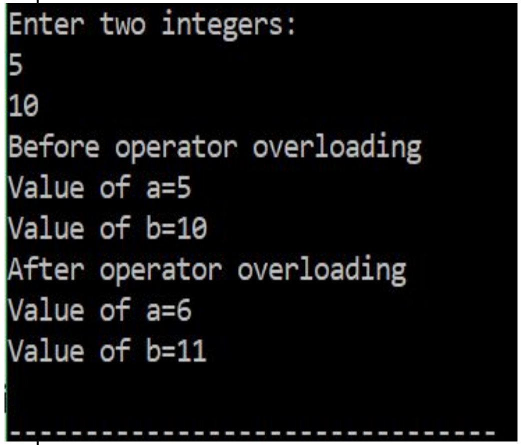
Sample program 2: WAP to overload increment operator (++) in prefix notation.



```cpp
#include<iostream>
using namespace std;
class Test
{
    int a,b;
    public:
        void getdata()
        {
            cout<<"Enter two integers:"<<endl;
            cin>>a>>b;
        }
        void operator++()
        {
            ++a;
            ++b;
        }
        void display()
        {
            cout<<"Value of a="<<a<<endl;
            cout<<"Value of b="<<b<<endl;
        }
};
```

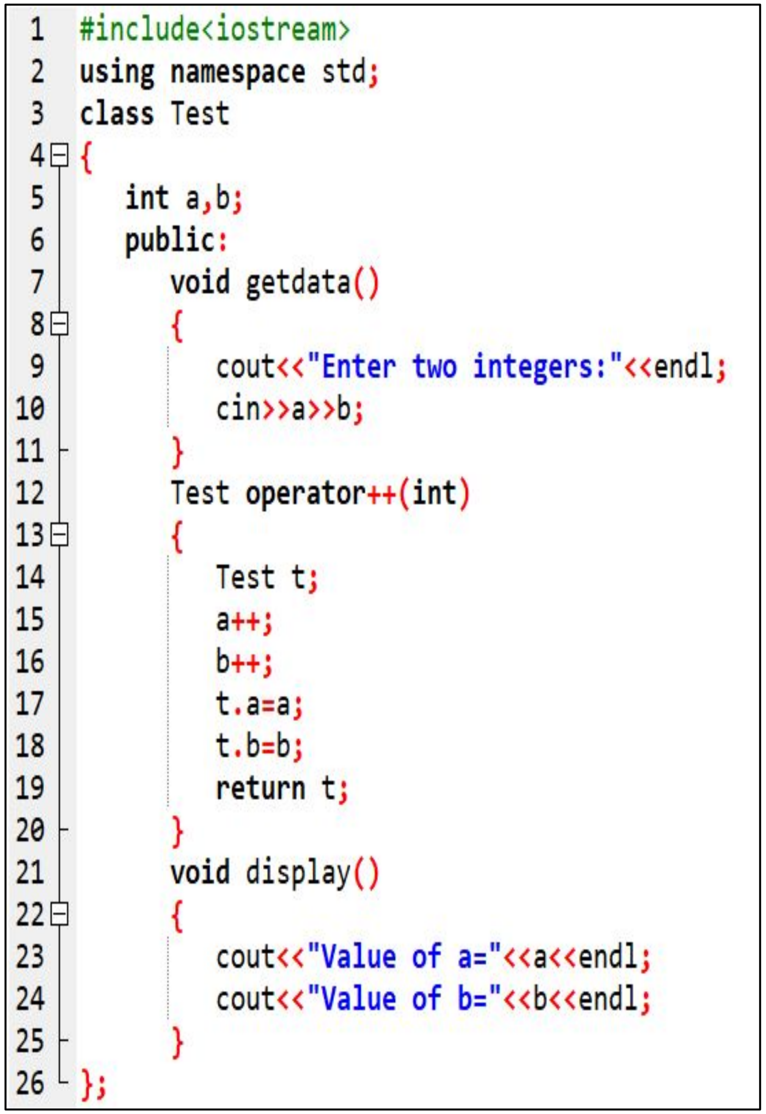


```cpp
{
    Test ob;
    ob.getdata();
    cout<<"Before operator overloading"<<endl;
    ob.display();
    ++ob;           //unary ++ operator overloading
    cout<<"After operator overloading"<<endl;
    ob.display();
}
```

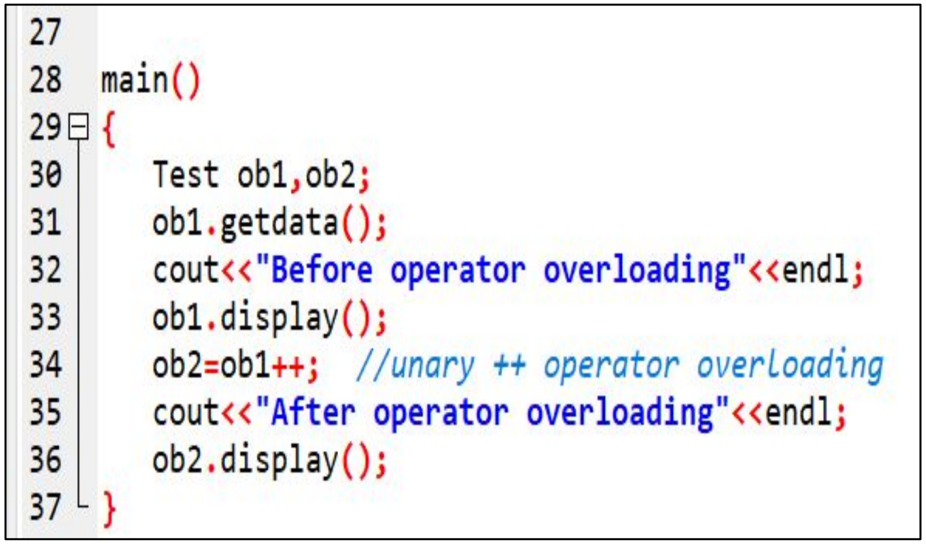


```
Enter two integers:
5
10
Before operator overloading
Value of a=5
Value of b=10
After operator overloading
Value of a=6
Value of b=11

--------------------------------
```

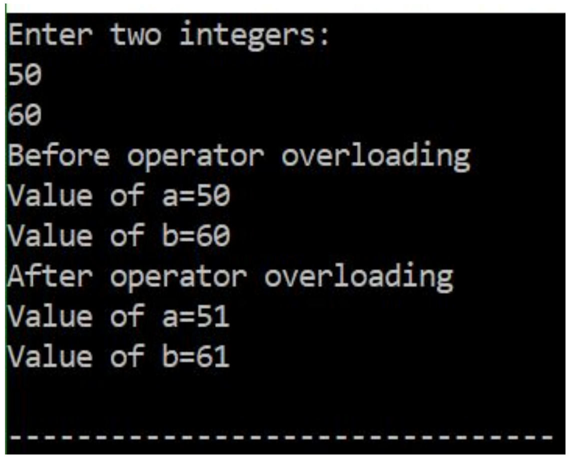
Sample program 3: WAP to overload increment operator (++) in postfix notation.



```cpp
#include<iostream>
using namespace std;
class Test
{
    int a,b;
    public:
        void getdata()
        {
            cout<<"Enter two integers:"<<endl;
            cin>>a>>b;
        }
        Test operator++(int)
        {
            Test t;
            a++;
            b++;
            t.a=a;
            t.b=b;
            return t;
        }
        void display()
        {
            cout<<"Value of a="<<a<<endl;
            cout<<"Value of b="<<b<<endl;
        }
};
```



```cpp

main()
{
    Test ob1,ob2;
    ob1.getdata();
    cout<<"Before operator overloading"<<endl;
    ob1.display();
    ob2=ob1++;   //unary ++ operator overloading
    cout<<"After operator overloading"<<endl;
    ob2.display();
}
```



```
Enter two integers:
50
60
Before operator overloading
Value of a=50
Value of b=60
After operator overloading
Value of a=51
Value of b=61

--------------------------------
```

Practise:

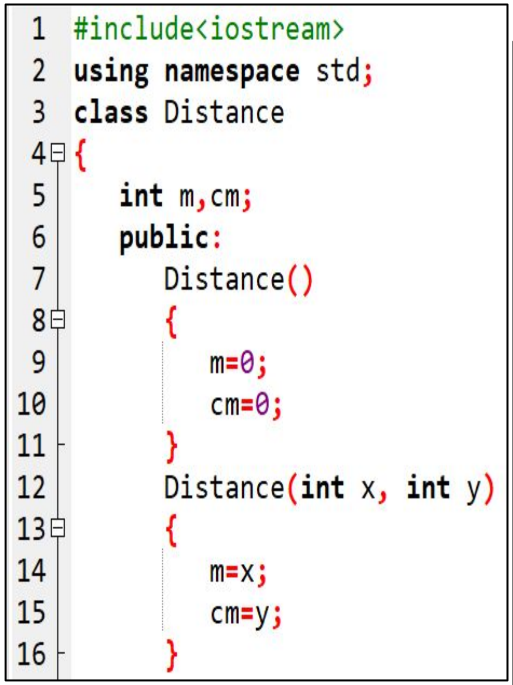
1. WAP to overload decrement operator (--) in prefix notation.
2. WAP to overload decrement operator (--) in postfix notation.
3. WAP to input x and y coordinate and move 4 units from both x and y direction by overloading unary ++ operator in prefix notation.
4. WAP to input x and y coordinate and move 4 units from both x and y direction by overloading unary ++ operator in postfix notation.

ii) Binary Operator overloading

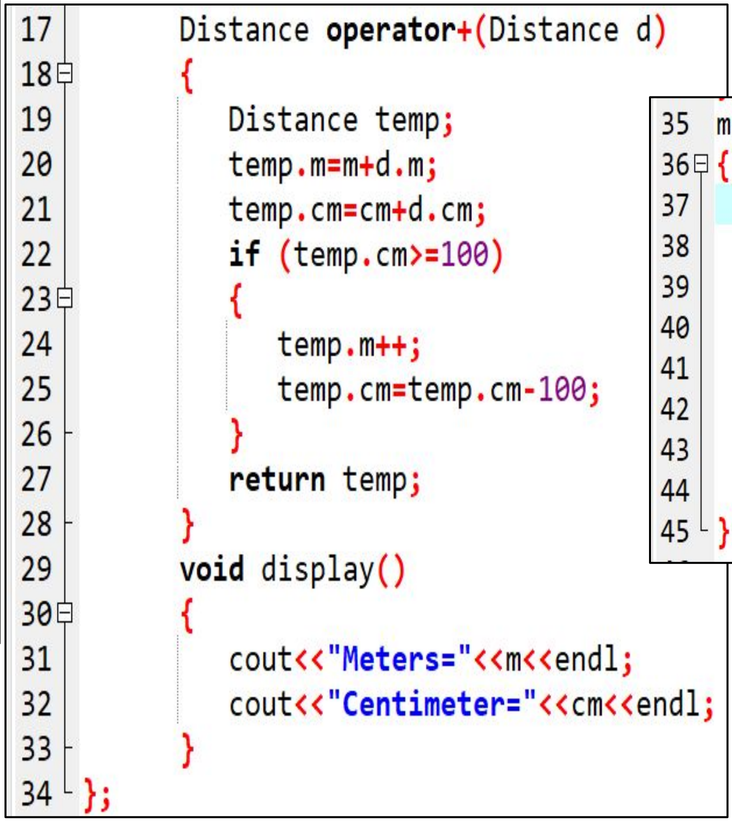
- The operators which require two operand for its operations are known as binary operators.
- Examples of binary operators :
  - binary plus(+),
  - binary minus(-),
  - greater than(>), equal to (==) , etc.
- The operations in which the binary operators are overloaded are known as binary operator overloading.

Sample program 1 :

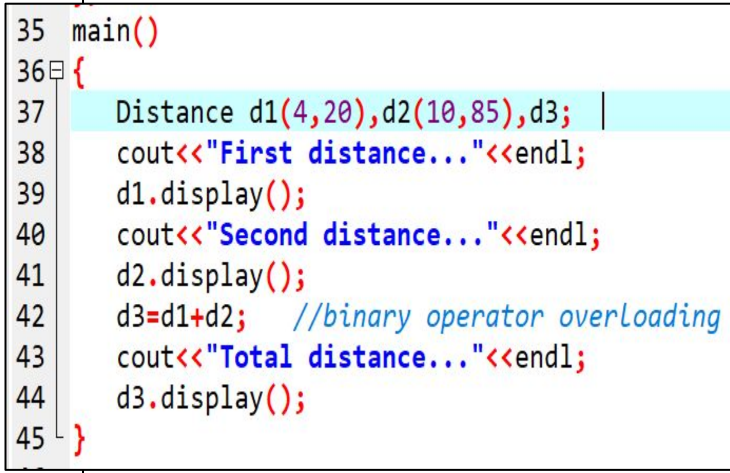
WAP to overload binary plus (+) operator for addition of two distances expressed in m and cm.



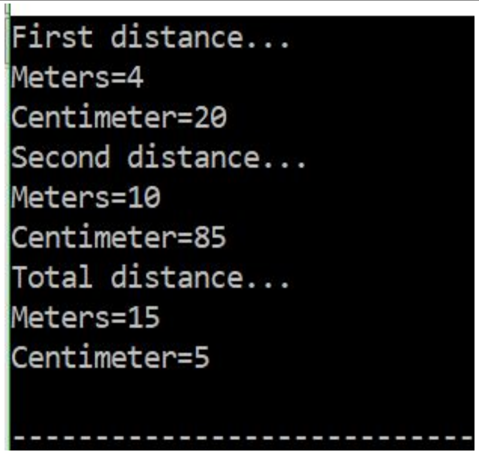
```cpp
#include<iostream>
using namespace std;
class Distance
{
    int m,cm;
    public:
        Distance()
        {
            m=0;
            cm=0;
        }
        Distance(int x, int y)
        {
            m=x;
            cm=y;
        }
```



```cpp
        Distance operator+(Distance d)
        {
            Distance temp;
            temp.m=m+d.m;
            temp.cm=cm+d.cm;
            if (temp.cm>=100)
            {
                temp.m++;
                temp.cm=temp.cm-100;
            }
            return temp;
        }
        void display()
        {
            cout<<"Meters="<<m<<endl;
            cout<<"Centimeter="<<cm<<endl;
        }
};
```



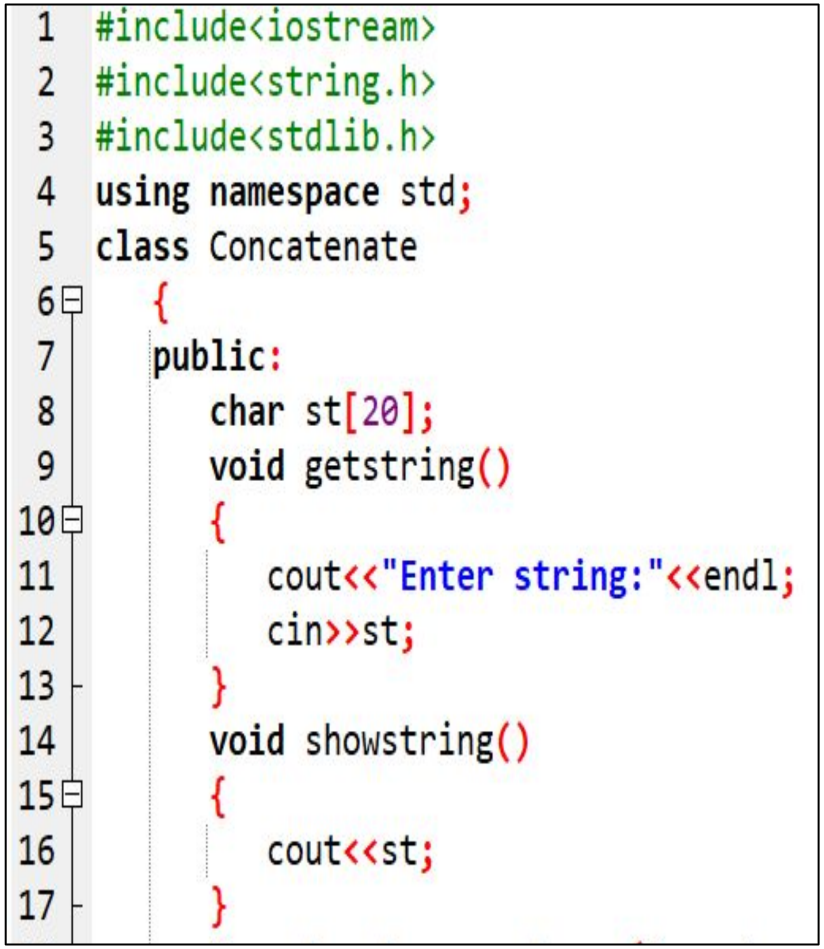
```cpp
main()
{
    Distance d1(4,20),d2(10,85),d3;
    cout<<"First distance..."<<endl;
    d1.display();
    cout<<"Second distance..."<<endl;
    d2.display();
    d3=d1+d2;    //binary operator overloading
    cout<<"Total distance..."<<endl;
    d3.display();
}
```



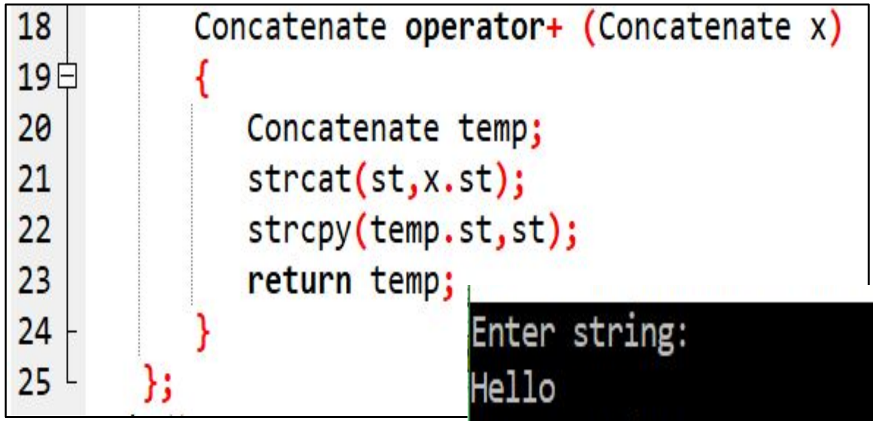
```
First distance...
Meters=4
Centimeter=20
Second distance...
Meters=10
Centimeter=85
Total distance...
Meters=15
Centimeter=5

----------------------------
```

Sample program 2 :

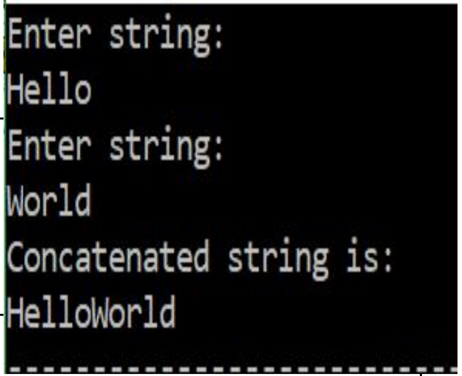
WAP to overload binary plus (+) operator for concatenation of two input strings.



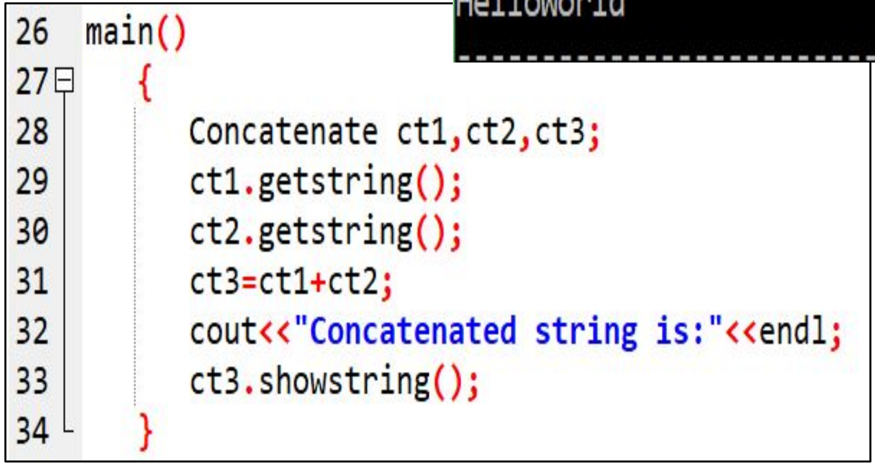
```
#include<iostream>
#include<string.h>
#include<stdlib.h>
using namespace std;
class Concatenate
  {
  public:
      char st[20];
      void getstring()
      {
          cout<<"Enter string:"<<endl;
          cin>>st;
      }
      void showstring()
      {
          cout<<st;
      }
```



```
      Concatenate operator+ (Concatenate x)
      {
          Concatenate temp;
          strcat(st,x.st);
          strcpy(temp.st,st);
          return temp;
      }
  };
```

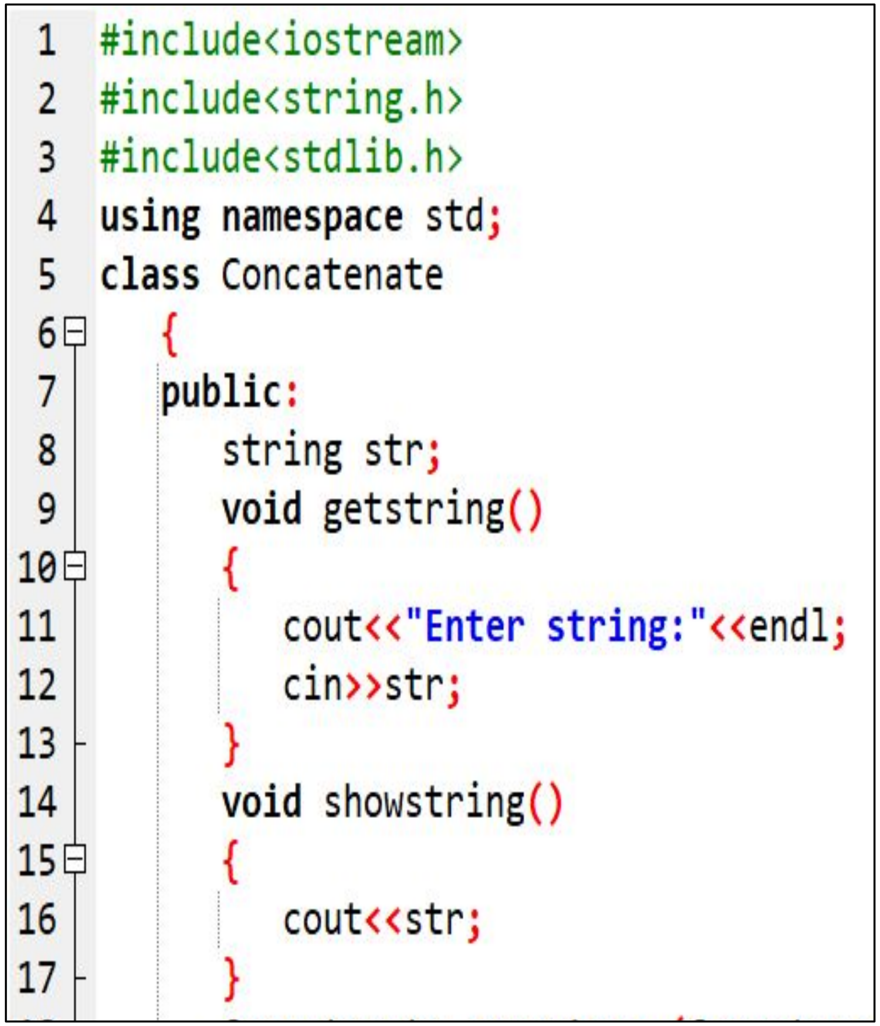


```
Enter string:
Hello
Enter string:
World
Concatenated string is:
HelloWorld
```

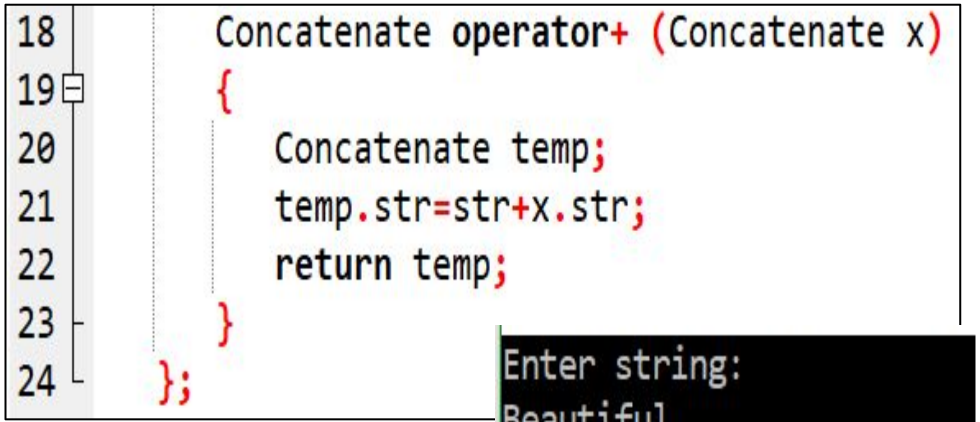


```
main()
  {
      Concatenate ct1,ct2,ct3;
      ct1.getstring();
      ct2.getstring();
      ct3=ct1+ct2;
      cout<<"Concatenated string is:"<<endl;
      ct3.showstring();
  }
```

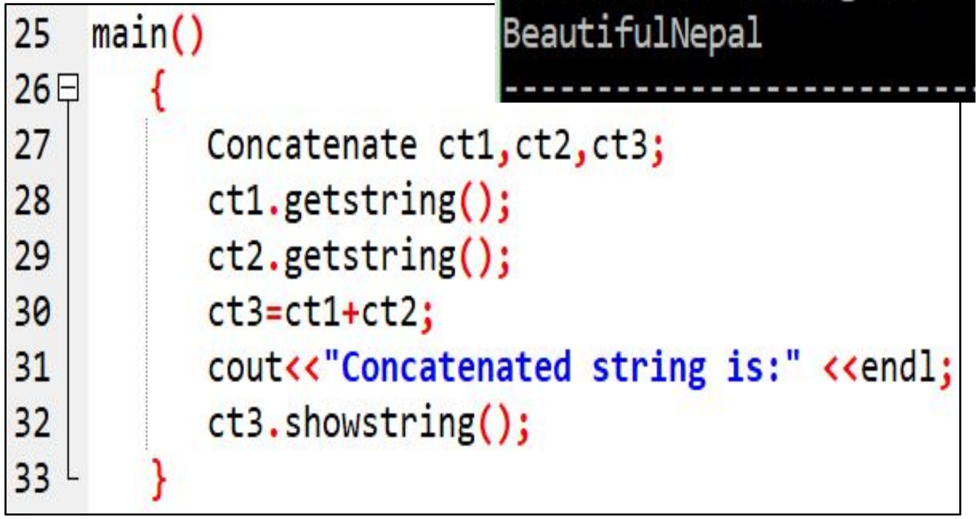
The above program can also be coded as:



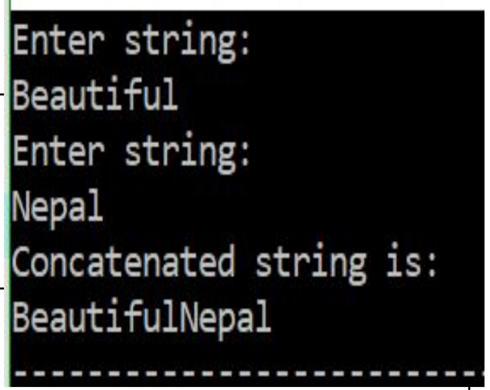
```cpp
#include<iostream>
#include<string.h>
#include<stdlib.h>
using namespace std;
class Concatenate
   {
   public:
      string str;
      void getstring()
      {
         cout<<"Enter string:"<<endl;
         cin>>str;
      }
      void showstring()
      {
         cout<<str;
      }
```



```cpp
      Concatenate operator+ (Concatenate x)
      {
         Concatenate temp;
         temp.str=str+x.str;
         return temp;
      }
   };
```



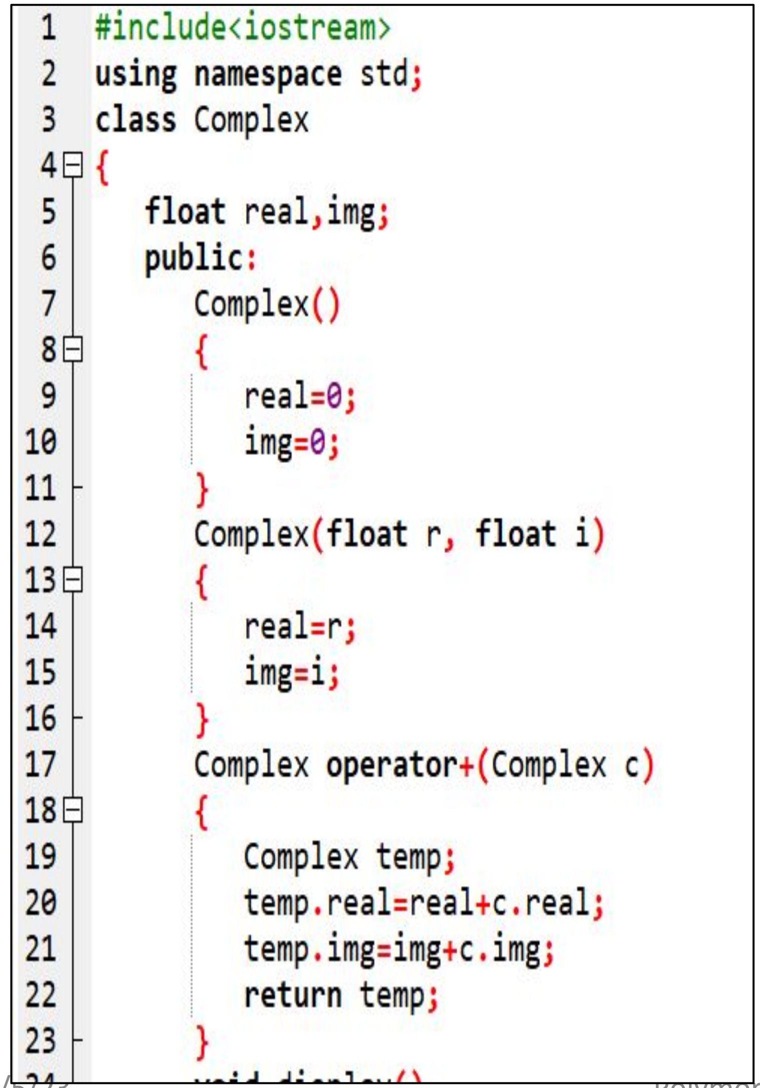
```cpp
main()
   {
      Concatenate ct1,ct2,ct3;
      ct1.getstring();
      ct2.getstring();
      ct3=ct1+ct2;
      cout<<"Concatenated string is:" <<endl;
      ct3.showstring();
   }
```



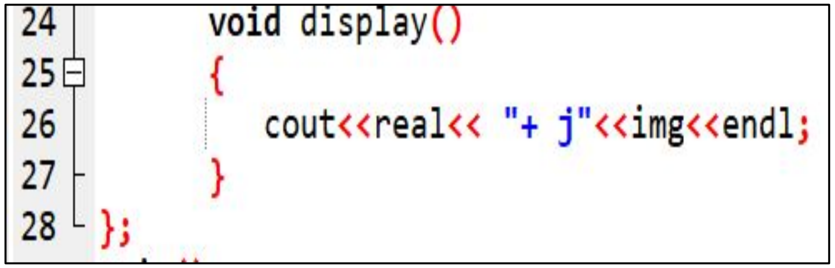
```
Enter string:
Beautiful
Enter string:
Nepal
Concatenated string is:
BeautifulNepal
-------------------------
```

Sample program 3 :

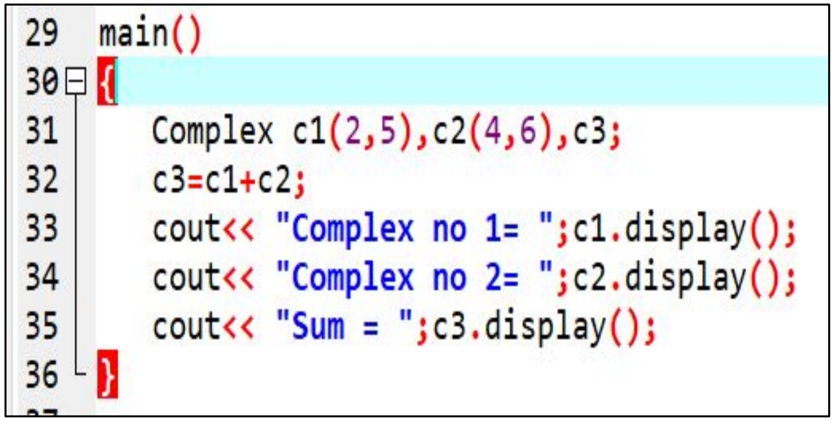
WAP to overload binary plus (+) operator for addition of two complex numbers.



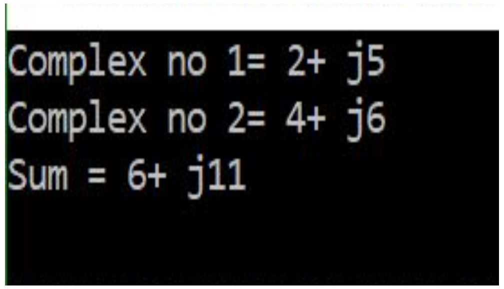
```cpp
#include<iostream>
using namespace std;
class Complex
{
    float real,img;
    public:
        Complex()
        {
            real=0;
            img=0;
        }
        Complex(float r, float i)
        {
            real=r;
            img=i;
        }
        Complex operator+(Complex c)
        {
            Complex temp;
            temp.real=real+c.real;
            temp.img=img+c.img;
            return temp;
        }
        void display()
```



```cpp
        void display()
        {
            cout<<real<< "+ j"<<img<<endl;
        }
};
```



```cpp
main()
{
    Complex c1(2,5),c2(4,6),c3;
    c3=c1+c2;
    cout<< "Complex no 1= ";c1.display();
    cout<< "Complex no 2= ";c2.display();
    cout<< "Sum = ";c3.display();
}
```



```
Complex no 1= 2+ j5
Complex no 2= 4+ j6
Sum = 6+ j11
```

Practise:

1. WAP to overload less than relational operator (<)
2. WAP to overload equality operator (==)
3. WAP to enter two amount of money in terms of rupees and paisa and then add them using binary plus (+) operator overloading.

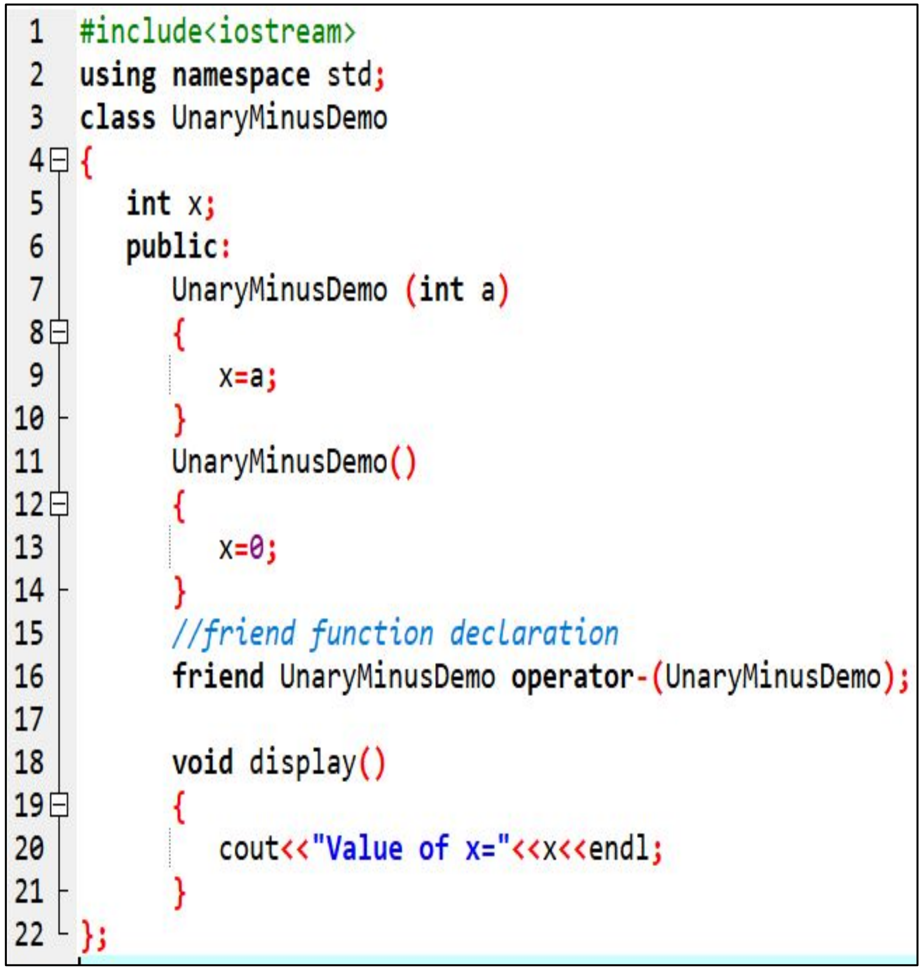
5.4.3 Operator Overloading using Friend function

- Friend function can be used as operator function in place of member function for overloading of operators.
- The only difference is that a friend function require:
  - only one argument for unary operators and
  - two arguments for binary operators

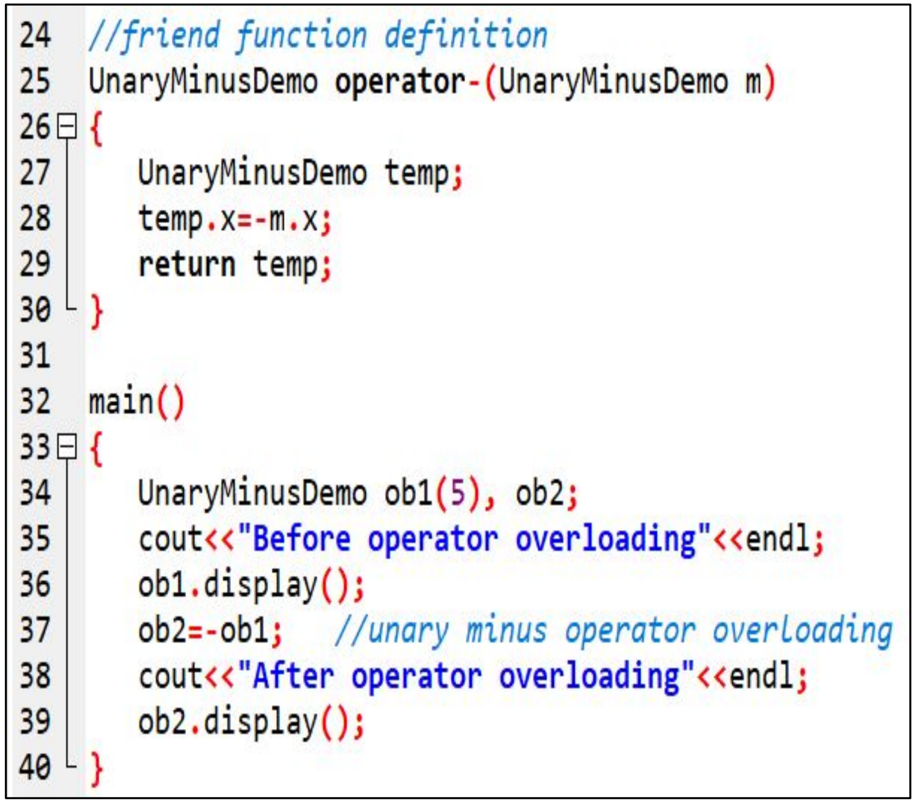
  to be explicitly passed to it.
- As the friend function is non-member function of a class, it is called without object.
- Therefore, there is no calling object and all objects are passed in operator function via arguments.

Sample program :

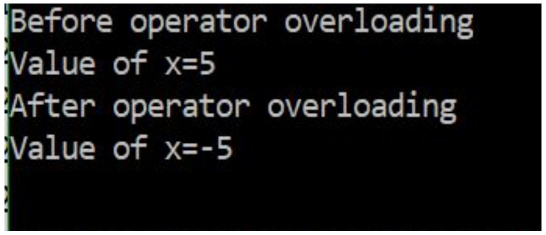
WAP to overload unary minus (-) operator using friend function.



```cpp
#include<iostream>
using namespace std;
class UnaryMinusDemo
{
    int x;
    public:
        UnaryMinusDemo (int a)
        {
            x=a;
        }
        UnaryMinusDemo()
        {
            x=0;
        }
        //friend function declaration
        friend UnaryMinusDemo operator-(UnaryMinusDemo);

        void display()
        {
            cout<<"Value of x="<<x<<endl;
        }
};
```



```cpp
//friend function definition
UnaryMinusDemo operator-(UnaryMinusDemo m)
{
    UnaryMinusDemo temp;
    temp.x=-m.x;
    return temp;
}

main()
{
    UnaryMinusDemo ob1(5), ob2;
    cout<<"Before operator overloading"<<endl;
    ob1.display();
    ob2=-ob1;   //unary minus operator overloading
    cout<<"After operator overloading"<<endl;
    ob2.display();
}
```



```
Before operator overloading
Value of x=5
After operator overloading
Value of x=-5
```

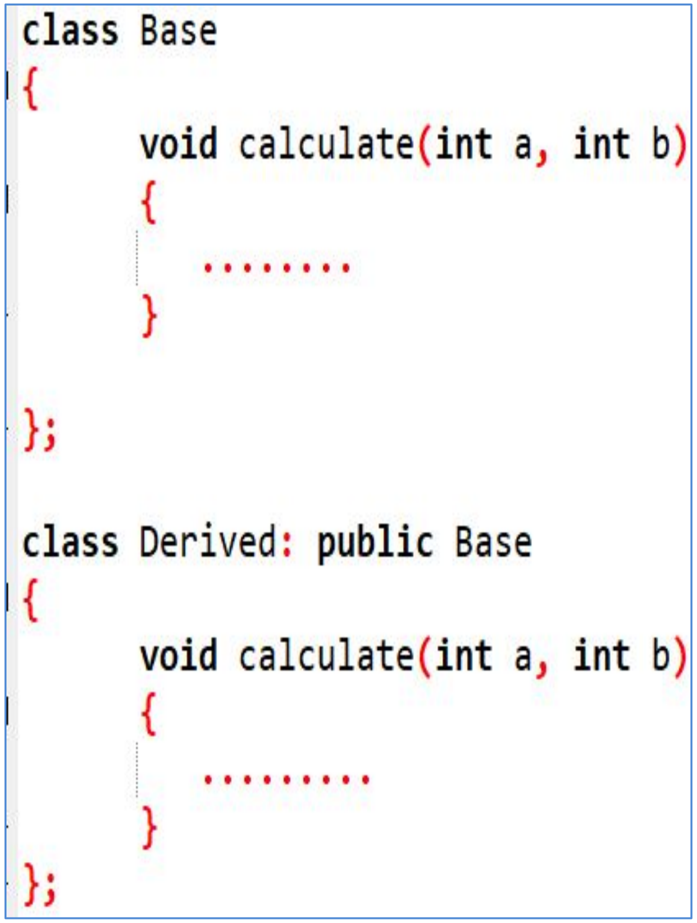
For other sample programs:

Please refer:

A textbook of Object Oriented Programming in C++, Ram Datta Bhatta, pp159-166

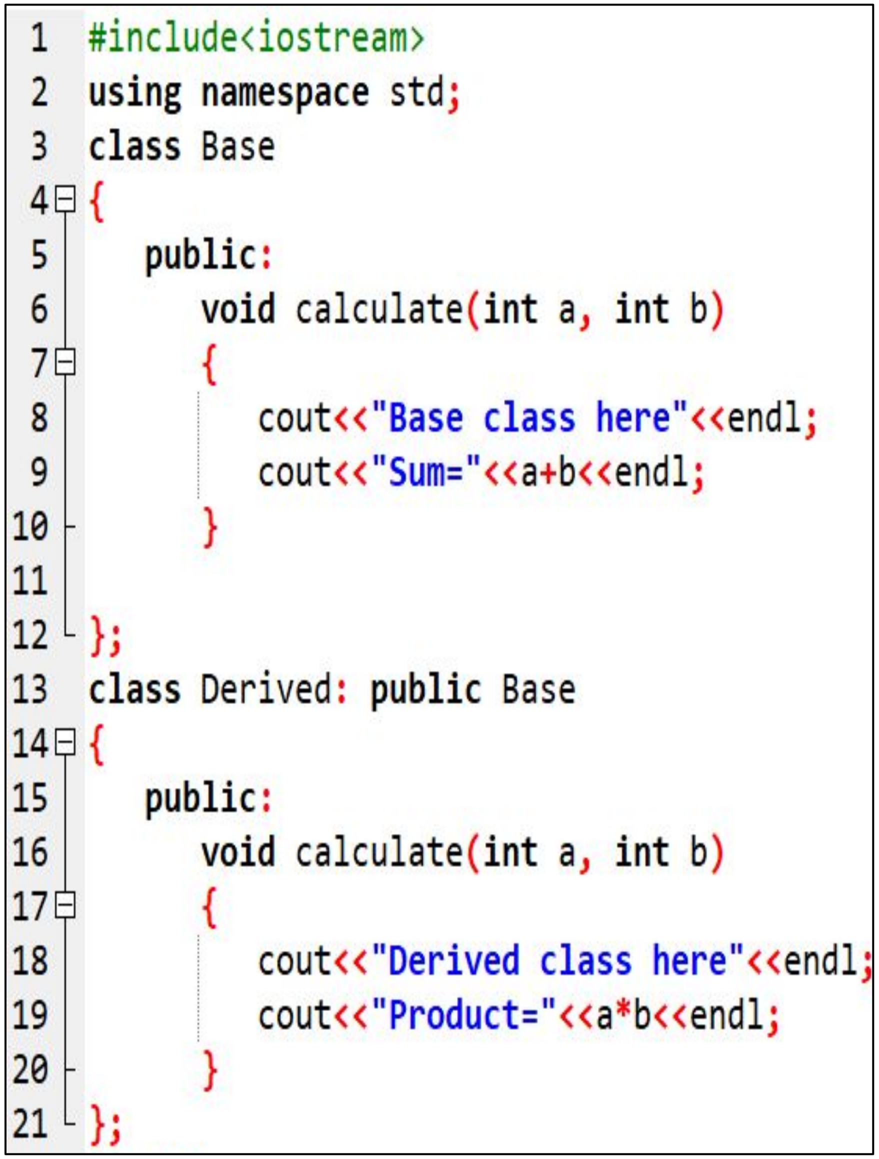
5.5 Function Overriding

- Function overriding is the use of two or more functions having same name and argument types, but defined one in the base class and other in derived class.
- The instance of base class is replaced (or overridden) by the instance of child class.
- Function overriding cannot be done within a class.
- However, when a function with same name and signature is defined in base class and derived class, ambiguity occurs.
- Such ambiguity is resolved in two ways:
  - Using scope resolution operator
  - Using virtual function.

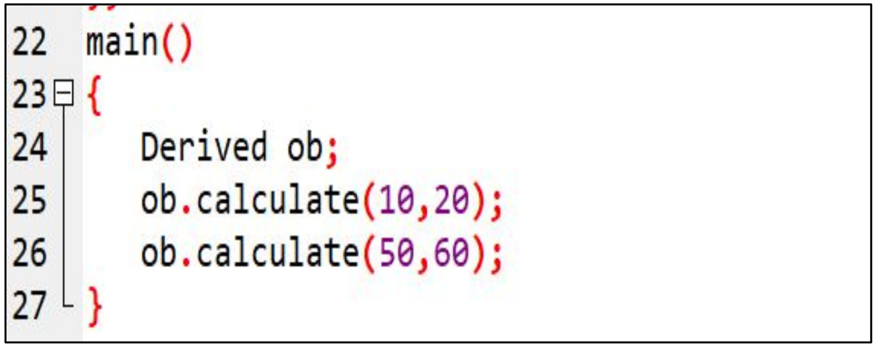


```
class Base
{
        void calculate(int a, int b)
        {
            ........
        }

};

class Derived: public Base
{
        void calculate(int a, int b)
        {
            .........
        }
};
```

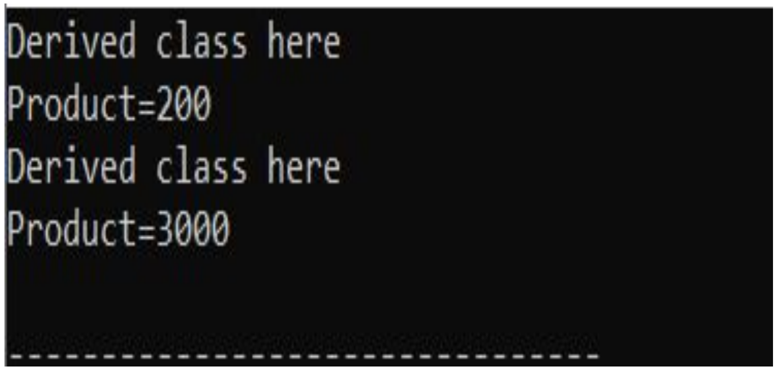
5.5 Function Overriding : Sample program



```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
        void calculate(int a, int b)
        {
            cout<<"Base class here"<<endl;
            cout<<"Sum="<<a+b<<endl;
        }

};
class Derived: public Base
{
    public:
        void calculate(int a, int b)
        {
            cout<<"Derived class here"<<endl;
            cout<<"Product="<<a*b<<endl;
        }
};
```

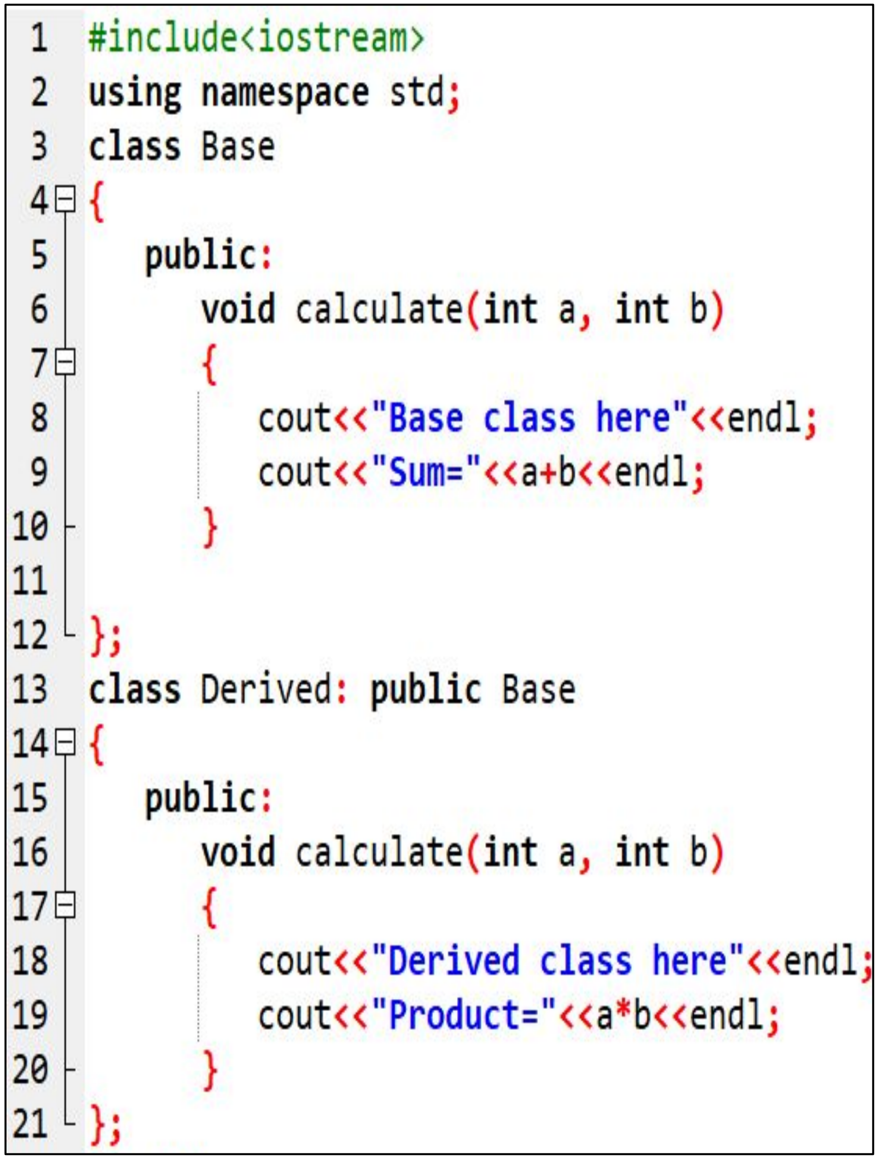


```cpp
main()
{
    Derived ob;
    ob.calculate(10,20);
    ob.calculate(50,60);
}
```

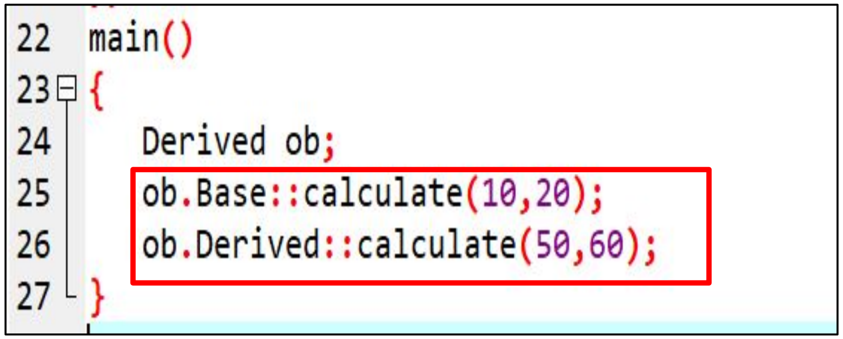


```
Derived class here
Product=200
Derived class here
Product=3000


--------------------------------
```

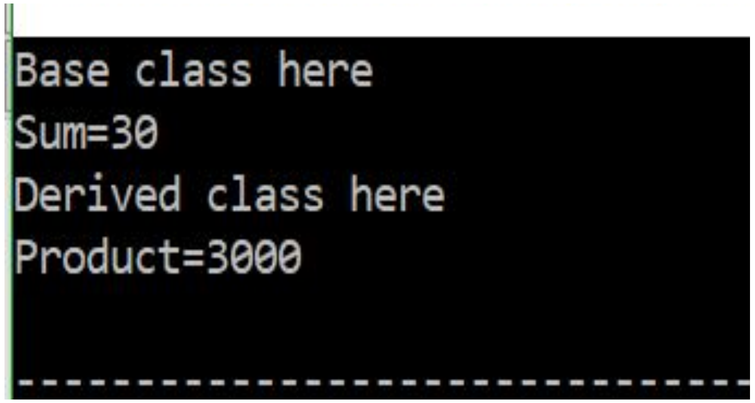
5.5 Function Overriding : Resolved



```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
        void calculate(int a, int b)
        {
            cout<<"Base class here"<<endl;
            cout<<"Sum="<<a+b<<endl;
        }

};
class Derived: public Base
{
    public:
        void calculate(int a, int b)
        {
            cout<<"Derived class here"<<endl;
            cout<<"Product="<<a*b<<endl;
        }
};
```



```cpp
main()
{
    Derived ob;
    ob.Base::calculate(10,20);
    ob.Derived::calculate(50,60);
}
```

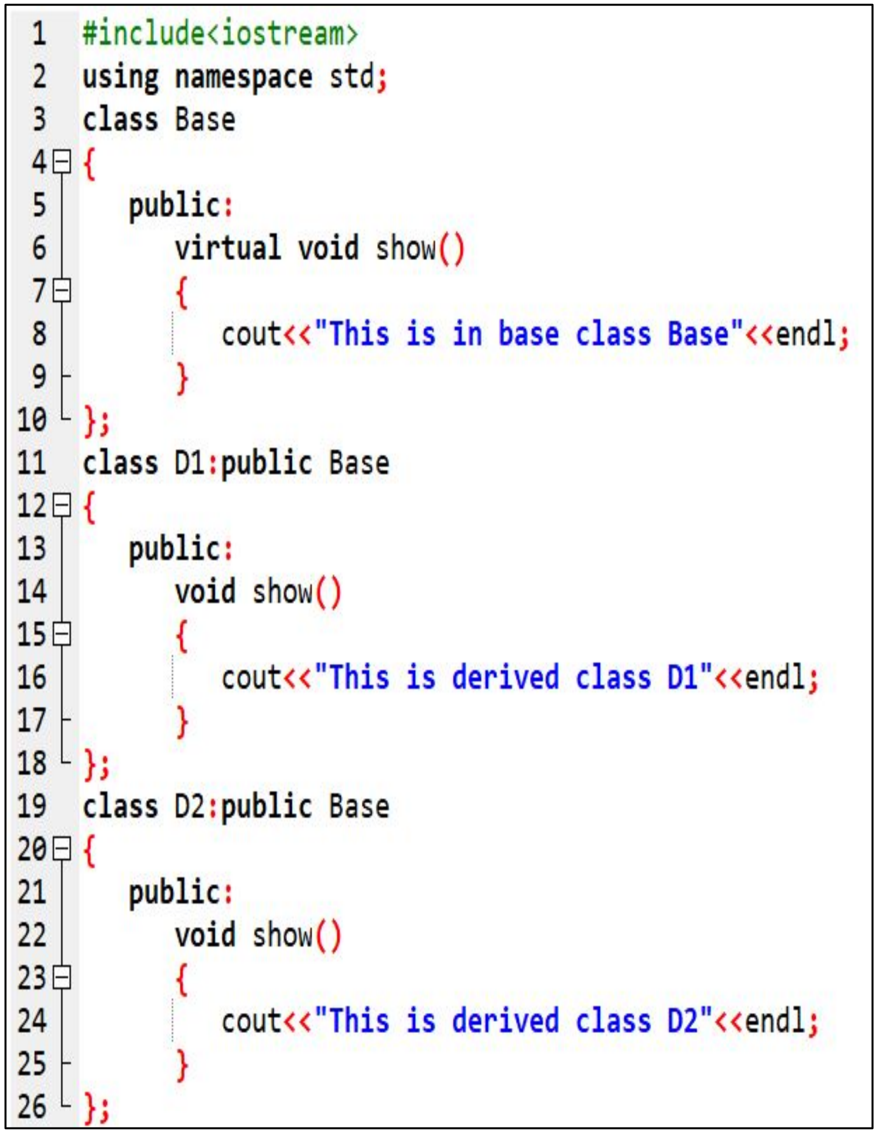


```
Base class here
Sum=30
Derived class here
Product=3000

------------------------------
```

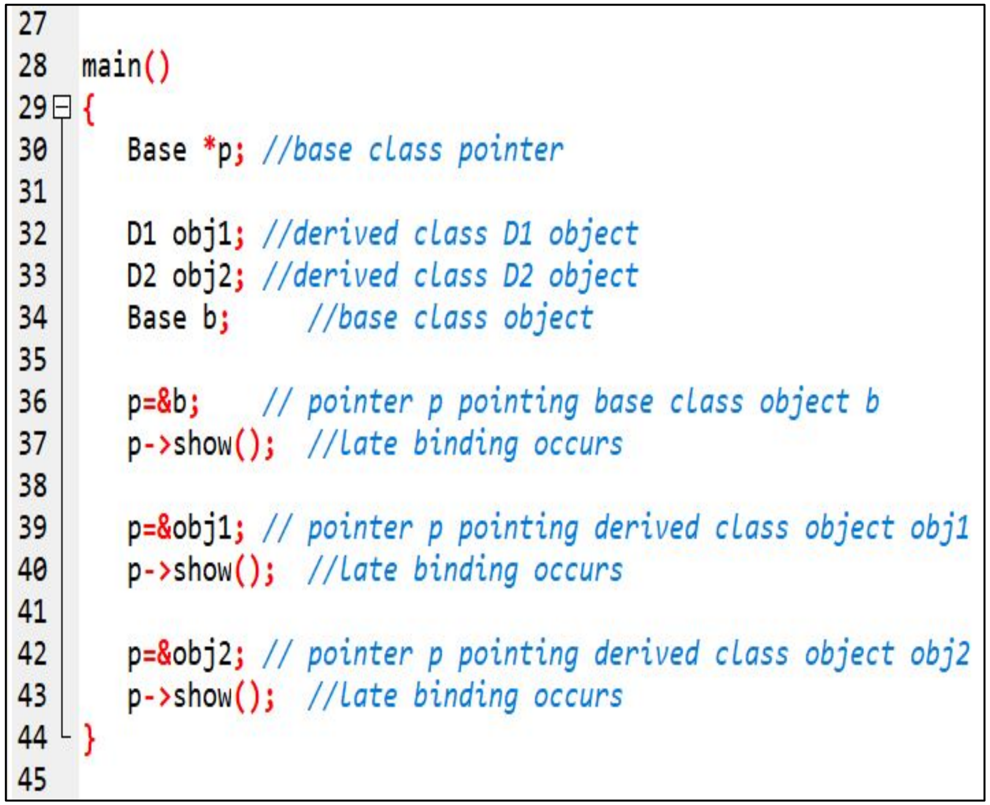
5.6 Virtual Function overriding

- A Virtual function is a member function declared in a base class with keyword ***virtual*** and referenced by derived class.
- Virtual means existing in effect but not in reality.
- Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy.
- For accessing the derived class function members, the base class pointers are used to point derived class objects.

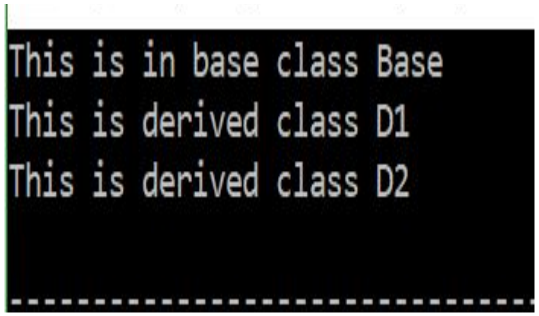
5.6 Virtual Function overriding : Sample program



```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
        virtual void show()
        {
            cout<<"This is in base class Base"<<endl;
        }
};
class D1:public Base
{
    public:
        void show()
        {
            cout<<"This is derived class D1"<<endl;
        }
};
class D2:public Base
{
    public:
        void show()
        {
            cout<<"This is derived class D2"<<endl;
        }
};
```



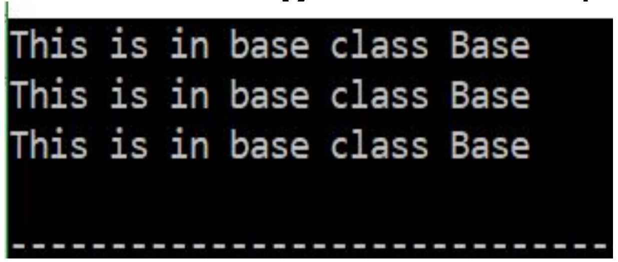
```cpp

main()
{
    Base *p; //base class pointer

    D1 obj1; //derived class D1 object
    D2 obj2; //derived class D2 object
    Base b;     //base class object

    p=&b;     // pointer p pointing base class object b
    p->show();  //late binding occurs

    p=&obj1; // pointer p pointing derived class object obj1
    p->show();  //late binding occurs

    p=&obj2; // pointer p pointing derived class object obj2
    p->show();  //late binding occurs
}

```



```
This is in base class Base
This is derived class D1
This is derived class D2


--------------------------------
```

5.6 Virtual Function overriding : Sample program

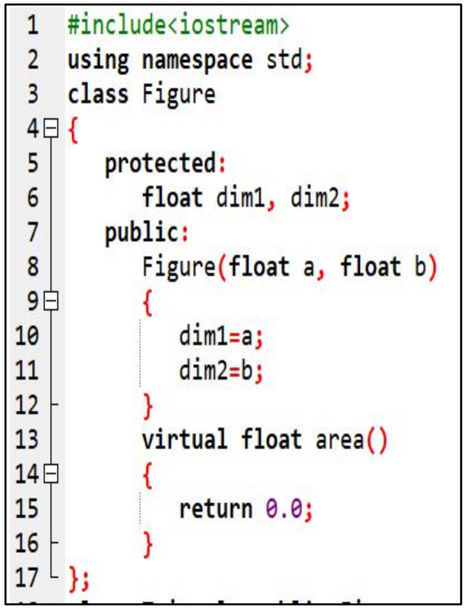
- In the above example, if the function ***show()*** in the **Base** class was not defined as **virtual**, then the Base class pointer **(*b)** would always call the base class version of ***show()*** and the output would be:



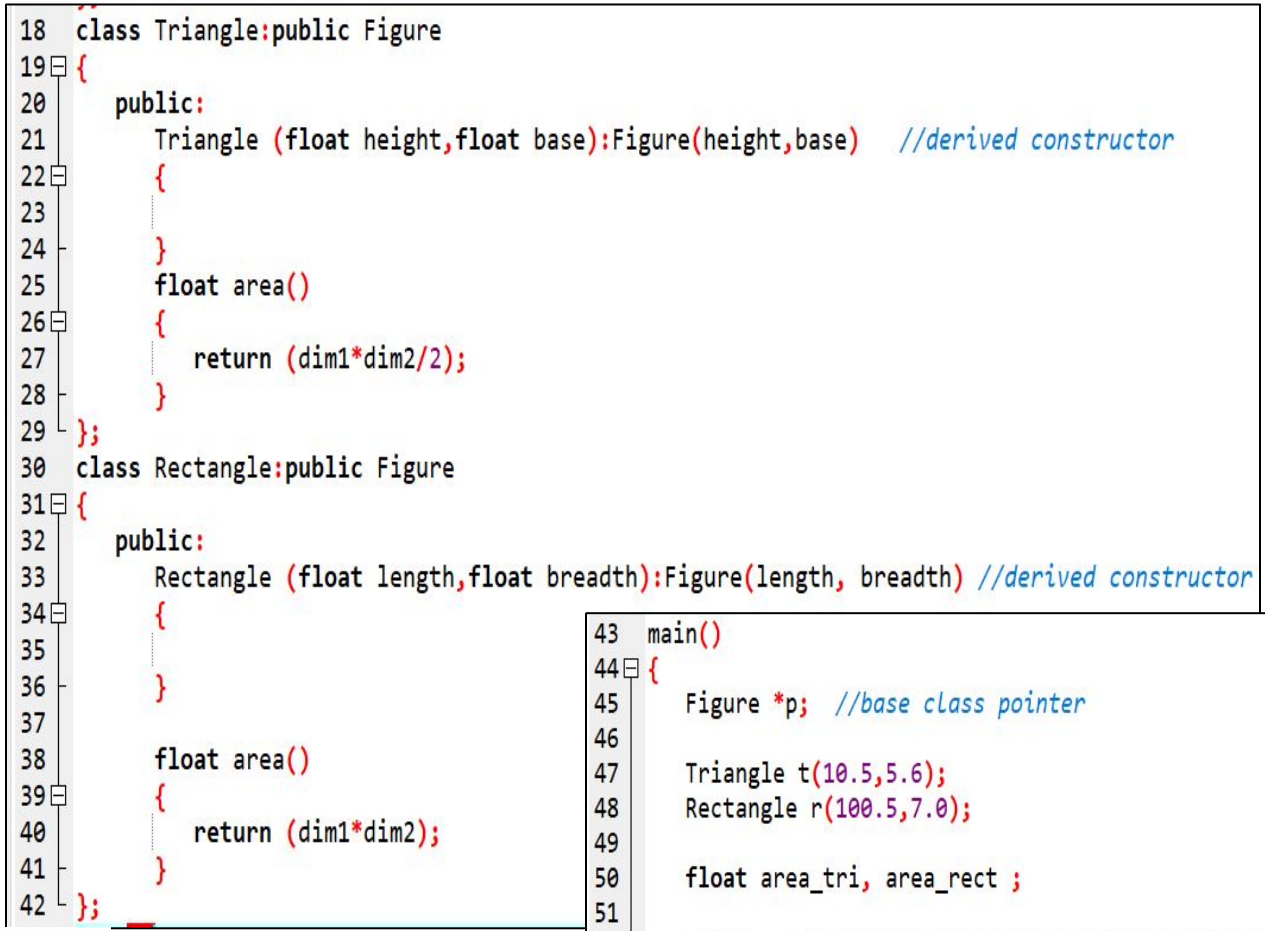
```
This is in base class Base
This is in base class Base
This is in base class Base

------------------------------
```

- This was because the C++ compiler ignores the content of **b** (which is address of **obj1** after statement ***p=&obj1;***) and execute the function which is inherited from class Base because the type **b** matches with class Base.
- Similarly, after statement ***p=&obj2;*** and ***p->show();*** , the compiler again executes the function which is inherited from class Base.
- Whenever this type of situation occurs (i.e.in derived class, there are two functions, both have the same name: one inherited from base and another of its own, and the pointer is Base type), then we want to execute the function through pointer, compiler chooses the function of Base class. So, to overcome this situation, we have to make the base class function as virtual.
- Thus the use of same function call at different places produces different result at run time. Hence, it provides the concept of polymorphism.

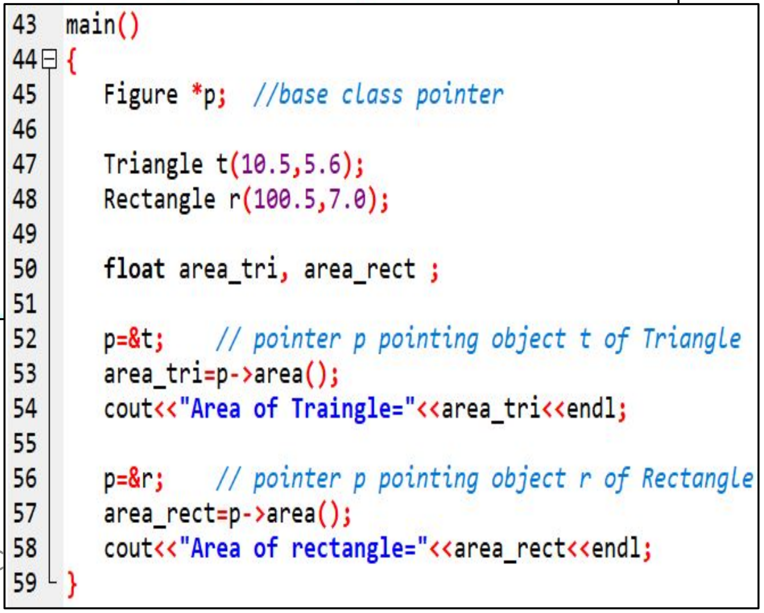
WAP to create a class *Figure* with *dim1 and dim2* as data members and constructor to initialize its data. Create a derived class called *Triangle* and define a member function *area()* in it to calculate the area of triangle. Create another derived class *Rectangle* and define a member function *area()* to calculate the area of rectangle. Implement this program using the concept of runtime polymorphism.



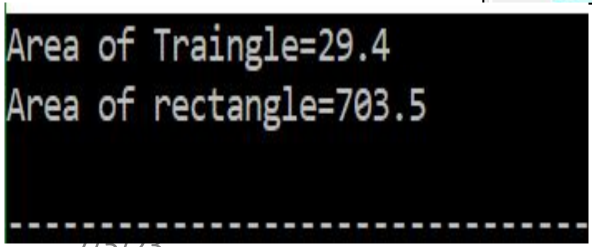
```cpp
#include<iostream>
using namespace std;
class Figure
{
    protected:
        float dim1, dim2;
    public:
        Figure(float a, float b)
        {
            dim1=a;
            dim2=b;
        }
        virtual float area()
        {
            return 0.0;
        }
};
```



```cpp
class Triangle:public Figure
{
    public:
        Triangle (float height,float base):Figure(height,base)    //derived constructor
        {

        }
        float area()
        {
            return (dim1*dim2/2);
        }
};
class Rectangle:public Figure
{
    public:
        Rectangle (float length,float breadth):Figure(length, breadth) //derived constructor
        {

        }

        float area()
        {
            return (dim1*dim2);
        }
};
```



```cpp
main()
{
    Figure *p;  //base class pointer

    Triangle t(10.5,5.6);
    Rectangle r(100.5,7.0);

    float area_tri, area_rect ;

    p=&t;    // pointer p pointing object t of Triangle
    area_tri=p->area();
    cout<<"Area of Traingle="<<area_tri<<endl;

    p=&r;    // pointer p pointing object r of Rectangle
    area_rect=p->area();
    cout<<"Area of rectangle="<<area_rect<<endl;
}
```



```
Area of Traingle=29.4
Area of rectangle=703.5

--------------------------------
```

5.7 Pure Polymorphism

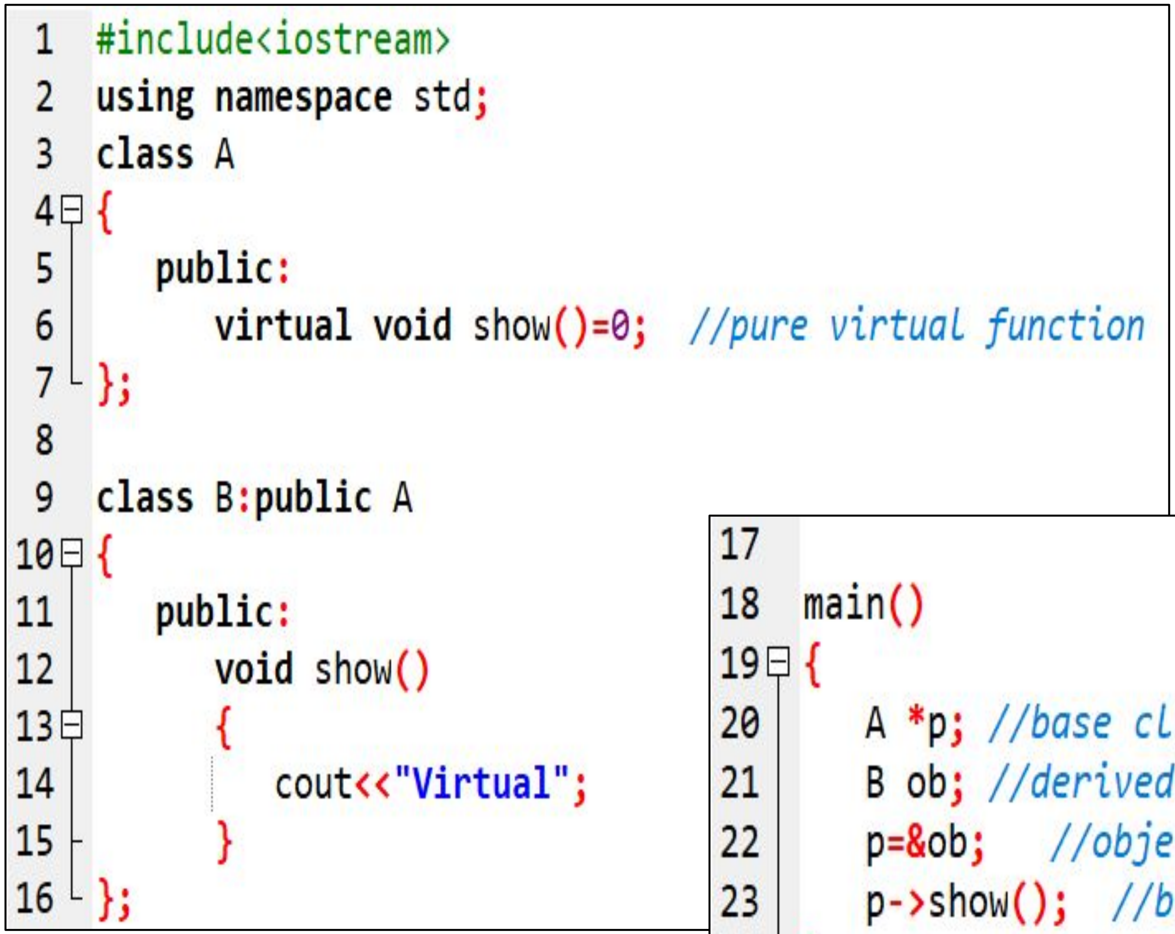
- A pure virtual function is a virtual function in the base class and has no body with it.
- They are just declared inside the base class and redefined in derived class.
- Pure virtual functions are also called “Do nothing” functions.
- In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.
- Pure virtual function cannot be used to declare any object of its own.
- The main objective of abstract base class (ABC) is to provide some facilities to the derived class and to create a base pointer required for achieving runtime polymorphism.
- General syntax for do-nothing function is:

```
virtual Return-type function-name()=0;
```

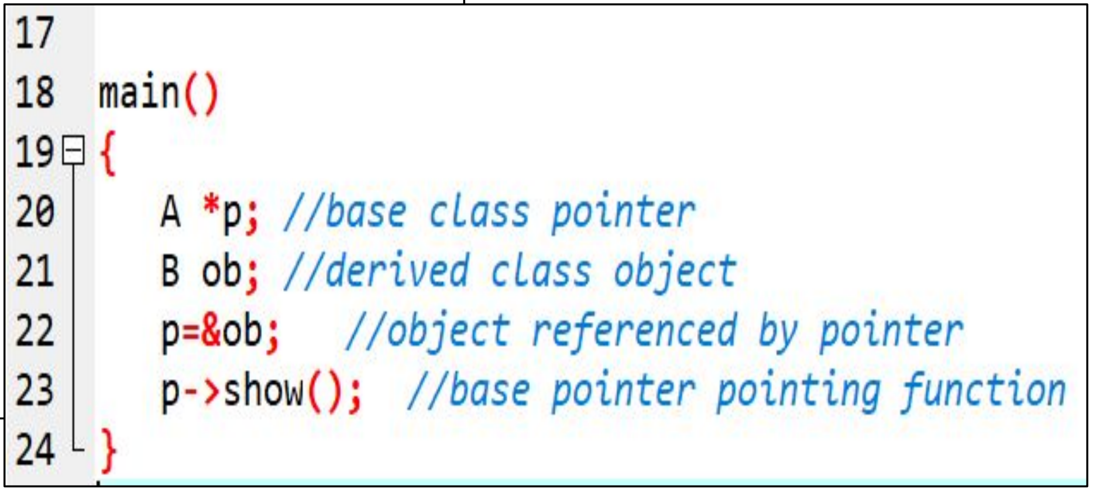
- Eg:

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

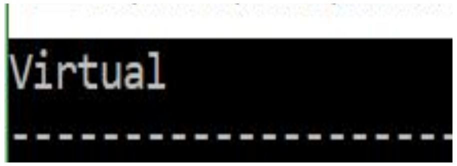
5.7 Pure Polymorphism: Sample program



```cpp
#include<iostream>
using namespace std;
class A
{
    public:
        virtual void show()=0;  //pure virtual function
};

class B:public A
{
    public:
        void show()
        {
            cout<<"Virtual";
        }
};
```

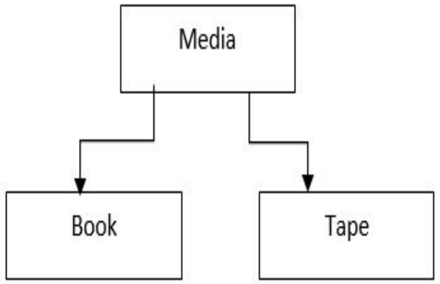


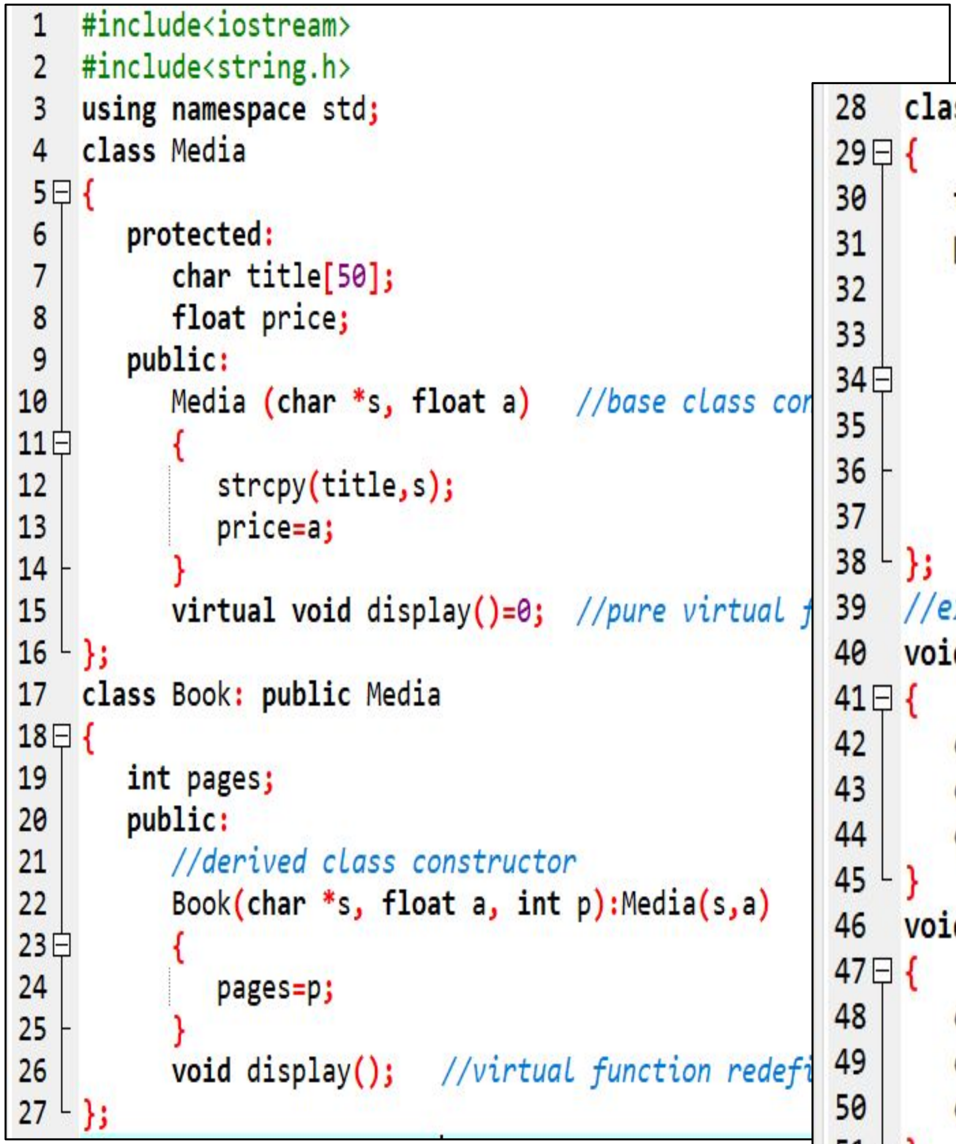
```cpp

main()
{
    A *p; //base class pointer
    B ob; //derived class object
    p=&ob;   //object referenced by pointer
    p->show();  //base pointer pointing function
}
```



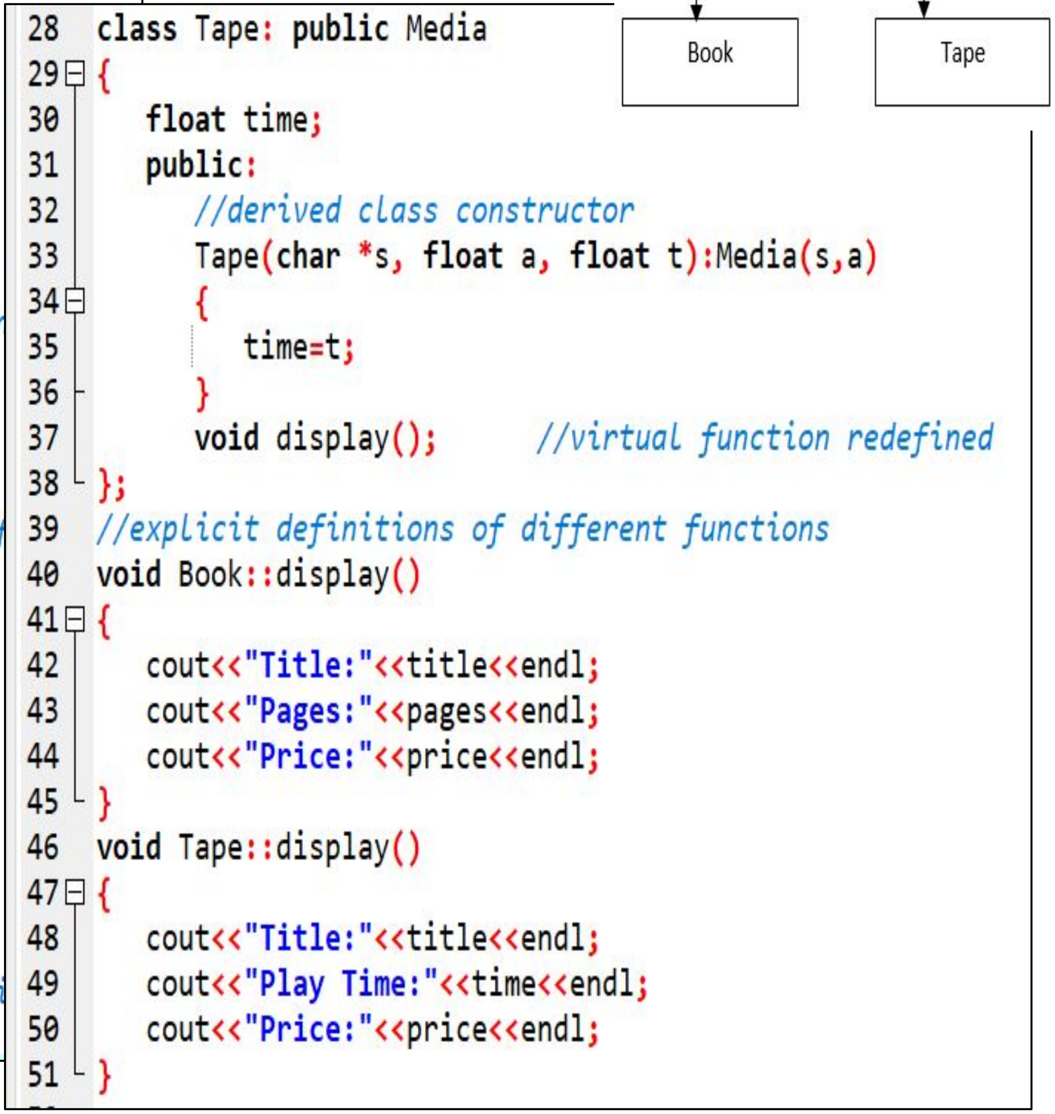
```
Virtual
--------------------
```

5.7 Pure Polymorphism: Sample program



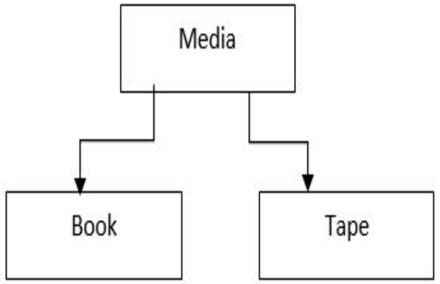


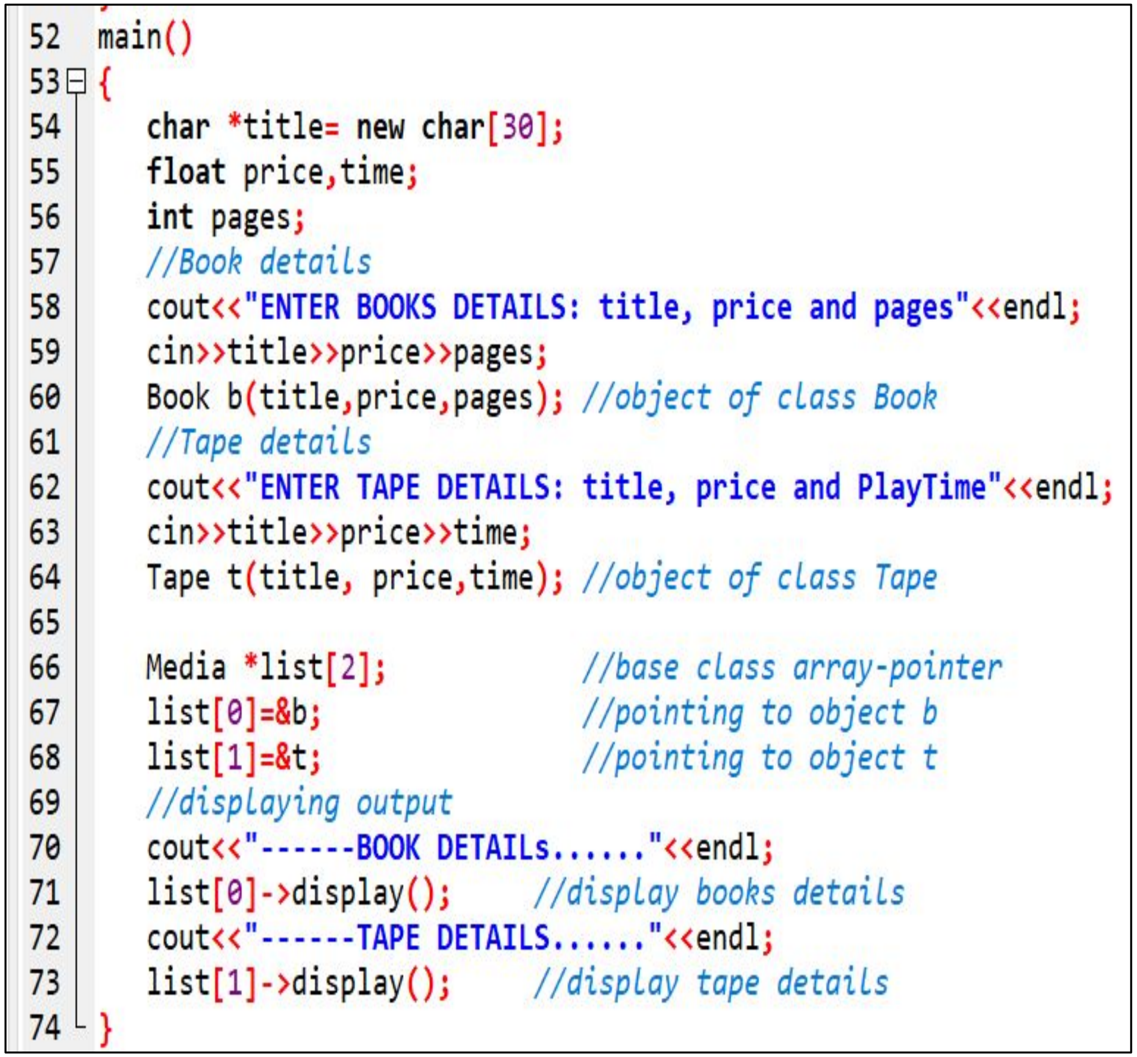
```
#include<iostream>
#include<string.h>
using namespace std;
class Media
{
    protected:
        char title[50];
        float price;
    public:
        Media (char *s, float a)    //base class cor
        {
            strcpy(title,s);
            price=a;
        }
        virtual void display()=0;  //pure virtual f
};
class Book: public Media
{
    int pages;
    public:
        //derived class constructor
        Book(char *s, float a, int p):Media(s,a)
        {
            pages=p;
        }
        void display();    //virtual function redefi
};
```



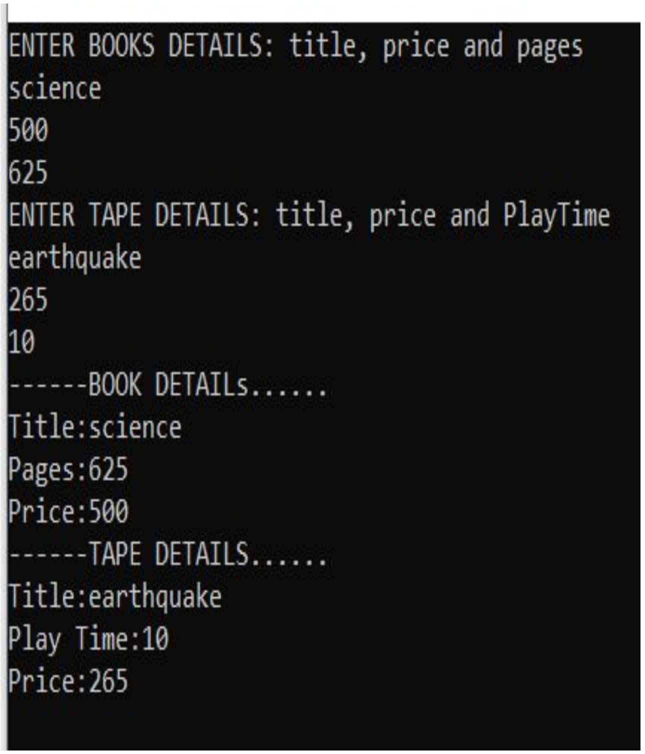
```
class Tape: public Media
{
    float time;
    public:
        //derived class constructor
        Tape(char *s, float a, float t):Media(s,a)
        {
            time=t;
        }
        void display();       //virtual function redefined
};
//explicit definitions of different functions
void Book::display()
{
    cout<<"Title:"<<title<<endl;
    cout<<"Pages:"<<pages<<endl;
    cout<<"Price:"<<price<<endl;
}
void Tape::display()
{
    cout<<"Title:"<<title<<endl;
    cout<<"Play Time:"<<time<<endl;
    cout<<"Price:"<<price<<endl;
}
```

5.7 Pure Polymorphism: Sample program





```cpp
main()
{
    char *title= new char[30];
    float price,time;
    int pages;
    //Book details
    cout<<"ENTER BOOKS DETAILS: title, price and pages"<<endl;
    cin>>title>>price>>pages;
    Book b(title,price,pages); //object of class Book
    //Tape details
    cout<<"ENTER TAPE DETAILS: title, price and PlayTime"<<endl;
    cin>>title>>price>>time;
    Tape t(title, price,time); //object of class Tape

    Media *list[2];           //base class array-pointer
    list[0]=&b;               //pointing to object b
    list[1]=&t;               //pointing to object t
    //displaying output
    cout<<"------BOOK DETAILs......"<<endl;
    list[0]->display();     //display books details
    cout<<"------TAPE DETAILS......"<<endl;
    list[1]->display();     //display tape details
}
```



```
ENTER BOOKS DETAILS: title, price and pages
science
500
625
ENTER TAPE DETAILS: title, price and PlayTime
earthquake
265
10
------BOOK DETAILs......
Title:science
Pages:625
Price:500
------TAPE DETAILS......
Title:earthquake
Play Time:10
Price:265
```

5.8 Type conversion

- If an expression contains different types of data like integer and float in left and right hand side, then the compiler automatically converts from one type to another type by applying type conversion rule provided by the compiler.
- The automatic type conversion means that the right side of the data is automatically converted to the left side type.
- This type of conversion rules cannot be applied in case of user-defined datatypes.
- In user-defined datatypes, we must define conversion routines ourselves.
- The possible types are:
  1. Conversion from basic type to another basic type
  2. Conversion from basic type to class type
  3. Conversion from class type to basic type
  4. Conversion from one class type to another class type

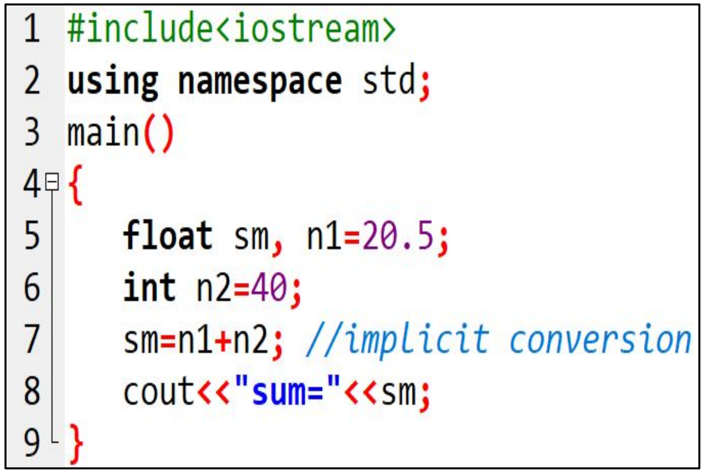
1. Basic type to Another Basic type

- Dataypes like int, float, double, long, long double, char etc are basic datatypes.
- We can convert one basic data type into another.
- There are two types of conversion: implicit and explicit.

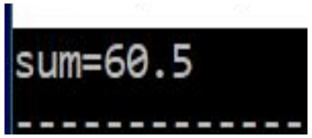
1. Basic type to Another Basic type

Implicit conversion:

- It is also known as automatic conversion, or promotion or widening.
- This does not require any operator and is performed by compiler itself.
- Eg:



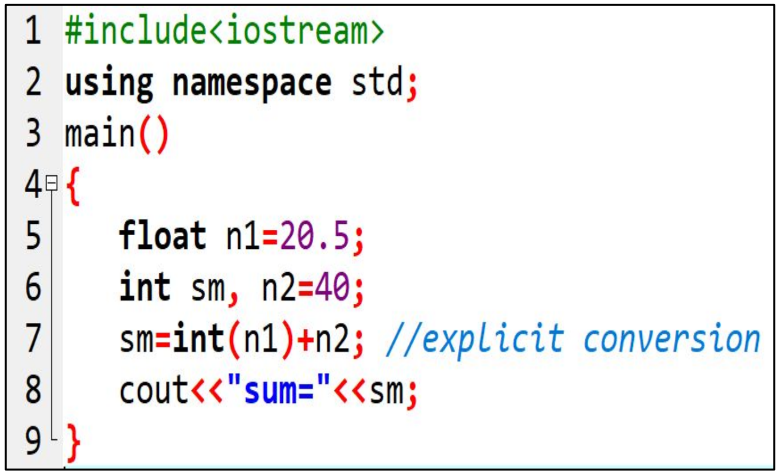
```
#include<iostream>
using namespace std;
main()
{
    float sm, n1=20.5;
    int n2=40;
    sm=n1+n2; //implicit conversion
    cout<<"sum="<<sm;
}
```



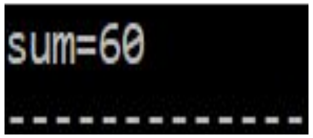
```
sum=60.5
-------------
```

Explicit conversion:

- This is also known as Type casting.
- The conversion of one datatype into another datatype is performed by programmer as per the need.
- Eg:



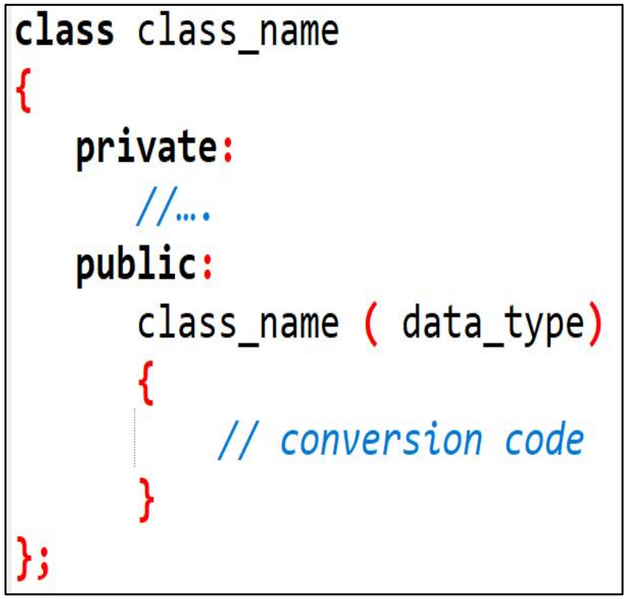
```
#include<iostream>
using namespace std;
main()
{
    float n1=20.5;
    int sm, n2=40;
    sm=int(n1)+n2; //explicit conversion
    cout<<"sum="<<sm;
}
```



```
sum=60
-------------
```

2. Basic type to Class type

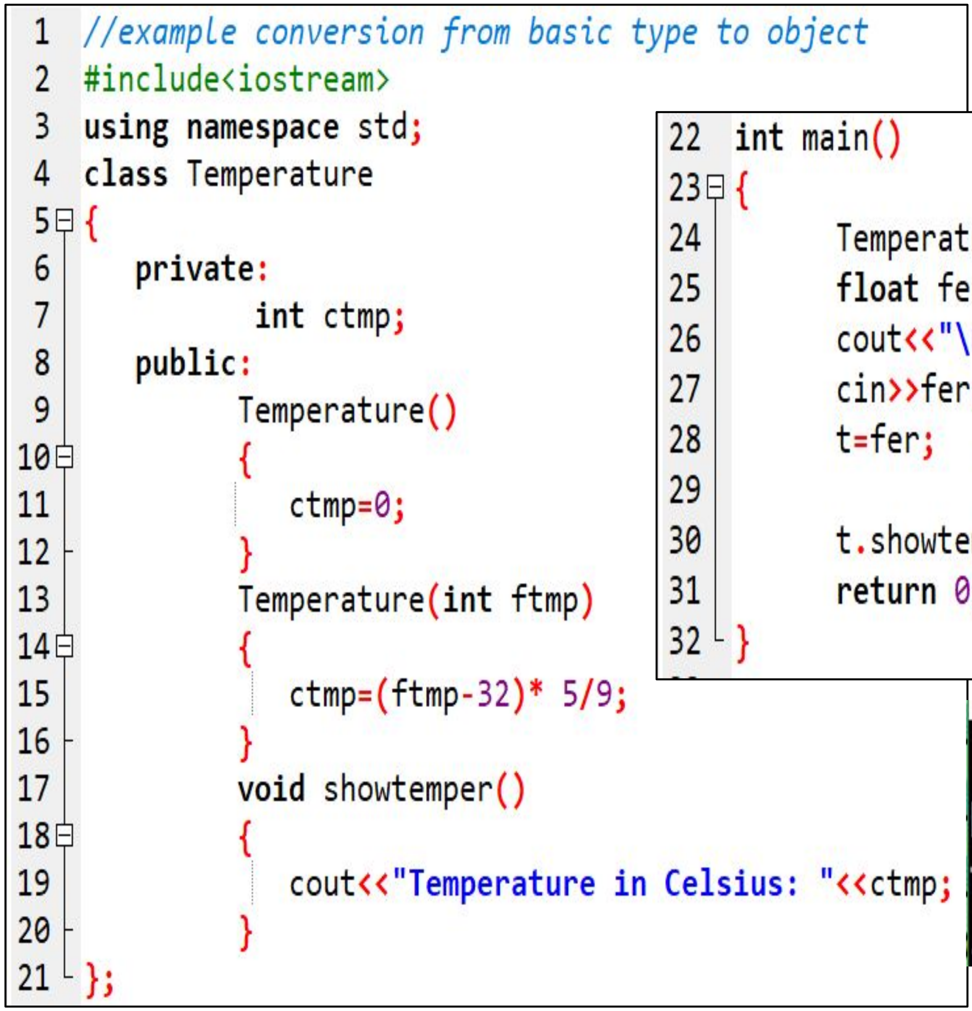
- The conversion from basic to user defined data types (class type) can be done using :
  - constructors with one argument of basic type.
- General form:



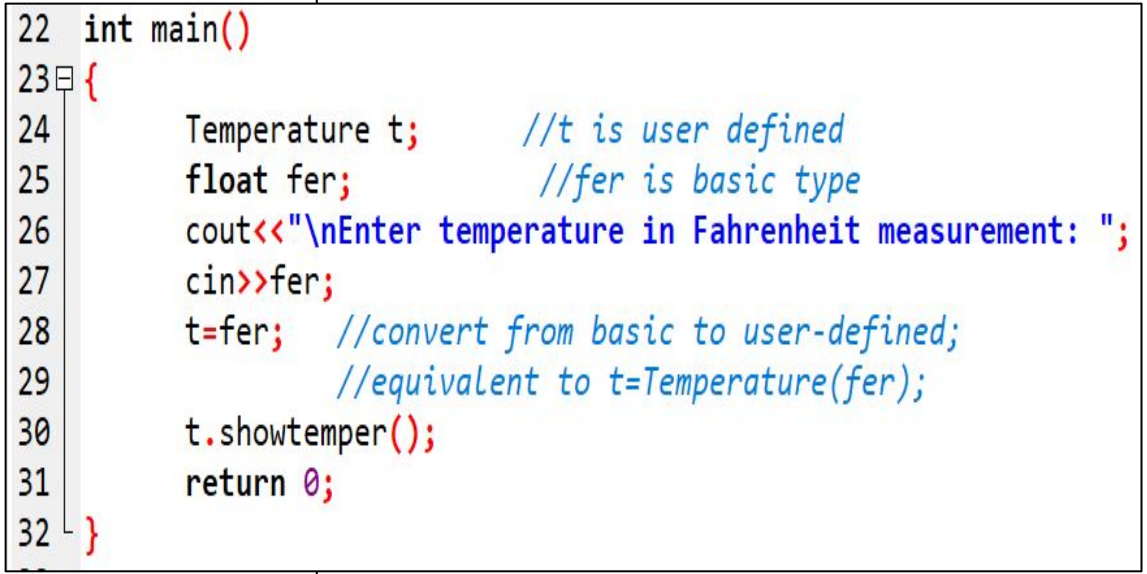
```
class class_name
{
   private:
      //….
   public:
      class_name ( data_type)
      {
         // conversion code
      }
};
```

2. Basic type to Class type : Sample program

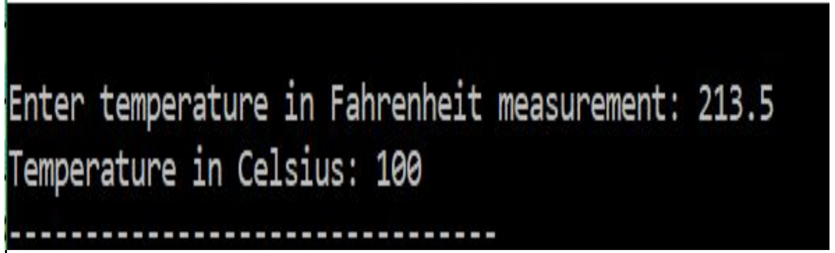
WAP to convert temperature in Fahrenheit into Celsius using type conversion from basic type to class type.



```cpp
//example conversion from basic type to object
#include<iostream>
using namespace std;
class Temperature
{
    private:
            int ctmp;
    public:
        Temperature()
        {
            ctmp=0;
        }
        Temperature(int ftmp)
        {
            ctmp=(ftmp-32)* 5/9;
        }
        void showtemper()
        {
            cout<<"Temperature in Celsius: "<<ctmp;
        }
};
```



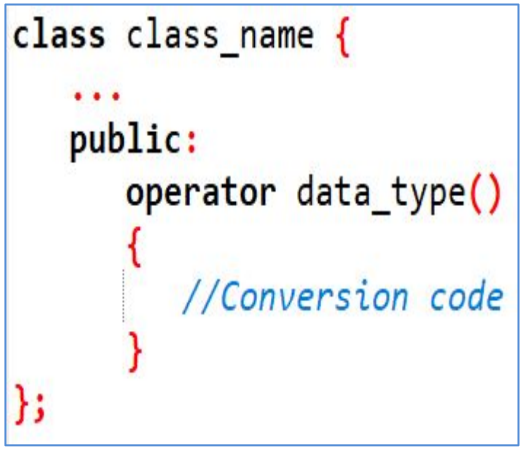
```cpp
int main()
{
        Temperature t;      //t is user defined
        float fer;           //fer is basic type
        cout<<"\nEnter temperature in Fahrenheit measurement: ";
        cin>>fer;
        t=fer;    //convert from basic to user-defined;
                  //equivalent to t=Temperature(fer);
        t.showtemper();
        return 0;
}
```



```
Enter temperature in Fahrenheit measurement: 213.5
Temperature in Celsius: 100
-----------------------------
```

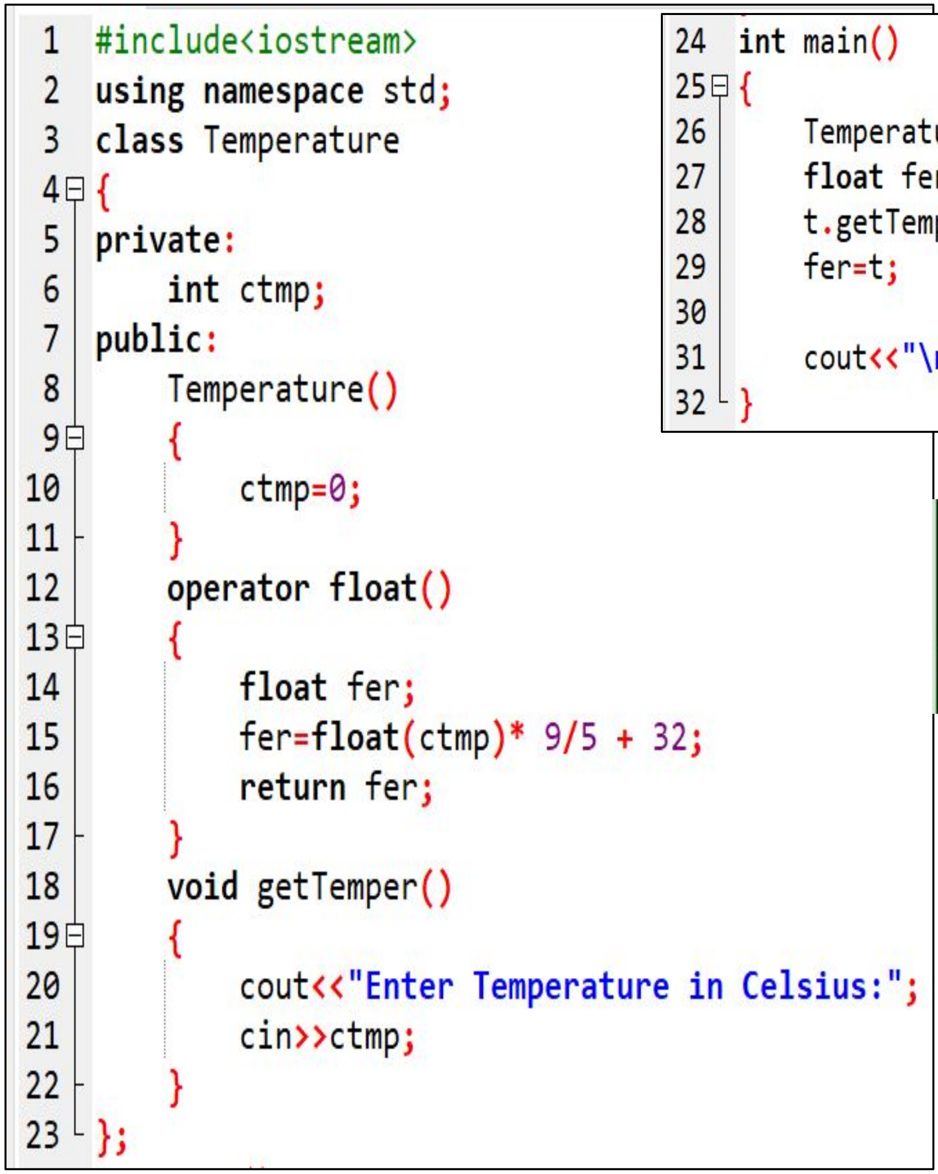
3. Class type to Basic type

- Conversion from Class type(user-defined type) to basic data type is done by overloading the casting operator of basic type as a member function.
- Operator function is defined as an overloaded basic data type which takes no arguments.
- Return type of operator is not specified because the casting operator function itself specifies the return type.
- The casting operator function should satisfy the following condition:
  - It must be a class member.
  - It must not specify any return type.
  - It must not have any arguments.
- Syntax

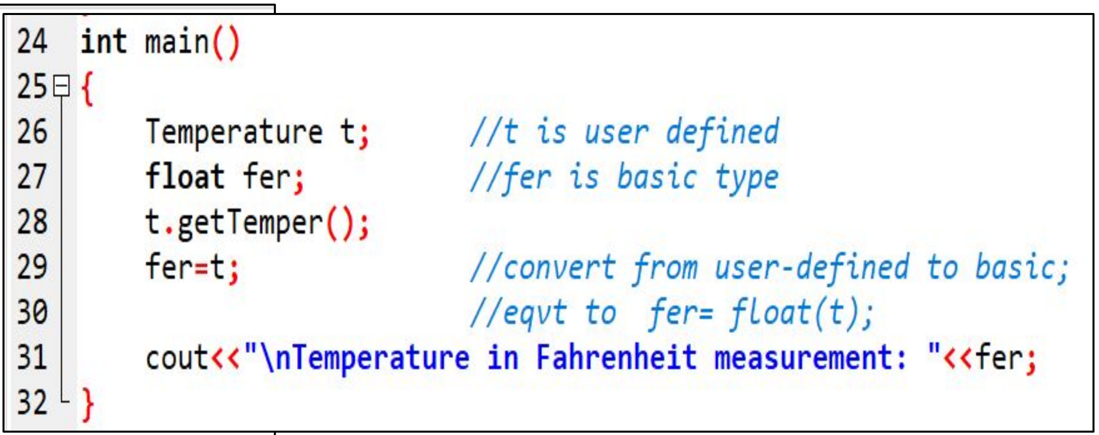


```
class class_name {
    ...
    public:
        operator data_type()
        {
            //Conversion code
        }
};
```

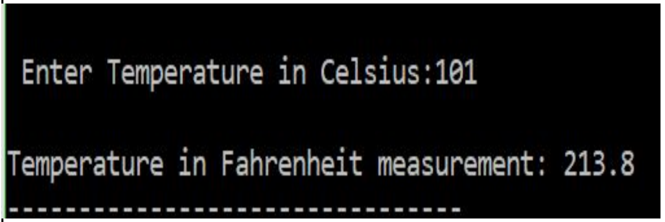
3. Class type to Basic type : Sample program



```cpp
#include<iostream>
using namespace std;
class Temperature
{
private:
    int ctmp;
public:
    Temperature()
    {
        ctmp=0;
    }
    operator float()
    {
        float fer;
        fer=float(ctmp)* 9/5 + 32;
        return fer;
    }
    void getTemper()
    {
        cout<<"Enter Temperature in Celsius:";
        cin>>ctmp;
    }
};
```



```cpp
int main()
{
    Temperature t;          //t is user defined
    float fer;              //fer is basic type
    t.getTemper();
    fer=t;                  //convert from user-defined to basic;
                            //eqvt to  fer= float(t);
    cout<<"\nTemperature in Fahrenheit measurement: "<<fer;
}
```



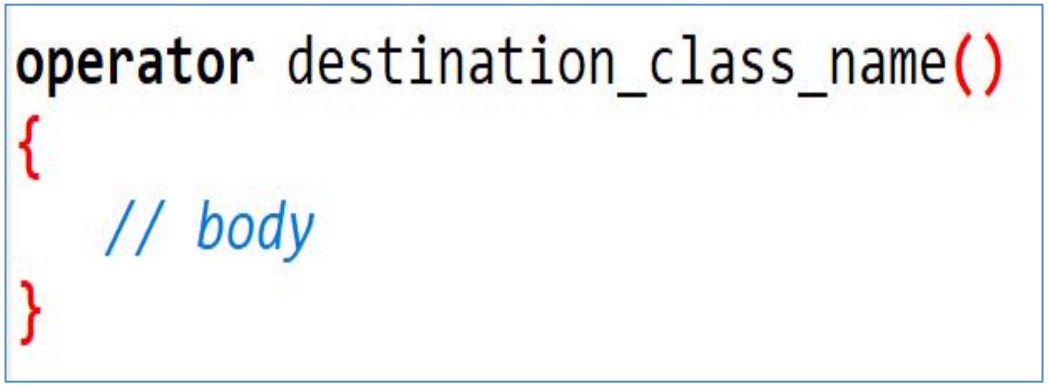
```

 Enter Temperature in Celsius:101

Temperature in Fahrenheit measurement: 213.8
--------------------------------
```

4. One ClassType to another ClassType

- The conversion form object of one class to object of another class can be performed using:
  - either one-argument constructor, or
  - a conversion function.
- The choice depends upon whether the conversion routine is declared in:
  a) the source class, or
  b) the destination class.

a) Routine in Source Object

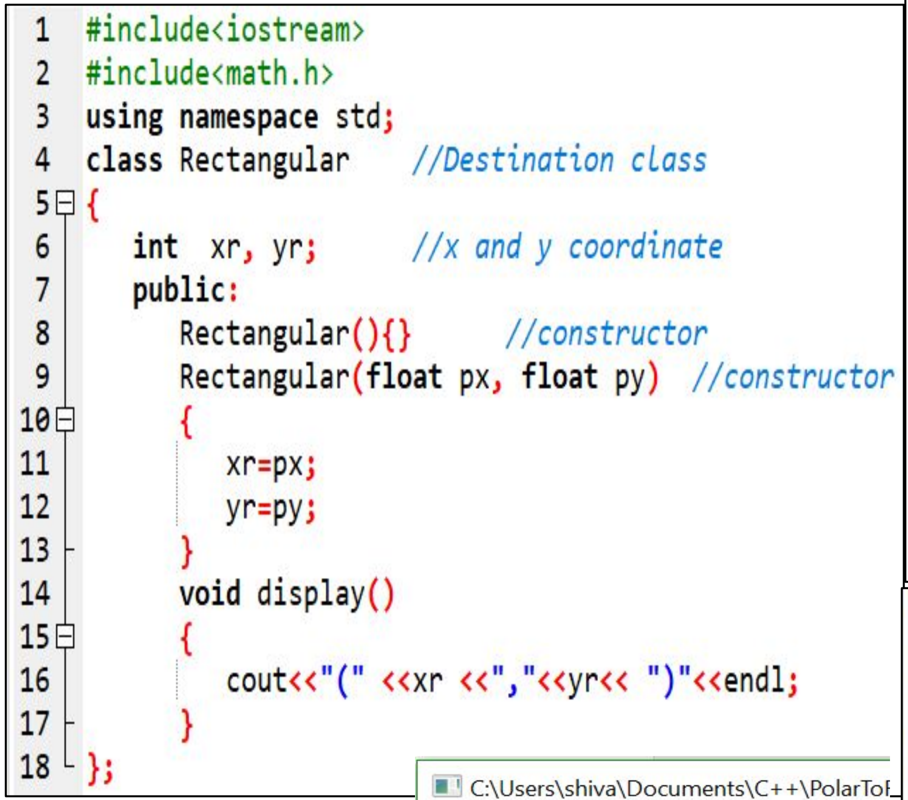
- When the conversion routine is in the source class, it is performed using a conversion function.
- The syntax of conversion function will be:



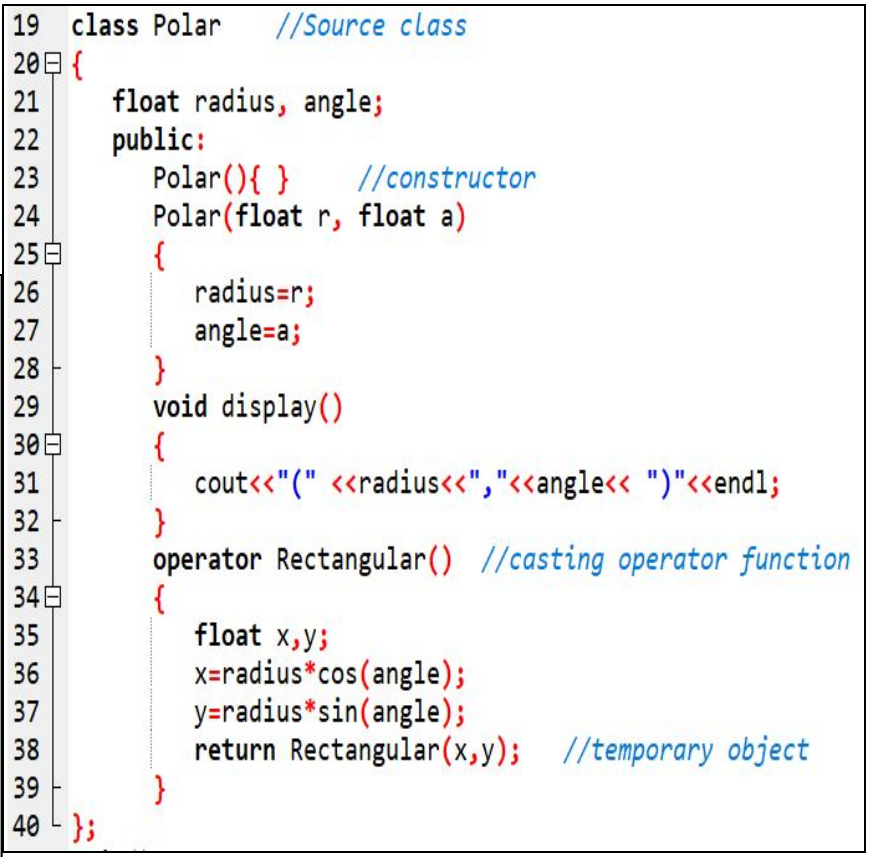
```
operator destination_class_name()
{
    // body
}
```

a) Routine in Source Object: Sample program

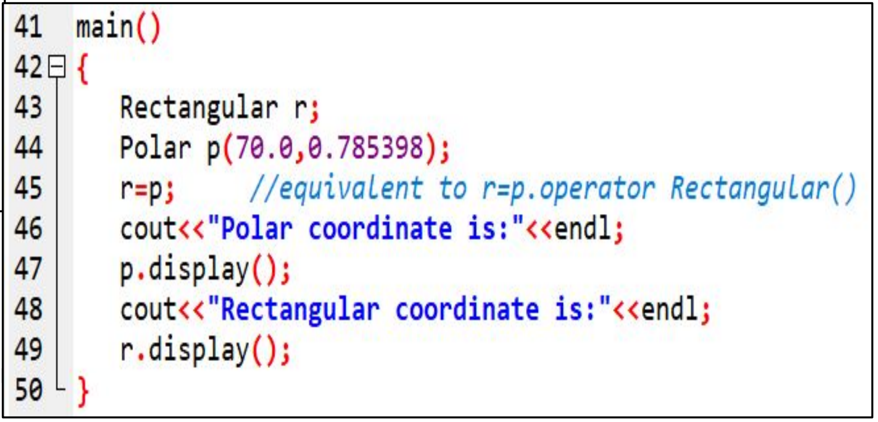
Define two classes Rectangular and Polar to represent the point in polar and rectangular form and use conversion routine to convert from one form to another(i.e convert object of class Polar into object of class Rectangular).



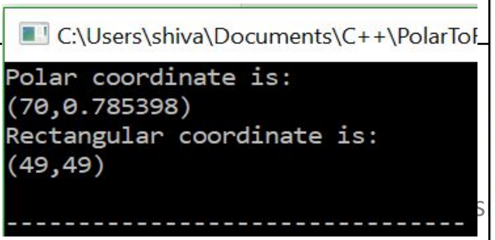
```cpp
#include<iostream>
#include<math.h>
using namespace std;
class Rectangular     //Destination class
{
    int  xr, yr;       //x and y coordinate
    public:
      Rectangular(){}        //constructor
      Rectangular(float px, float py)  //constructor
      {
         xr=px;
         yr=py;
      }
      void display()
      {
         cout<<"(" <<xr <<","<<yr<< ")"<<endl;
      }
};
```



```cpp
class Polar    //Source class
{
    float radius, angle;
    public:
      Polar(){ }     //constructor
      Polar(float r, float a)
      {
         radius=r;
         angle=a;
      }
      void display()
      {
         cout<<"(" <<radius<<","<<angle<< ")"<<endl;
      }
      operator Rectangular()  //casting operator function
      {
         float x,y;
         x=radius*cos(angle);
         y=radius*sin(angle);
         return Rectangular(x,y);   //temporary object
      }
};
```



```cpp
main()
{
    Rectangular r;
    Polar p(70.0,0.785398);
    r=p;      //equivalent to r=p.operator Rectangular()
    cout<<"Polar coordinate is:"<<endl;
    p.display();
    cout<<"Rectangular coordinate is:"<<endl;
    r.display();
}
```

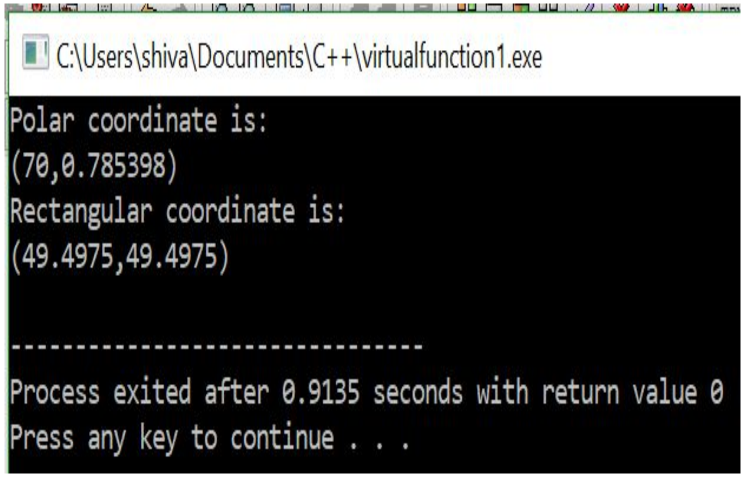


C:\Users\shiva\Documents\C++\PolarToF

```
Polar coordinate is:
(70,0.785398)
Rectangular coordinate is:
(49,49)

--------------------------------
```

a) Routine in Source Object: Sample program

- Here, in the above program, the data members in the *Polar* class is *float* type (i.e. Polar type).
- This is converted into *Rectangular* type (i.e. int type).
- If we had declared as ***int xr, yr;*** in the class Rectangular, our output would be.



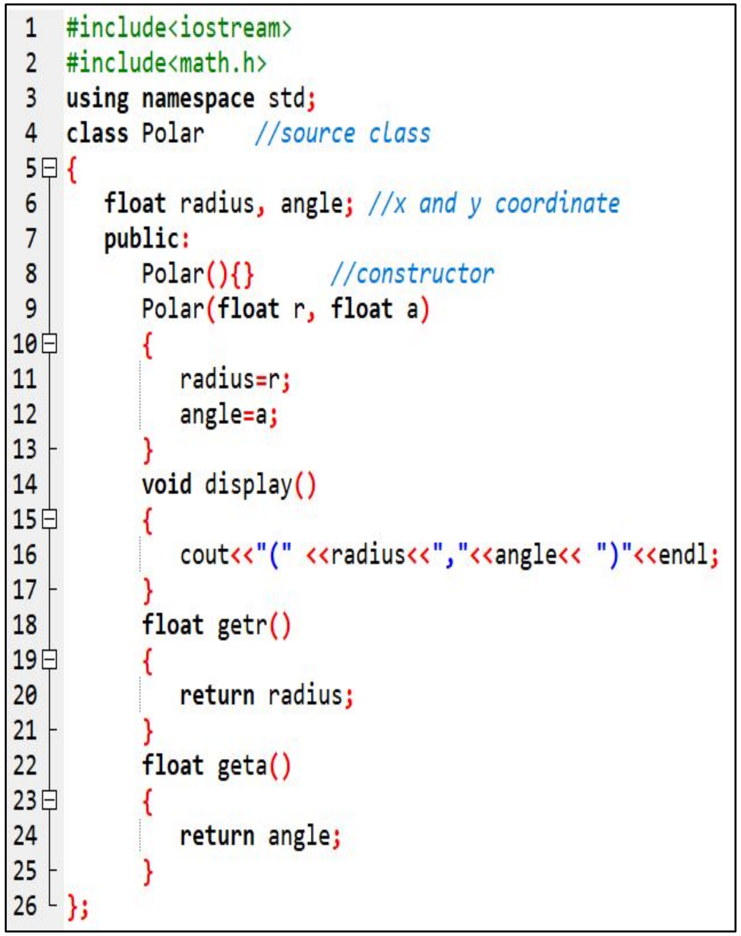
C:\Users\shiva\Documents\C++\virtualfunction1.exe

```
Polar coordinate is:
(70,0.785398)
Rectangular coordinate is:
(49.4975,49.4975)


--------------------------------

Process exited after 0.9135 seconds with return value 0
Press any key to continue . . .
```

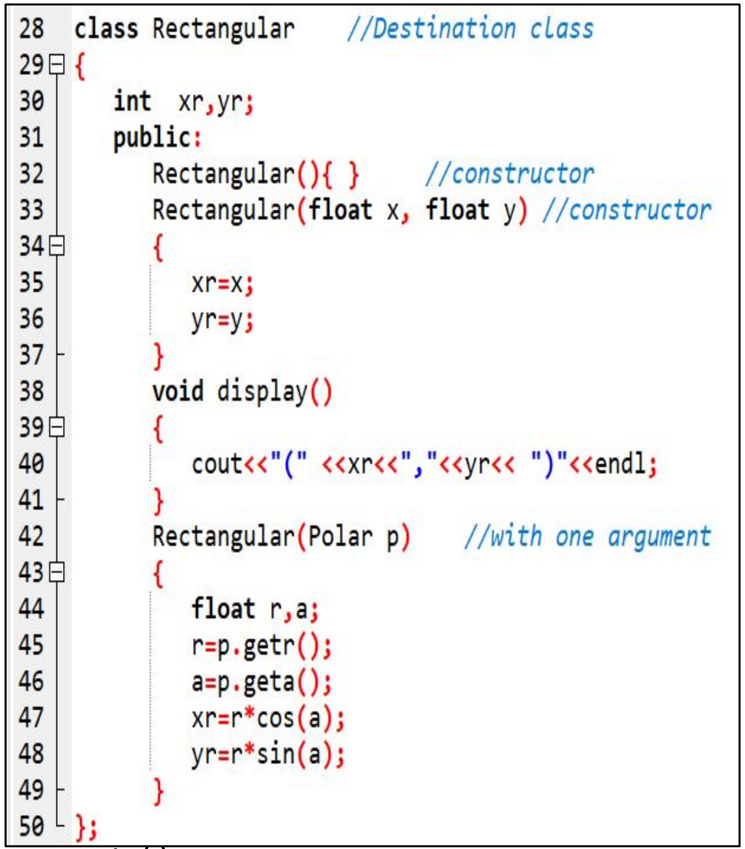
b) Routine in Destination Object

- When the conversion routine is in the destination class, we use a one-argument constructor.

b) Routine in Destination Object: Sample program

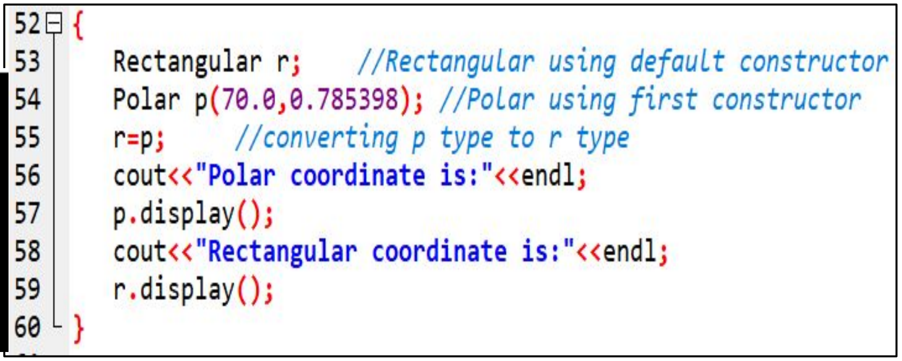


```cpp
#include<iostream>
#include<math.h>
using namespace std;
class Polar     //source class
{
    float radius, angle; //x and y coordinate
    public:
        Polar(){}       //constructor
        Polar(float r, float a)
        {
            radius=r;
            angle=a;
        }
        void display()
        {
            cout<<"(" <<radius<<","<<angle<< ")"<<endl;
        }
        float getr()
        {
            return radius;
        }
        float geta()
        {
            return angle;
        }
};
```

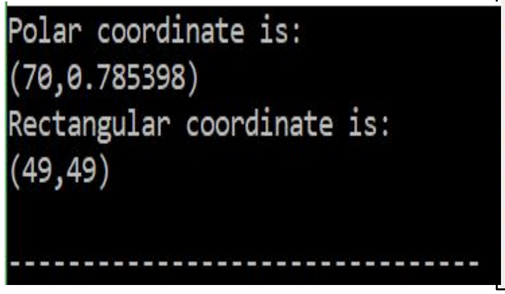


```cpp
class Rectangular    //Destination class
{
    int  xr,yr;
    public:
        Rectangular(){ }      //constructor
        Rectangular(float x, float y) //constructor
        {
            xr=x;
            yr=y;
        }
        void display()
        {
            cout<<"(" <<xr<<","<<yr<< ")"<<endl;
        }
        Rectangular(Polar p)     //with one argument
        {
            float r,a;
            r=p.getr();
            a=p.geta();
            xr=r*cos(a);
            yr=r*sin(a);
        }
};
```

main( )



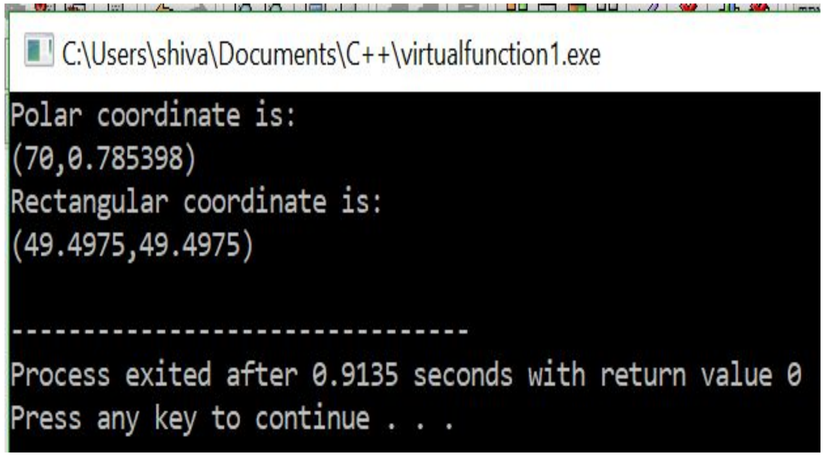
```cpp
{
    Rectangular r;     //Rectangular using default constructor
    Polar p(70.0,0.785398); //Polar using first constructor
    r=p;      //converting p type to r type
    cout<<"Polar coordinate is:"<<endl;
    p.display();
    cout<<"Rectangular coordinate is:"<<endl;
    r.display();
}
```



```
Polar coordinate is:
(70,0.785398)
Rectangular coordinate is:
(49,49)


--------------------------------
```

b) Routine in Destination Object: Sample program

- Here, in the above program, the data members in the *Polar* class is *float* type (i.e. Polar type).
- This is converted into *Rectangular* type (i.e. int type).
- If we had declared as ***float xr, yr;*** in the class Rectangular, our output would be.



C:\Users\shiva\Documents\C++\virtualfunction1.exe

```
Polar coordinate is:
(70,0.785398)
Rectangular coordinate is:
(49.4975,49.4975)

--------------------------------
Process exited after 0.9135 seconds with return value 0
Press any key to continue . . .
```

5.9 This pointer

- The member function of every object have access to a pointer named ***this***, which point to the object itself.
- When we call a member function, it comes into existence with the value of ***this*** set to the address of the object for which it was called.
- The ***this*** pointer can be treated like any other pointer to an object.

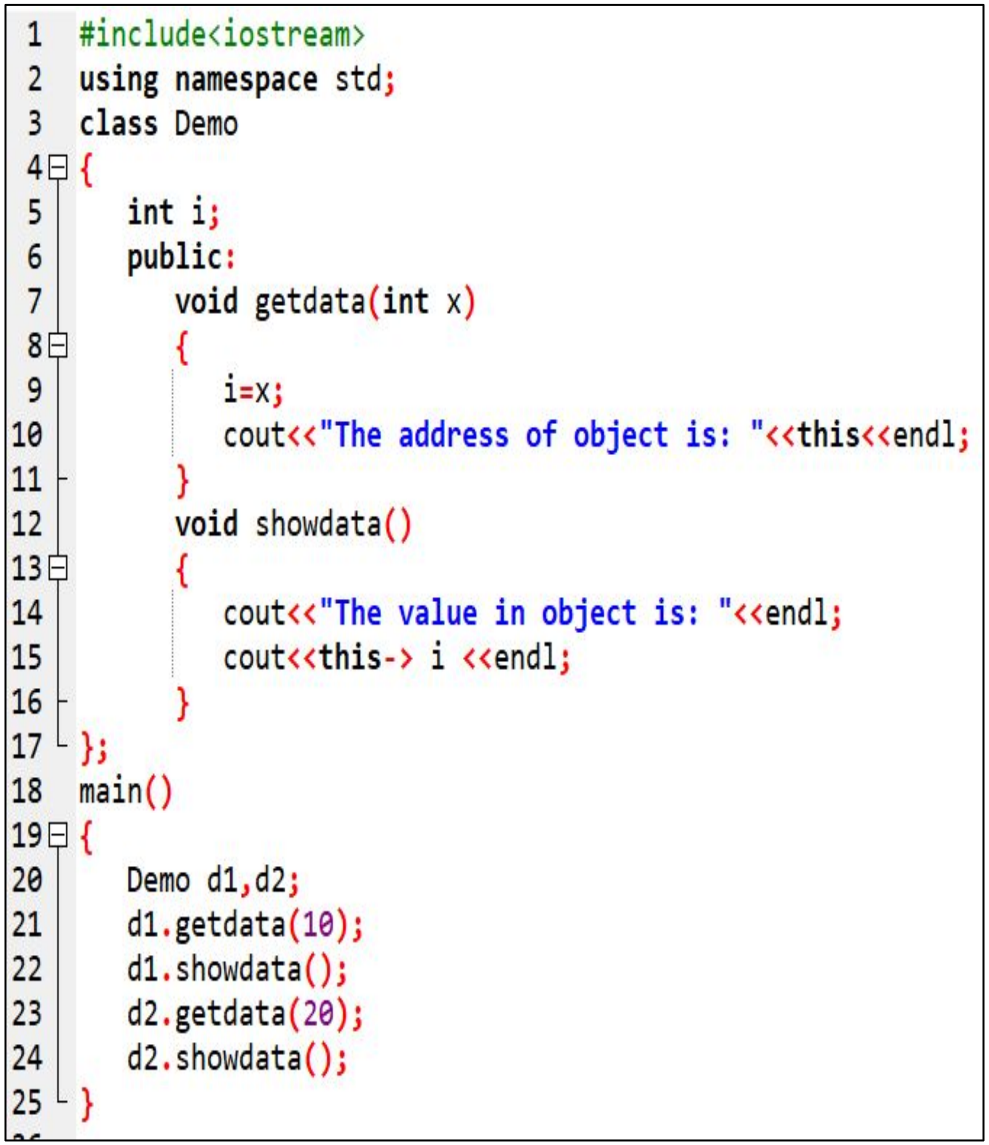
Characteristics of *this* pointer:

- ***this*** pointer stores the address of invoking object of the class.
- ***this*** pointers are not accessible for static member functions.
- ***this*** pointers are not modifiable.

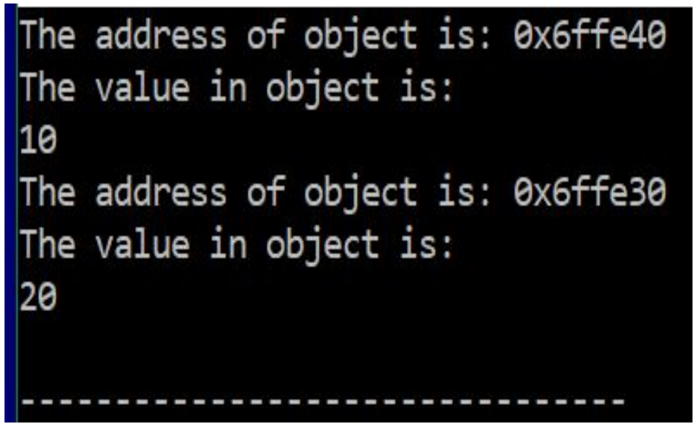
Uses of *this* pointer:

- To specify memory address of an object.
- To access data members.

5.9 This pointer : Sample program



```cpp
#include<iostream>
using namespace std;
class Demo
{
    int i;
    public:
        void getdata(int x)
        {
            i=x;
            cout<<"The address of object is: "<<this<<endl;
        }
        void showdata()
        {
            cout<<"The value in object is: "<<endl;
            cout<<this-> i <<endl;
        }
};
main()
{
    Demo d1,d2;
    d1.getdata(10);
    d1.showdata();
    d2.getdata(20);
    d2.showdata();
}
```



```
The address of object is: 0x6ffe40
The value in object is:
10
The address of object is: 0x6ffe30
The value in object is:
20

--------------------------------
```

5.10 Object Pointer

- Just like other pointers, the object pointer are declared by placing in front of an object's name.
- General syntax is :

```
Class_name  * ObjectPointerName;
```

- Eg:

```
Demo * p;
```

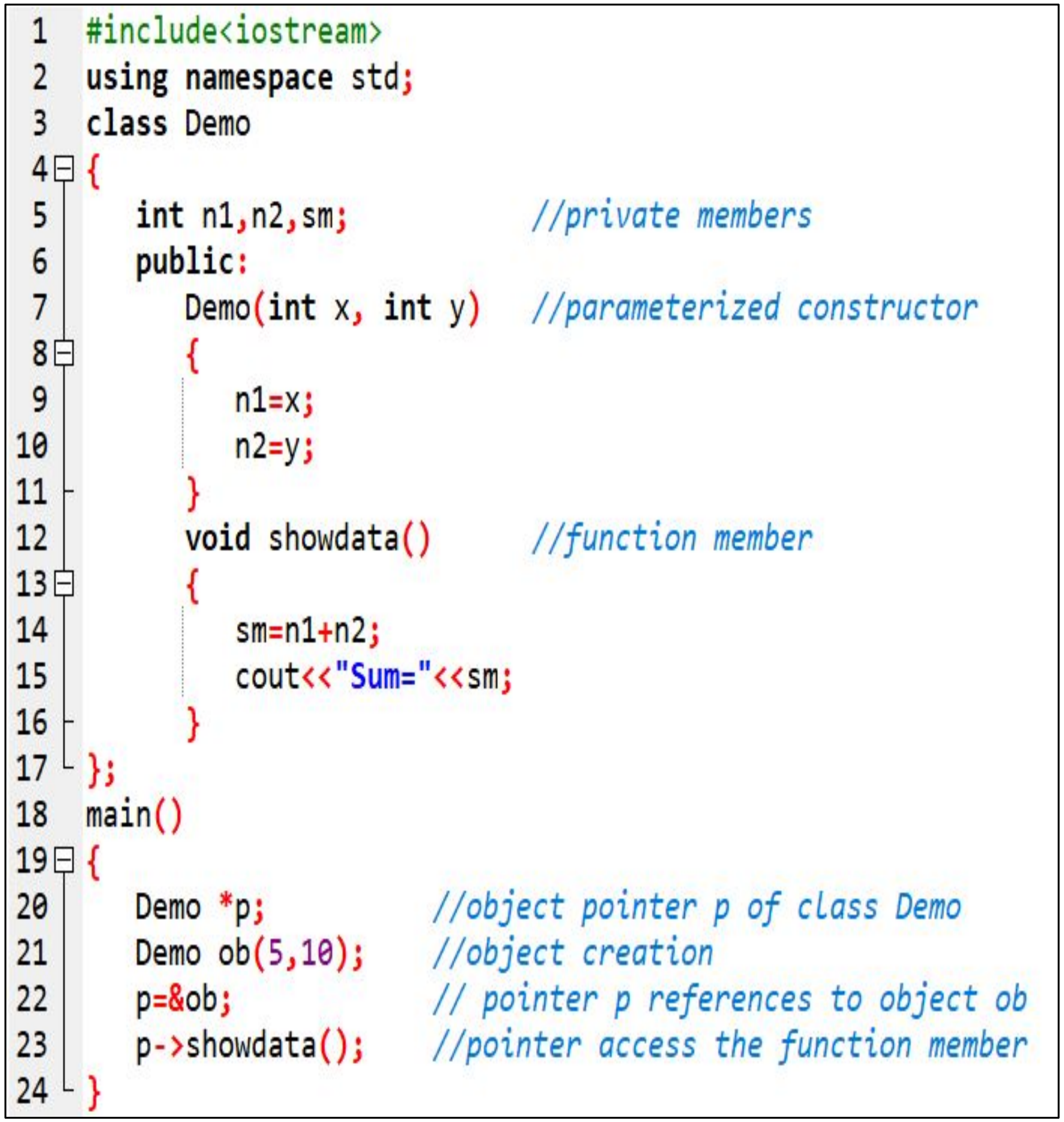
- The object pointer should reference the object to which it has to point.
- Eg:

```
p=&ob;
```

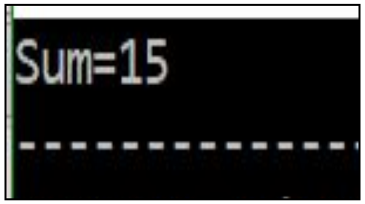
- When accessing members of a class using an object pointer the Arrow Operator (->) is used instead of the Dot (.) operator.
- Eg:

```
p-> showdata();
```

5.10 Object Pointer : Sample program



```cpp
#include<iostream>
using namespace std;
class Demo
{
    int n1,n2,sm;           //private members
    public:
        Demo(int x, int y)  //parameterized constructor
        {
            n1=x;
            n2=y;
        }
        void showdata()     //function member
        {
            sm=n1+n2;
            cout<<"Sum="<<sm;
        }
};
main()
{
    Demo *p;          //object pointer p of class Demo
    Demo ob(5,10);    //object creation
    p=&ob;            // pointer p references to object ob
    p->showdata();    //pointer access the function member
}
```



```
Sum=15
--------------
```

5.11 Polymorphic variable

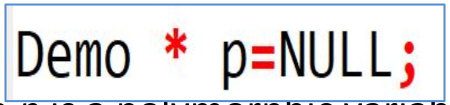
- Polymorphic variables are variables that can refer to a variety of objects of different type during the execution of a program.
- The types to which a polymorphic variable refers are not necessarily the type of the polymorphic variable itself.
- It has two types:
  a) Static type
  b) Dynamic type

a) Static Type

- The type used in the declaration of a polymorphic variable is called the static type.
- In C++, a polymorphic variable is declared in several ways, including:

i) As a pointer to an object of related type,

- Eg:

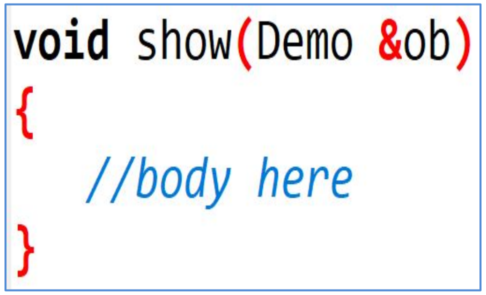


```
Demo * p=NULL;
```

- Here *p* is a polymorphic variable.

ii) As a by-reference parameter to an object of related type.

- Eg:

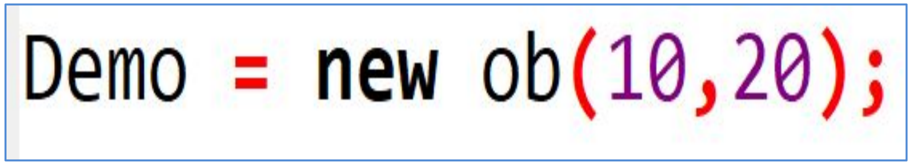


```
void show(Demo &ob)
{
    //body here
}
```

- Here *ob* is a polymorphic variable.

b) Dynamic Type

- The type of the object referred by a polymorphic variable is called dynamic type.
- The dynamic type need not to be the same as the static type.
- The dynamic type may depend upon the execution path through the program.
- Eg:



```
Demo = new ob(10,20);
```

- Here ***ob*** is a polymorphic variable.

End of Chapter 4