Polymorphism

Polymorphism

- Introduction to pointers: Pointers to objects, pointer to derived class object
- Compile time polymorphism: Review of Function Overloading and Operator overloading
- Run time polymorphism: virtual functions, pure virtual functions, abstract class, virtual constructors & destructors

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access (->) operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

```
#include <iostream>
 using namespace std;
class Box {
   public:
      // Constructor definition
Box(double l = 2.0, double b = 2.0,
      double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }
private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
int main(void) {
   Box Box1(3.3, 1.2, 1.5);
   Box Box2(8.5, 6.0, 2.0);
   Box *ptrBox; // Declare pointer to a class.
   // Save the address of first object
   ptrBox = &Box1;
   // Now try to access a member using
member access operator
   cout << "Volume of Box1: " << ptrBox-
>Volume() << endl;
// Save the address of first object
   ptrBox = &Box2;

   // Now try to access a member using
member access operator
   cout << "Volume of Box2: " << ptrBox-
>Volume() << endl;

   return 0;
}
```

output:

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

Pointers to base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

```
#include <iostream>
using namespace std;
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};


class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};
```

```
class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0; }
```

output:

```
20 10
```

Explanation:

Function main declares two pointers to Polygon (named ppoly1 and ppoly2). These are assigned the addresses of rect and trgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing ppoly1 and ppoly2 (with *ppoly1 and *ppoly2) is valid and allows us to access the members of their pointed objects.

For example, the following two statements would be equivalent in the previous example:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Demonstrate pointer to derived class.

```
#include <iostream>
using namespace std;
class BaseClass {
  int x;
public:
  void setx(int i) {
     x = i;
  }
  int getx() {
     return x;
  }
};
class DerivedClass : public BaseClass {
  int y;
public:
  void sety(int i) {
     y = i;
  }
 int gety() {
     return y;
 }
};
int main()
{
  BaseClass *p;
//pointer to BaseClass type
  BaseClass baseObject;
//object of BaseClass
  DerivedClass derivedObject;
//object of DerivedClass
  p = &baseObject;
// use p to access BaseClass object
  p->setx(10);
 // access BaseClass object
  cout << "Base object x: " << p->getx() <<
'\n';
```

```
  p = &derivedObject;
 // point to DerivedClass object
  p->setx(99);
// access DerivedClass object


  derivedObject.sety(88);
   // can't use p to set y, so do it directly
  cout << "Derived object x: " << p->getx()
<< '\n';
  cout << "Derived object y: " <<
derivedObject.gety() << '\n';
  return 0;
}
```

output:

```
Base object x:10
Derived object x:99
Derived object y:88
```

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So a same person posses have different behavior in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

1. Compile time Polymorphism
2. Runtime Polymorphism

- Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

- Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

- Operator Overloading: C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Compile time polymorphism is also known as static binding or early binding.

```cpp
 // C++ program for function overloading
 #include<iostream>
using namespace std;
class Demo{
    public:
         // function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
  // function with same name but 1 double
parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    } // function with same name and 2 int
parameters
    void func(int x, int y)
    {
cout << "value of x and y is " << x << ", "
<< y << endl;
}
};
 int main() {
      Demo obj1;

    // Which function is called will depend
on the parameters passed
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);
      // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Output:

```
value of x is 7
value of x is 9.132
value of x and y is    85, 64
```

Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

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

It is also known as dynamic binding or late binding.

```
// C++ program for function overriding
 #include <bits/stdc++.h>
using namespace std;
class Parent
{
    public:
    void print()
    {
        cout << "The Parent print function
was called" << endl;
    }
};
class Child : public Parent
{
    public:
    // definition of a member function
already present in Parent
void print()
    {
        cout << "The child print function was
called" << endl;
    }
};
 int main()
{
     Parent obj1;
      Child obj2 = new Child();

// obj1 will call the print function in Parent
obj1.print();
    // obj2 will override the print function in
Parent and call the print function in Child
    obj2.print();
    return 0;
}
```

Output:

```
The Parent print function was called
The child print function was called
```

Virtual member functions:

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword:

Syntax:

```
virtual return_type funtionName (arguments){}
```

```
// virtual members
#include <iostream>
using namespace std;
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};


class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};
class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}output:20 10 0
```

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

They are mainly used to achieve Runtime polymorphism

- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

- They Must be declared in public section of class.
- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate concept of
Virtual Functions
#include<iostream>
using namespace std;
class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};
```

```
class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};
```

```
 int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //virtual function, binded at runtime
   bptr->print();

   // Non-virtual function, binded at compile
time
   bptr->show();
}
```

Output:

```
print derived class
show base class
```

Explanation:

- Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer ‘bptr’ contains the address of object ‘d’ of derived class.

- Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be binded at run-time (output is print derived class as pointer is pointing to object of derived class) and show() is non-virtual so it will be binded during compile time(output is show base class as pointer is of base type).

NOTE: If we have created virtual function in base class and it is being overrided in derived class then we don’t need virtual keyword in derived class, functions are automatically considered as virtual functions in derived class.

Working of virtual functions(concept of VTABLE and VPTR)

If a class contains a virtual function then compiler itself does two things:

1)If object of that class is created then a virtual pointer(VPTR) is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

2)Irrespective of object is **created** or not, a static array of function pointer called VTABLE where each cell contains the address of each virtual function contained in that class.

Consider the example below:

```
// CPP program to illustrate
// working of Virtual Functions
#include<iostream>
using namespace std;
class base
{
public:
    void fun_1() {
               cout << "base-1\n";
 }
 virtual void fun_2() {
              cout << "base-2\n";
}
    virtual void fun_3() {
              cout << "base-3\n";
 }
    virtual void fun_4() {
              cout << "base-4\n";
}
};
```

```
class derived : public base
{
public:
    void fun_1() {
              cout << "derived-1\n";
}
    void fun_2() {
              cout << "derived-2\n";
 }
    void fun_4(int x) {
cout << "derived-2\n";
}
};
```

```
int main()
{
    base *p;
    derived obj1;
    p = &obj1;
    // Early binding because fun1() is non-
virtual in base
    p->fun_1();
    // Late binding (RTP)
    p->fun_2();
    // Late binding (RTP)
    p->fun_3();
    // Late binding (RTP)
    p->fun_4();
    // Early binding but this function call is
    // illegal(produces error) becasue
//pointer is of base type and function is of
    // derived class
    //p->fun_4(5); //illegal declaration
produces error
}
```

Output:

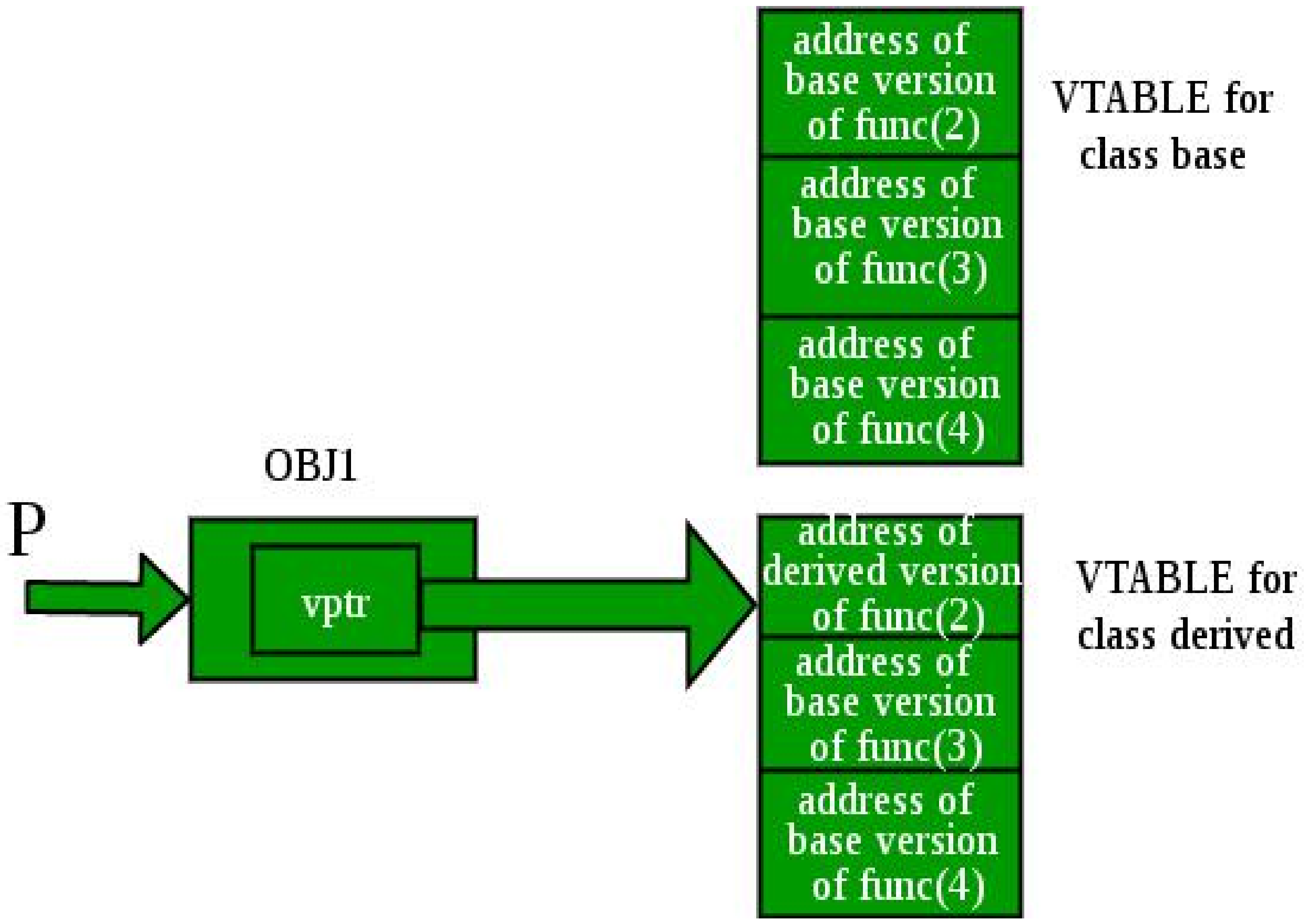
```
base-1
derived-2
base-3
base-4
```

Explanation:

Initially, we create pointer of type base class and initialize it with the address of derived class object. When we create an object of derived class, compiler creates a pointer as a data member of class containing the address of VTABLE of derived class.

Similar concept of Late and Early Binding is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overrided in derived class so derived class version is called, fun_3() is not overrided in derived class and is virtual function so base class version is called, similarly fun_4() is not overrided so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different.


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

Pure Virtual Functions and Abstract Classes

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

```
// An abstract class syntax
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

   /* Other members */
};
```

A pure virtual function is implemented by classes which are derived from a Abstract class..

Following is a simple example to demonstrate the same

```
#include<iostream>
using namespace std;
class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};
// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};
```

```
 int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output:

```
fun() called
```

1) A class is abstract if it has at least one pure virtual function.

In the following example, Test is an abstract class because it has a pure virtual function show().

```
// pure virtual functions make a class abstract
#include<iostream>
using namespace std;
class Test
{
  int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};
int main(void)
{
    Test t;
    return 0;
}
```

Output:

Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the function show() is pure virtual functions.

2) We can have pointers and references of abstract class type.

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};
 int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Output:

```
In Derived
```

3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
 class Derived : public Base { };

int main(void)
{
  Derived d;
  return 0;
}
```

output:

```
Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived'  because the following virtual functions are pure within
'Derived': virtual void Base::show()
```

4) An abstract class can have constructors.

For example, the following program compiles and runs fine.

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

// An abstract class with constructor
class Base
{
protected:
   int x;
public:
  virtual void fun() = 0;
  Base(int i) { x = i; }
};
class Derived: public Base
{
    int y;
public:
    Derived(int i, int j):Base(i) { y = j; }
    void fun() { cout << "x = " << x << ", y = "
<< y; }
};

int main(void)
{
    Derived d(4, 5);
    d.fun();
    return 0;
}
```

Output:

```
x = 4, y = 5
```

virtual constructors & destructors

Why don't we have virtual constructors?

- A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only any interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

- If you have virtual function, it means that this member function can be redefined by derived class. When we have a virtual function in a class, it will have vtable (virtual table). This vtable will have the address of the virtual function which is defined in derived.

- Now the point is, how you invoke the virtual function which is defined in derived class with the help of object of the class?
- Once you create an object of class, vptr (virtual pointer) will be created. Which inturn will be pointing to the base address of the vtable. So now it is clear that you can invoke the virtual funtion of derived class only with an object that has created.

- Now coming to virtual constructor, If we make constructor as virtual in base, it means that it could be redefined in derived. Keep in mind that constructor is invoked during object creation (object is not created yet. still it is in the status "creating". Object will create only after executing constructor part code).

- Assume you are trying to create object of the class which has virtual constructor. During this process constructor of the class will be invoked. It looks for virtual keyword. Now it tries to look for virtual constructor in derived. But not possible because there is no vptr and no vtable avaibale at this point of time. So, when object is not created, then there is no vptr. If no vptr for this object, then how the consturtor of derived is invoked?. No address of this construtor will available in vtable. Hence there is no point in having virtual constructor.

Virtual Destructor

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  return 0;
}
```

output:

```
Constructing base
Constructing derived
Destructing base
```

Note:

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
For example,

```
// A program with virtual destructor
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
 class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
```

```
 int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

This Pointer:

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

1)Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

```
#include <iostream>
 using namespace std;
class Box {
   public:
      Box(double l = 2.0, double b = 2.0,
            double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }
int compare(Box box) {
         return ((this->Volume()) >
(box.Volume()));
      }
private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

```
int main(void) {
   Box Box1(3.3, 1.2, 1.5);
 // Declare box1
   Box Box2(8.5, 6.0, 2.0);
// Declare box2
   if(Box1.compare(Box2)) {
      cout << "Box2 is smaller than Box1" <<endl;
   }
else {
  cout << "Box2 is equal to or larger than Box1" <<endl;
   }
   return 0;
}
```

Output:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than
Box1
```

2) When local variable's name is same as member's name

```
#include<iostream>
using namespace std;
/* local variable is same as a member's
name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve
the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x <<
endl; }
};
```

```
int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Output:

```
 x = 20
```

Note:

For constructors, initializer list can also be used when parameter name is same as member's name.

3) Reference to the calling object can be returned

```
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

3) To return reference to the calling object

When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
#include<iostream>
using namespace std;
class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0)
{
this->x = x;
this->y = y;
}
  Test &setX(int a) {
 x = a;
return *this;  }
Test &setY(int b) {
 y = b;
return *this;
 }
void print() {
cout << "x = " << x << " y = " << y << endl;
}
};
int main()
{
  Test obj1(5, 5);
   // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
   obj1.setX(10).setY(20);
   obj1.print();
  return 0;
}
```

Output:

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

Example 2: function chaining calls using this pointer

```
#include <iostream>
using namespace std;
class Demo {
private:
  int num;
  char ch;
public:
  Demo &setNum(int num){
    this->num =num;
    return *this;
  }
  Demo &setCh(char ch){
    this->num++;
    this->ch =ch;
    return *this;
  }
void displayMyValues(){
    cout<<num<<endl;
    cout<<ch;
  }
};
int main(){
  Demo obj;
  //Chaining calls
  obj.setNum(100).setCh('A');
  obj.displayMyValues();
  return 0;
}
```

Output:

```
101
A
```