Object Oriented Programming Using Cpp

Topic:

Run Time Polymorphism

9.3 Run Time Polymorphism

9.3.1 Method Overriding

We know that the process of inheritance normally does not allow use of member function inside base class. However, in some circumstances base class may have member functions. In C++, when member function of a base class is redefined inside the derived class, the process is known as method overriding. For, method overriding, it is very important that function header in the base class and derived class must be same.

Program 63: Write a program in C++ to illustrate method overriding.

Code:

```
#include<iostream.h>
#include<conio.h>
class ONE
{
        int n1;
    public:
        void fun()
        {
            n1=5;
            cout<<"n1="<<n1;
        }
};
class TWO:public ONE
{
        int n2;
    public:
        void fun()
        {
            n2=10;
```

```
                cout<<"n2="<<n2<<endl;
            }
};
void main()
{
    clrscr();
    TWO ob1;
    ob1.fun();
    getch();
}
```

O/P:

n2=10

Explanation: In the above example, class TWO is derived from class ONE. So, ONE acts as base class and TWO acts as derived class. Now, class ONE contains a member function show() which is again redefined inside the derived class TWO. This process is known as method overriding. In main function, since inheritance is used object declaration is done for derived class and we can see that the object executes the member function of derived class.

9.3.2 Binding in C++

C++ supports a mechanism known as binding which links two piece codes written at different locations. Formally binding can be defined as "The process of establishing the link between function call and function definition." Binding can be of two types known as static binding and dynamic binding.

9.3.2.1 Static Binding

It is also known as compile time or early binding. In this method, the decision to establish the link between the function call and function definition is made at compile time. Now when binding takes place, a new type of object is used known as object pointer.

Object pointer: An object pointer is a pointer which refers to an object. Object pointer may be of base class as well as of derived class. Most of the time, object pointer of base class is used because it can refer to base class object as well as derived class object. On the other hand, object pointer of derived class can access only derived class objects.

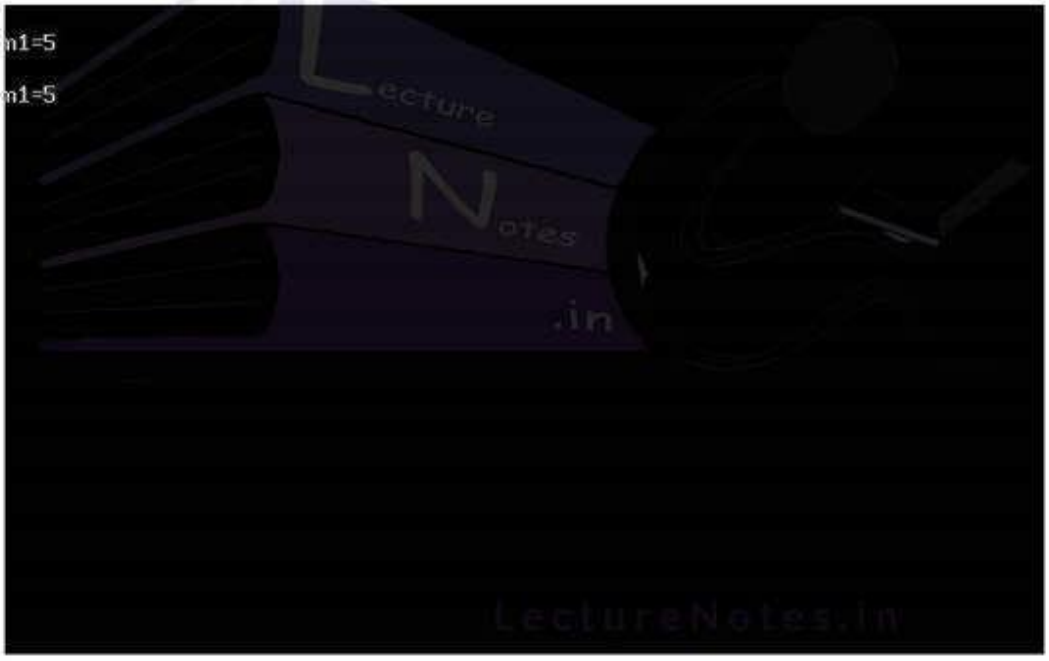
Program 64: Write a program in C++ to show the effect of static binding.

Code:

```
#include<iostream.h>
#include<conio.h>
class ONE
{
        int n1;
    public:
        void fun()
        {
            n1=5;
            cout<<endl<<"n1="<<n1<<endl;
        }
};
class TWO:public ONE
{
        int n2;
    public:
        void fun()
        {
            n2=10;
            cout<<endl<<"n2="<<n2<<endl;
        }
};
void main()
{
    clrscr();
```

```
    ONE *ptr;          //Base class object pointer
    ONE ob1;
    ptr=&ob1;
    ptr->fun();
    TWO ob2;
    ptr=&ob2;
    ptr->fun();
    getch();
}
```

O/P:



Explanation: In the above program, class ONE contains an integer data member n1 and a member function fun(). Now, class TWO is derived from class ONE and it also contains a data member n2 and the base class member function fun() is redefined in derived class. Base class object pointer ptr is used which at one instance refers to base class object ob1 and in another instance it refers to derived class object ob2. However, during both function calls system executes the base class member function. This is due to static binding where the decision to call the function is made at compile time. Now, at compile time system only declares the object pointer. It

does not get sufficient time for initialization which happens during run time. So, base class member function is called at both the locations.

9.3.2.2 Dynamic Binding

It is also known as run time or late binding. In this method, the decision to establish the link between the function call and function definition is made at run time i.e. the binding is delayed till run time. This is achieved using a new mechanism known as "Virtual Function".

Virtual Function: A function preceded by **virtual** keyword while defining is known as virtual function. Generally, virtual functions are associated with base class. The virtual functions of base class must be redefined inside the derived class.

Rules for virtual function

- Virtual function must not be static and it must be member of a class i.e. A friend function can't be virtual.
- Virtual functions can be accessed only using object pointers.
- Virtual functions must be defined as public. However, it may be defined inside the class or outside the class.
- Constructors can't be virtual. This is because constructors are used to create objects which must be done at compile time. However, destructors can be virtual as it deletes the object which always occurs during runtime.

Program 65: Write a program in C++ to show the effect of dynamic binding using virtual function.

Code:

```
#include<iostream.h>
#include<conio.h>
class ONE
{
        int n1;
    public:
        virtual void fun()          //Virtual Function
        {
            n1=5;
            cout<<endl<<"n1="<<n1<<endl;
```

```
        }
};
class TWO:public ONE
{
        int n2;
    public:
        void fun()          //Virtual function is redefined
        {
            n2=10;
            cout<<endl<<"n2="<<n2<<endl;
        }
};
void main()
{
    clrscr();
    ONE *ptr;           //Base class object pointer
    ONE ob1;
    ptr=&ob1;
    ptr->fun();
    TWO ob2;
    ptr=&ob2;
    ptr->fun();
    getch();
}
```

O/P:

Explanation: In the above program, class ONE contains an integer data member n1 and a virtual member function fun(). Now, class TWO is derived from class ONE and it also contains a data member n2 and the base class virtual member function fun() is redefined in derived class. Base class object pointer ptr is used which at one instance refers to base class object ob1 and in another instance it refers to derived class object ob2. Here, during 1st function call system executes the base class member function whereas during 2nd function call system executes derived class member function. This is due to dynamic binding where the decision to call the function is made at runtime time. Due to this system gets some extra time for initialization and then binding takes place which leads to the call of base class member function and derived class member function at different times.

9.3.3 VPTR and VTABLE

To perform late binding, the compiler establishes a virtual table represented by VTABLE. This table contains the addresses of the virtual functions. When objects of base or derived class is created, a void pointer is inserted in the VTABLE known as v VPTR (vpointer). The VPTR points to the VTABLE. When a virtual function is invoked using base class object pointer, the compiler speedily puts the code to obtain the VPTR and searches for the address in VTABLE. In this way appropriate function is invoked and dynamic binding takes place.

9.3.4 Pure virtual function

We know that inheritance generally does not allow use of member function inside base class. Here we have seen that method overriding is not possible if base class does not contain member function. Now, this is a contradictory situation where inheritance and method overriding have a face off. To resolve this situation, C++ provides a mechanism known as **pure virtual function**. Actually, inheritance does not say that we can't write a member function. It says that base class should not perform any operation. So, if base class function contains a member function which does not perform any operation then inheritance condition is satisfied. Since , we write a base class member function, it can be redefined inside the derived class. Hence, method overriding criteria is also satisfied.

Formally, **a pure virtual function is defined as a virtual function without any definition i.e. the function won't perform any operation.** Due to this it is also known as do nothing or dummy function.

A pure virtual function cab be defined as below:

Syntax:

```
virtual        return_type        function_name()=0;
```

Now, a class containing pure virtual function is known as **abstract class**. For abstract class, objects can't be created. If we try to do so, system would produce error message **"cannot create instance of abstract class".**Now, the classes derived from abstract class are known as concrete classes. The pure virtual function must be redefined inside the concrete classes.

9.4 Object Copying

An object is a composite data type in OOPS as it contains data members of different types. The process of copying the attributes of one object to another object of same type is known as object copying.

It can be done in following three ways:

i. Shallow copy: Shallow copy is the process where some or all members of an object can be copied to another object as per the requirement. So, it is fast process as it is not necessary that all members will be used all the time. Consider the diagram shown below:

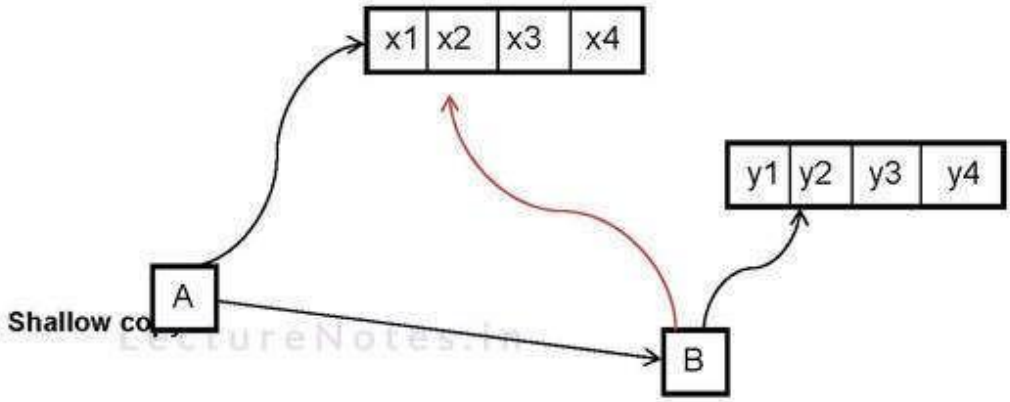


Fig 9.2: Shallow Copy

Here, all or some members of object of object A can be copied to object B as per the requirement.

The **disadvantage** is that changing the value of a member using object B will also change the value of the member for object A i.e. the change is also reflected to object A.

ii. Deep Copy: In deep copy all the members of one object is copied to another object irrespective of the requirement. This avoids overwriting of data members which happened during shallow copy. But, this process is slow as well as costly.

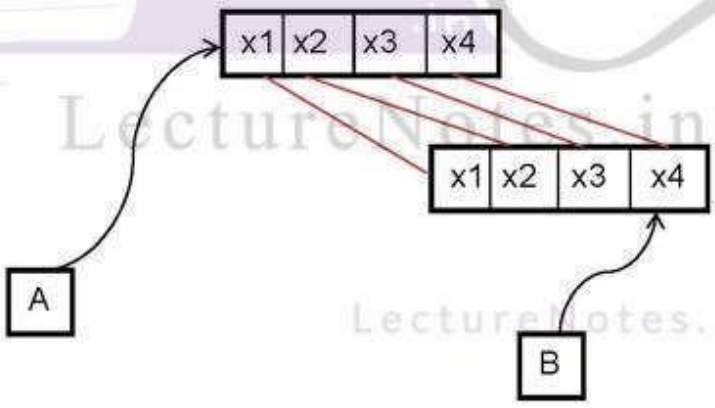


Fig 9.3: Deep copy

Deep copy is slow and costly as all the members are copied in this method.

iii. Lazy copy: Lazy copy is the combination of shallow copy and deep copy. Basically it is deep copy in nature but it can switch to shallow copy as per the requirement.

9.5 Object Slicing

Virtual function permits us to manipulate both base class and derived class objects using same member functions with no modifications. These functions can be invoked using a reference object. If we do so, object slicing takes place.

In **object slicing**, an object of derived class is assigned to a base class object. In this process it copies only the base class members of the object.

Program 66: Write a program in C++ to illustrate object slicing.

Code:

```
#include<iostream.h>
#include<conio.h>
class A
{
        public:
                int a;
                A()
                {
                        a=10;
                }
};
class B:public A
{
        public:
                int b;
                B()
                {
                        a=40;
                        b=50;
                }
```

```
};
void main()
{
        clrscr();
        A x;
        B y;
        cout<<"a="<<x.a<<endl;
        x=y;            //Object slicing
        cout<<"again a="<<x.a<<endl;
        getch();
}
```

O/P: