

Polymorphism

'Poly' means 'many' and 'morph' means 'form' that is Polymorphism means having many forms in one thing. It is the ability of an object to assume or become many different forms of object. In general term polymorphism is occurrence of entity that has single name and many forms which acts differently in different situations. The best real time example is Air Conditioner. It is a single device which performs different actions according to different situations, like in winter it heats the air and in summer it cools the air. One more example is human being. A human being having different behaviors in different situations like a man behaves differently with his father, with his wife he has different behavior, and with his friends he has different behavior and so on.

In inheritance polymorphism is done by method (function) overriding, when base class and derived class have member functions with same declaration but different definition.

Function Overriding

Function overriding is object oriented programming feature that enables a child class to provide different implementation for a method that is already defined in its parent class or one of its parent classes. The overridden method in the child class should have the same name, signature and arguments as the one in its parent class. If the base class and derived class have member functions with same name and arguments and if we create an object of derived class and write code to access that member function then the member function in derived class is only invoked that is the member function of derived class overrides the member function of base class, this mechanism is known as function overriding.

Important points to remember:

- For function overriding Inheritance must be there. Function overriding cannot be done within a class, for this we need a derived class and a base class.
- When a function of base class redefined in the derived class is called overriding.
- Function that is redefined must have exactly the same declaration in base class and derived class with same name, same arguments and same return type.

- Functions should have same data type.
- Overriding is the concept of runtime polymorphism.
- Functions should be defined in public section.

Example for Function overridden:

The following example programs show the function overridden mechanism.

```
class A
{
public:
void print()
{
cout << "Base Class" << endl;
}
};

class B : public A
{
public:
void print()
{
cout << "Derived Class" << endl;
}
};

int main ()
{
B obj;
obj.print();
return 0;
}
```

This function is not invoked due to member function overriding

This functions is invoked instead of function in class A because of member function overriding

Figure1: Function overriding mechanism

Accessing overridden function in base class from derived class using scope resolution operator:

We can access the overridden function in base class from derived class using '::' scope resolution operator.

```
class A
{
    public:
    void print()
    {
        cout << "Base Class" << endl;
    }
};

class B : public A
{
    public:
    void print()
    {
        cout << "Derived Class" << endl;
        A::print(); // it will call print function in base class A
    }
};

int main ()
{
    B obj;
    obj.print();
    return 0;
}
```

A diagram with three arrows. One arrow points from the `obj.print();` line in the `main` function to the `void print()` line in the `class B` definition. A second arrow points from the `A::print();` line in the `class B` definition to the `void print()` line in the `class A` definition. A third arrow points from the `obj.print();` line in the `main` function directly to the `void print()` line in the `class A` definition.

Figure2: Accessing overridden function in base class using scope resolution operator

Let us consider an example which will show the function overridden mechanism and accessing overridden function in base class from derived class using scope resolution operator.

```
#include <iostream>
```

```
#include <string>

using namespace std;

class A
{
    private:
        int roll_no;
        string name;

    public:

        void print()
        {
            cout << "Base Class" << endl;
        }

        void get_info(int x, string y)
        {
            roll_no = x;
            cout << roll_no << endl;
            name = y;
            cout << name << endl;
        }
};

class B : public A
{
    private:
        int roll_no;
        string name;

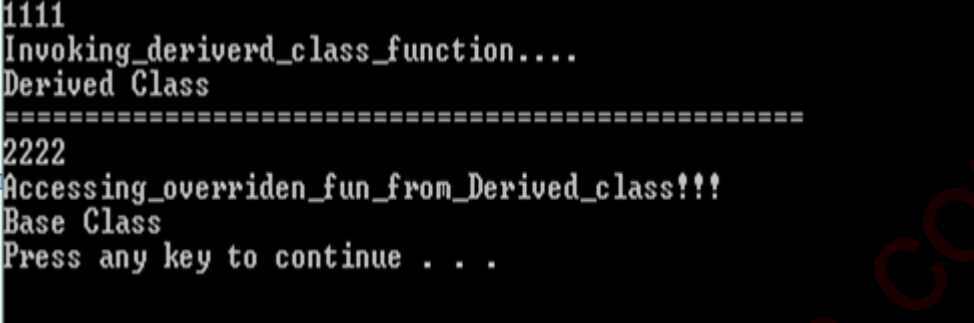
    public:

        void get_info(int x, string y)
        {
            roll_no = x;
            cout << roll_no << endl;
            name = y;
            cout << name << endl;
        }

        void print()
        {
            cout << "Derived Class" << endl;
            cout << "=====" << endl;
            A::get_info(2222, "Accessing_overriden_fun_from_Derived_class!!!");
            A::print();
        }
};

int main()
{
    B obj;
    obj.get_info(1111, "Invoking_derived_class_function....");
}
```

```
obj.print();  
system("pause");  
return 0;  
}
```



```
1111  
Invoking_derived_class_function...  
Derived Class  
=====  
2222  
Accessing_overridden_fun_from_Derived_class!!!  
Base Class  
Press any key to continue . . .
```

Figure3: Output of the program

Virtual Function

A Virtual function is a member function that is declared within the base class and redefined by the derived class. The virtual function is overridden in the derived class, which tells the compiler to perform late binding on this function. 'Virtual' is a keyword is used to declare virtual function in the base class. When a class contains a virtual function which is inherited, then the derived class redefines the virtual function to perform their own actions.

Example program without using virtual keyword:

```
#include <iostream>  
using namespace std;  
  
class Base //Base class  
{  
public:  
void display()  
{  
cout << "Base class" << endl;  
}  
};  
  
class Derived : public Base // Derived class  
{  
public:
```

```
void display()
{
    cout << "Derived class" << endl;
}
};

int main()
{
    Base *ptr;           // Base class pointer
    Derived obj;        // Derived class object
    ptr = &obj;         // assigning derived class object to base class pointer
    ptr ->display();
    system("pause");
    return 0;
}
```

A screenshot of a terminal window with a black background and white text. The text displayed is "Base class" on the first line and "Press any key to continue . . ." on the second line.

Figure4: Output of the program

When Base class pointer points to derived class object and using base class pointer if we call functions which are in both classes, then the base class function is invoked. But if we want call only derived class function using base class pointer, it can be done by using virtual keyword by defining the functions as virtual functions in the base class. In this way the virtual functions support runtime polymorphism.

Example program using virtual keyword:

In the same example we are changing the 'display ()' function in the base class as virtual function by using 'virtual' keyword. Here virtual keyword will lead to late binding of that function as shown in below.

```
Virtual void display ()
{
}
```

```
#include <iostream>

using namespace std;

class Base //Base class
{
public:
    virtual void display()
    {
        cout << "Base class" << endl;
    }
};

class Derived : public Base // Derived class
{
public:
    void display()
    {
        cout << "Derived class" << endl;
    }
};

int main()
{
    Base *ptr; // Base class pointer
    Derived obj; // Derived class object
    ptr = &obj; // assigning derived class object to base class pointer
    ptr ->display();
    system("pause");
    return 0;
}
```



```
Derived class
Press any key to continue . . .
```

Figure5: Output of the program

In the above program using virtual function in the base class, late binding takes place and the derived class function will be called because the base class pointer points to derived class object. So the output here is “Derived class”.

Pure Virtual Function

Pure virtual function is a virtual function which has only declaration part and don't have any definition. It start with 'virtual' keyword and ends with '=0'.

Syntax:

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

Example:

```
class Base
{
    public:
    virtual void display () = 0; // pure virtual function
};
```

The '=0' notation indicates that the virtual function is pure virtual function and it has no definition.

Abstract Class

The Abstract class is a class which contains at least one pure virtual function. These classes are used to provide an interface for its derived classes. Classes inheriting an abstract class must provide definition to the pure virtual function otherwise they will also become abstract classes.

Characteristics of Abstract class:

- Abstract class can have normal variables and functions along with a pure virtual function.
- Abstract class cannot be instantiated, but pointers and references of abstract class can be created.

- Classes inheriting an abstract class must implement all pure virtual functions otherwise they will become abstract classes.
- Abstract classes mainly used for up casting, so that it's derived classes can use its interface.

Example for Abstract class and pure virtual function

```
#include <iostream>

using namespace std;

class Base //Base class
{
public:
    virtual void display() = 0; // pure virtual function
};

class Derived : public Base // Derived class
{
public:
    void display()
    {
        cout << "Function defination in Derived class of pure virtual function" <<
endl;
    }
};

int main()
{
    Base *ptr; // Base class pointer
    //Base obj1; // It will give compile time error -
    // Cannot create object of base class
    Derived obj; // Derived class object
    ptr = &obj; // assigning derived class object to base class pointer
    ptr ->display();
    system("pause");
    return 0;
}
```

```
Function defination in Derived class of pure virtual function
Press any key to continue . . .
```

Figure6: Output of the program

In the above example the base class is abstract class, so we cannot create object of base class.

Virtual Destructors

The destructors in base class can be Virtual but not the constructors. Whenever up casting is done, the destructors of the base class must be made virtual for proper destruction of the object when the program exits.

Let us consider an example with destructor and without destructor and observe the changes.

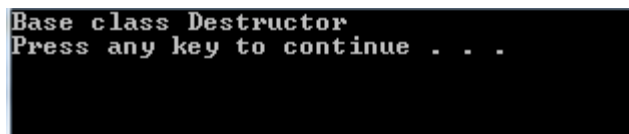
Example without Virtual destructor

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

class Base
{
public:
    ~Base ()
    {
        cout << "Base class Destructor" << endl;
    }
};

class Derived : public Base
{
public:
    ~Derived()
    {
        cout << " Derived Class Destructor" << endl;
    }
};

int main()
{
    Base *ptr;
    ptr = new Derived;
    delete ptr;
    system("pause");
    return 0;
}
```



```
Base class Destructor
Press any key to continue . . .
```

Figure7: Output of the program

In the above example the 'delete b' will only call the base class destructor, which is undesirable because the object of Derived class is un-destroyed, its destructor is never called and which results memory leak.

Example with Virtual destructor

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

class Base
{
public:
    virtual ~Base ()
    {
        cout << "Base class Destructor" << endl;
    }
};

class Derived : public Base
{
public:
    ~Derived()
    {
        cout << "Derived Class Destructor" << endl;
    }
};

int main()
{
    Base *ptr;
    ptr = new Derived;
    delete ptr;
    system("pause");
    return 0;
}
```

A screenshot of a terminal window showing the output of the program. The text displayed is: "Derived Class Destructor", "Base class Destructor", and "Press any key to continue . . ." on three separate lines. The terminal has a black background with white text.

Figure8: Output of the program

In the above example we have used virtual destructor inside the class, so first the derived class destructor is called and then the base class destructor is called as shown in the output figure.

Pure Virtual Destructors

The pure virtual destructor will make its base class abstract, so that we cannot create object of that class. There is no requirement of defining pure virtual destructor in the derived classes. The only difference between virtual destructor and pure virtual destructor is pure virtual destructor make its base class abstract hence you cannot create object of that class.

Example:

```
class Base
{
    public:
    virtual ~Base() = 0; // pure virtual destructor
};
class Derived : public Base
{
    public:
    ~Derived ()
    {
        cout << "Derived Destructor";
    }
};
```