OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 13

Polymorphism in C++

Instructor: Muhammad Abdullah Orakzai

Semester Fall, 2021

Course Code: CL1004

Fast National University of Computer and Emerging Sciences
Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

Polymorphism

Polymorphism is another most important feature of OOP. In polymorphism, the member functions with the same name are defined in each derived class and also in the base class. Polymorphism is used to keep the interface of base class to its derived classes.

Poly means many and ***morphism*** means form. Polymorphism, therefore is the ability for objects of different classes related by inheritance to response differently to the same function call.

Polymorphism is achieved by means of virtual functions. It is rendered possible by fact that one pointer to a base class object may also point to any object of its derived class.

The word **polymorphism** means having many forms. Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Pointer to objects

The member of a class can be accessed through the pointer to the class. The arrow (->) symbol is used to access them. This symbol is also known as member access operator. It is denoted by a hyphen (-) and a greater than sign (>).

The general syntax to access a member of a class through its pointer is:

p-> class member

where

p is the pointer to the object.

-> It is the member access operator. It is used to access the member of the object through the pointer.

class member it is the member of object.

Program 1:

```
#include<iostream>
using namespace std;
class Student
{
    private:
    char name[20];
    int age;
```

```
    public:
    void input()
    {
        cout<<"Enter Name: "<<endl;
        cin>>name;
        cout<<"Enter age: "<<endl;
        cin>>age;
    }
    void show()
    {
        cout<<"Name is: "<<name<<endl;
        cout<<"Age is : "<<age<<endl;
    }
};
int main() {
    Student s, *p;
    p=&s;
    p->input();
    p->show();
    return 0;
}
/*
Output
Enter Name:
Ali
Enter age:
77
Name is: Ali
Age is : 77
*/
```

In the above program, the "**Student**" class is defined and "**P**" is declared as pointer object. The public member functions of the "**Student**" are accessed with the member access operator (->).

If a pointer object is defined to point to a class, then is can also point to any of its subclass. However, even if the pointer points to an object of its derived class, it remains of the base class type.

A program may have a base class and several derived classes. When a pointer is declared to point to the base class, it can also point to the classes derived from the base class.

When a base class pointer is used to point to an object of a derived class, the pointer type remains unchanged, its types remains of the base class. When this pointer is used to access a member of a class with the member access operator (->), the pointer type determines which actual function to be called. It always accesses the member of the class with which types matches. Thus, it will execute the member function of the base class. This is the default method of execution of a member function through pointer.

A program example is given below to illustrate this concept. The program has a base class and two derived classes. The base class and its derived classes have their member functions and the names of their member functions is same.

Program 2

```
#include<iostream>
using namespace std;
class BB
{
    public:

    void ppp()
    {
        cout<<"Hello, it is the base class"<<endl;
    }
};

class d1 : public BB
{
    public:
    void ppp()
    {
        cout<<"It is the first derived class"<<endl;
    }
};
class d2 : public BB
{
    public:
    void ppp()
    {
        cout<<"It is the second derived class"<<endl;
    }
};
int main() {
    BB *p;
    d1 a;
    d2 b;
    p = &a;
    p->ppp();

    p=&b;
    p->ppp();
    return 0;
}
/*
```

```
Output
Hello, it is the base class
Hello, it is the base class
*/
```

In the above program, **BB** is the base class and **d1** & **d2** are the derived classes. Both these classes have been derived from the base class **BB**.

In the main function(), the objects declarations are:

One pointer object "**p**" of the **base class BB** is declared.

One pointer object "**a**" of the **derived class d1** is declared.

One pointer object "**b**" of the **derived class d2** is declared.

The statements are executed as given below:

The statement "**p=&a;"** assigns the address of object a to the pointer object "**p**". when the first statement "**p->ppp();** " is executed, the member function of the base class is executed. Since the type the pointer matches the type member function of the base class, the base class member function is executed.

Similarly, when the second statement "**p->ppp();"** is executed after assigning the address of the object **b** of the derived class **d2** to "**p**", again the member function of the base class is executed.

Program 3

Consider the following example where a base class has been derived by other two classes

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
```

```
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

To solve this, we need **virtual Functions**

Early Binding

The event that take place at the compile time are called early binding. It is also called static binding. It indicates that all information required to call a function is known at the compile time. The compiler decides at compile time what method will respond to the message sent to pointer.

The accessing of member function of a class through the pointer is an example of early binding. In the above program example, the compiler decides at the compile time to always execute the member function of the class whose type matches the types of the pointer irrespective of the contents of the pointer. Thus, whatever the object points to during execution of the program, the pointer always accesses the member of the class whose type matches with its type.

Virtual Functions

A virtual function is a special type of member function. It is defined in the base class and may be redefined in any class derived from this base class. Its name in the base class and the in the derived classes remains the same. Its definition in these classes may be different.

The virtual function in the derived class is executed through the pointer of the public base class.

A virtual function is declared by writing the word **virtual** before function declaration in the base class. The functions with the same name are declared in the derived classes. The use of word **virtual** before function declaration in the derived classes is optional. Once a function is declared virtual in a base class, it becomes **virtual** in all derived classes, even when the keyword virtual is not written in its definition in the derived classes.

The derived classes may have different versions of a virtual function. A virtual function may be redefined in the derived classes. A redefined function is said to ***override*** the base class function.

Now we make a small change in the above program example by declaring the member function “ppp()” as virtual as shown below:

Program 4:

```
#include<iostream>
using namespace std;
class BB
{
    public:

    virtual void ppp()
    {
        cout<<"Hello, it is the base class"<<endl;
    }
};
```

```
class d1 : public BB
{
    public:
    void ppp()
    {
        cout<<"It is the first derived class"<<endl;
    }
};

class d2 : public BB
{
    public:
    void ppp()
    {
        cout<<"It is the second derived class"<<endl;
    }
};
int main() {
    BB *p;
    d1 a;
    d2 b;
    p = &a;
    p->ppp();

    p=&b;
    p->ppp();

    return 0;
}
/*
Output
It is the first derived class
It is the second derived class
*/
```

In the above program the keyword "**virtual**" appears only in the definition of the base class **BB**, it has not been repeated in the derived classes **d1** & **d2** that are derived from the base **BB**. Each derived class **d1** and **d2** and base class **BB** have a member function with the same name i.e. **ppp**.

The statement "**p=&a;"** assigns the address of the object a to the pointer object "**p**". When the statement "**p->ppp();"** is executed for the first time, the pointer object of the base class contains the address of the object of the derived class **d1** and the virtual member function of the derived class **d1** is executed.

Similarly, when the "**p->ppp();**" statement is executed for the second time, the pointer object "**p**" of the base class contains the address of the derived class **d2** and thus the virtual function of the derived d2 is executed.

In executing virtual function, the computer selects the function to be called based on the contents of the pointer and not on the bases of the type of the pointer.

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Now coming back to above example, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this –

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result –

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Late Binding

The events that take place at the time of execution of the program are called late binding. It is also called dynamic binding. With the late binding, the compiler does not know which method will actually respond to the message because the type of the pointer is not known at the compile time.

If the keyword virtual is used, the system used the late binding.

The execution of virtual functions through pointers is an example of late binding. While executing a virtual function variable, the computer decides at the time of execution of the program. It executes the function depending upon the value in the pointer during execution of the program.

Pure Virtual Function

The virtual function that is only declared but not defined in the base class is called the pure virtual functions.

A function is made pure virtual by preceding its declaration with the keyword **virtual** and made by post fixing it with =**0**.

The pure virtual function simply tells the compiler that the function is pure. The class that contains the pure virtual function exists only to act as a parent or base of the derived classes. The function is implemented in the derived classes.

For example,

```
#include<iostream>
using namespace std;
class BB
{
    public:

    virtual void ppp()=0;
};
class d1 : public BB
{
    public:
    void ppp()
    {
        cout<<"It is the first derived class"<<endl;
    }
};
class d2 : public BB
{
    public:
    void ppp()
    {
```

```
        cout<<"It is the second derived class"<<endl;
    }
};
int main() {
    BB *p;
    d1 a;
    d2 b;
    p = &a;
    p->ppp();
////////////////////////////////////////////////////
    p=&b;
    p->ppp();
    return 0;
}
/*
Output
It is the first derived class
It is the second derived class
*/
```

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following –

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Abstract Base Class and Concrete Derived Class

The base class that has one or more pure virtual functions is called the Abstract Base Class. These classes are designed as a framework or layout for derived classes. An abstract base class cannot be instantiated i.e. an object of its type cannot be defined. But, a pointer to an abstract base class can be defined.

The derived class that does not have any pure virtual function is called Concrete derived class. The pure virtual function of the Abstract Base Class is implemented in the Concrete Derived Class. The derived classes usually have their own versions of the virtual functions.

In Object Oriented Programming a hierarchy of classes are defined. The Abstract Base Class is defined at the top of this hierarchy. The Concrete Derived Classes are derived from the Abstract Base Class. The Pure virtual functions are declared in the base class. Each derived class contains its own versions of these pure virtual functions. In this way a uniformity is achieved within the hierarchy of the class.

A pure virtual function is sometimes called an abstract function, and a class with at least one pure virtual function is called an abstract class. The C++ compiler will not allow you to instantiate an abstract class. Abstract classes can only be subclassed: that is, you can only use them as base classes from which to derive other classes.

A class derived from an abstract class inherits all functions in the base class, and will itself be an abstract class unless it overrides all the abstract functions it inherits. The usefulness of abstract classes lies in the fact that they define an interface that will then have to be supported by objects of all classes derived from it.

```
#include <iostream>
using namespace std;
class Shape
{
  protected:
    double width, height;
  public:
    void set_data (double a, double b)
    {
        width = a;
        height = b;
    }
    virtual double area() = 0;
};
class Rectangle: public Shape
{
public:
    double area ()
    {
        return (width * height);
    }
```

```
};
class Triangle: public Shape
{
public:
    double area ()
    {
        return (width * height)/2;
    }
};

int main ()
{
    Shape *sPtr;

    Rectangle Rect;
    sPtr = &Rect;

    sPtr -> set_data (5,3);
    cout << "Area of Rectangle is " << sPtr -> area() << endl;

    Triangle Tri;
    sPtr = &Tri;

    sPtr -> set_data (4,6);
    cout << "Area of Triangle is " << sPtr -> area() << endl;
    return 0;
}
/*
Output
Area of Rectangle is 15
Area of Triangle is 12
*/
```

References

https://beginnersbook.com/2017/08/cpp-data-types/

http://www.cplusplus.com/doc/tutorial/basic_io/

https://www.w3schools.com/cpp/default.asp

https://www.javatpoint.com/cpp-tutorial

https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/?ref=lbp

https://www.programiz.com/

https://ecomputernotes.com/cpp/