OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 14

Polymorphism in C++

Instructor: Fariba Laiq

Semester Spring, 2023

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.

Consider the following code:

```
#include <iostream>
#include <string>
using namespace std;
class Shape
{
	public:
	void draw()
	{
		cout<<"Drawing shape"<<endl;
	}
};
class Rectangle
{
	int length, width;
	public:
		void draw()
		{
			cout<<"Drawing rectangle"<<endl;
		}
};

class Circle: public Shape
{
	int radius;
	public:
		void draw()
		{
			cout<<"Drawing Circle"<<endl;
		}
};
int main()
{
	Shape *s=new Circle();
	s->draw();
}
```

Output:

Drawing shape

In the above code, we have override the draw() method in the chid's class. In main we have declared a pointer of a base class (Shape *s) and initialized it with the object of the child's class (Circle). But when we call the draw() method, the draw() method of the parent is executed.

Now to achieve the polymorphic behaviour by calling the draw() method with the pointer of the based class that actually points to the object of the child's class. We need to make the draw() method virtual, for late binding. This makes the draw() method to be called upon the actual object, not on the pointer 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:

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**.

```
class Shape
{
        public:
        virtual void draw() // virtual function
        {

        }
};
```

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 exists only to act as a parent or base of the derived classes. The function is implemented in the derived classes. A class containing a pure virtual function becomes an abstract class. An abstract class cannot be instantiated means that the object for that class cannot be created. However a pointer of that type can be created.

Example Code:

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

Here's the updated version of the previous code.

```
#include <iostream>
#include <string>
using namespace std;
class Shape
{
	public:
	virtual void draw() = 0; // pure virtual function
};
class Rectangle: public Shape
{
	int length, width;
	public:
		void draw()
		{
			cout<<"Drawing rectangle"<<endl;
		}
};

class Circle: public Shape
{
	int radius;
	public:
		void draw()
		{
			cout<<"Drawing Circle"<<endl;
		}
};
int main()
{
	Shape *s=new Circle();
	s->draw();
	s=new Rectangle();
	s->draw();
}
```

This code demonstrates the use of polymorphism in C++ with the help of abstract classes and virtual functions.

The abstract class Shape has a pure virtual function draw(), which means it has no implementation and must be overridden by derived classes. This allows us to create an object of the Shape class and call the draw() function on it, without knowing exactly which derived class it belongs to.

The derived classes Rectangle and Circle inherit from the Shape class and implement their own version of the draw() function. This is where the polymorphism comes in - when we create a pointer to a Shape object and assign it to a derived class object (Circle or Rectangle), we can call the draw() function on that pointer and it will call the appropriate derived class's implementation of the function.

This demonstrates the power of polymorphism - we can create a single pointer to a Shape object and assign it to any derived class object, and call the same draw() function on that pointer, and it will call the appropriate implementation of the function based on the actual object it is pointing to. This allows for more flexible and extensible code, as new derived classes can be added without changing the existing code that uses the Shape class.

Achieving polymorphism through array of Base class Pointers

```
#include <iostream>
#include <string>
using namespace std;
class Shape
{
	public:
	virtual void draw() = 0;
};
class Rectangle: public Shape
{
	int length, width;
	public:
		void draw()
		{
			cout<<"Drawing rectangle"<<endl;
		}
};
class Circle: public Shape
{
	int radius;
	public:
		void draw()
		{
			cout<<"Drawing Circle"<<endl;
		}
};
void drawAll(Shape *s[])
{
	cout<<"\nDrawing shapes in drawAll() method"<<endl;
	for(int i=0; i<2; i++)
	{
		s[i]->draw();
	}
}
int main()
{
	Shape *s[2];
	s[0]=new Circle();
	s[1]=new Rectangle();
	cout<<"Drawing shapes in main"<<endl;
	for(int i=0; i<2; i++)
	{
		s[i]->draw();
	}
	drawAll(s);
}
```

Output:

Drawing shapes in main

Drawing Circle

Drawing rectangle

Drawing shapes in drawAll() method

Drawing Circle

Drawing rectangle

This program that demonstrates the use of polymorphism with an array of pointers to objects of different derived classes. The program creates an array of pointers to Shape objects, which includes a Circle object and a Rectangle object.

The program then calls the draw() method on each object in the array using a for loop in the main() function. This demonstrates how a single pointer to a Shape object can be used to call different implementations of the draw() method depending on the actual object it is pointing to.

The program also includes a drawAll() function that takes an array of Shape pointers as a parameter and calls the draw() method on each object in the array. This is another example of polymorphism, as the function can be used with any array of Shape objects, regardless of their actual type.

Virtual Destructor in C++

Due to early binding, when the object pointer of the parent class is deleted, which was pointing to the object of the derived class then, only the destructor of the parent class is invoked; it does not invoke the destructor of the child class, which leads to the problem of memory leak in our program.

So, When we use a Virtual destructor, i.e., a virtual keyword preceded by a tilde(~) sign and destructor name, inside the parent class, it makes sure that first the destructor of the child class should be invoked. And then, the destructor of the parent class is called so that it releases the memory occupied by both destructors.

The following code will only call the parent's destructor not the child destructor.

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

class Parent
{
	public:
	~Parent()
	{
		cout<<"Parent destructor called"<<endl;
	}
};
class Child: public Parent
{
	public:
	~Child()
	{
		cout<<"Child destructor called"<<endl;
	}
};
int main()
{
	Parent *p=new Child();
	delete p;
}
```

Output:

Parent destructor called

In order to call the destructor of the child class along with the destructor of the parent's class, the parent's class destructor must be declared as virtual.

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

class Parent
{
	public:
	virtual ~Parent()
	{
		cout<<"Parent destructor called"<<endl;
	}
};
class Child: public Parent
{
	public:
	~Child()
	{
		cout<<"Child destructor called"<<endl;
	}
};
int main()
{
	Parent *p=new Child();
	delete p;
}
```

Output:

Child destructor called

Parent destructor called

Now it will make sure that first the destructor of child's class is invoked then the destructor of the parent's class is invoked. That's why destructor of the parent's class must always be made virtual.

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/