Lesson 14: Polymorphism

14.1. Introduction

In our previous lesson, we learnt about one of the features of OOP called inheritance- definition of inheritance, base class and derived class, data members and member functions, classification of member functions, types of inheritance and finally advantages and disadvantages of inheritance. Today, we will learn about another feature of OOP called polymorphism.

14.2. Lesson objectives

By the end of this lesson you will be able to:

- Define polymorphism
- Describe virtual functions
- Describe different types of polymorphism
- Write simple program to demonstrate polymorphism.

14.3. Lesson outline

This lesson is organized as follows:

14.4. Definition of polymorphism

Polymorphism is a feature of OOP that allows programmers to create two functions with identical names. In essence, polymorphism means that some code or operation or object behaves differently in different contexts. In c++ polymorphism refers to using virtual functions. There are two types of polymorphism- static polymorphism and dynamic polymorphism.

a) **Subtype polymorphism :** It's the ability to use derived classes through **base class pointers and references** .It is also known as **runtime polymorphism**. The resolution of polymorphic function calls happens at runtime through an **indirection**. Also called inclusion polymorphism.
b) **Parametric polymorphism :** This is also known as **compile-time polymorphism**. This Polymorphism provides a means for executing the same code for any type. Can be implemented using templates or **overriding** or **late binding**.
c) **Ad-hoc polymorphism :** This allows functions with the same name act differently for each type. It is also known as **overloading** or **early binding.**

d) **Coercion polymorphism:** This is also known as (implicit or explicit) casting. Coercion happens when an object or a primitive is cast into another object type or primitive type. **Converting one type of data to another type**

14.5. Static polymorphism.

Static polymorphism is also referred to as early binding (compile time polymorphism or ad-hoc polymorphism). This uses function overloading. However, the decision of which function will be called at run time is made at compile time. In the following example, the functions show () is overloaded. The decision on which show() will be called, is made during compilation.

Example 1: A program to demonstrate the use of function overloading (early binding).

```
#include <iostream>  using namespace std;
class A
{ public:
        void show();
};
void A::show()//definition of base function
{
        cout << "Good morning" << endl;
}
class B :public A
{ public:
        void show();
};
void B::show()
{
        cout << "Good afternoon" << endl;
};
class C :public A
{ public:
        void show();
};
void C::show()
{
        cout << "Good evening" << endl;
}
void main()
{
A       a;
B       b;
C       c;
            a.show();
            b.show();
            c.show();
```

}

14.6. Dynamic polymorphism.

Dynamic polymorphism is also called dynamic binding (Subtype polymorphism). If an overridden function of base class is called with respect to a pointer or a reference of the base class type, then which of the functions (the base class function or one of the derived class versions) will virtually be called, is decided based on the type of the object pointed at or referred to at run time. The same function has more than one form (in the base class and derived classes). Its call can lead to execution of a particular version depending upon circumstances arising during run time. It is implemented using virtual functions. **Example 2**: A program to demonstrate the use of virtual function (late binding).

```
// Virtual functions and polymorphism.
#include <iostream> using namespace std;
class A //Base class
{ public:
        int i;
        A(int x)
        {
                i = x;
        }
        virtual void func()
        {
                cout << "Using base version of func(): ";
                cout << i << "\n";
        }
};
class B : public A //Derived class B
{
public:
        B(int x) : A(x) { }
        void func()
        {
             cout << "Using derived1's version of func(): ";          cout << i*i << "\n";
}
};
class C : public A  //Derived class C
{
public:
        C(int x) : A(x) { }
        // C does not override func( )
};
void main()
{
```

```
        A *p;
A       a(5);
B       b(5);   C c(5);
        p = &a;
        p->func(); // use A's func( )
        p = &b;
        p->func(); // use B's func( )
        p = &c;
        p->func(); // use A's func( )
}
```

Example 3: A program to demonstrate the use of virtual function (late binding). When the code below is compiled it produces following result:
Rectangle class area
Triangle class area
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.

```
#include <iostream>  using namespace std; 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;
        }
};
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 << "Rectangle class area :" << endl;
                return (width * height / 2);
        }
};
void main() // Main function for the program
{
        Shape *shape;
        Rectangle rec(10, 7);
        Triangle  tri(10, 5);
shape = &rec;   // store the address of Rectangle
        shape->area();// call rectangle area.
    shape = &tri; // store the address of Triangle       shape->area();// call triangle area.
}
```

14.7. Virtual function

This is a member function declared with the keyword virtual. If a virtual function is overridden in a derived class and called with respect to a pointer of base class type that points at an object of the same derived class then the function called would be of the derived class and not of base class. 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 polymorphism (early binding) 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 polymorphism** or **late binding**.

Pure Virtual Functions:

It's possible that you'd 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**.

14.9. Revision questions

[a]. Explain how the compiler resolves a call to virtual function.
[b]. Explain the difference between early and late binding.
[c]. Explain what is meant by pure virtual function.
[d]. With example explain how virtual function achieves run time polymorphism. Write a program to demonstrate this. The program should have the following.

i. Base class with a virtual function
ii. Derived class with similar function as virtual in base class
iii. Declaration of base and derived object and pointer showing access to the functions.

14.10. Summary

In this lesson we have learnt about polymorphism as one of the principles of OOP. Polymorphism was noted as the ability of an object to respond differently to the same message (function call). We learnt about static and dynamic polymorphism. We observed that the code contained in the functions of the base class remains inextensible without the use of virtual functions. The virtual functions make expected overrides in the derived class effective. Marking a base class function as pure virtual function forces its override in the derived class.

14.11. Suggested reading.

[1]. Object-oriented programming with c++ by Sourav Sahay.
[2]. Sams Teach Yourself C++ in 24 hours by Jesse Liberty and Rogers Cadenhead.
[3]. Object oriented programming using c++ by Joyce Farrell 4th.Edition