Lab 09:

Introduction to Polymorphism and Abstract Base Classes

Polymorphism

Polymorphism is a mechanism that allows you to implement a function in different ways. Polymorphism is by far the most important and widely used concept in object oriented programming.

Pointers to base class

We have seen that it is possible to derive a class from a base class and that we can add functionality to member functions.

One of the features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism takes advantage of this feature.

Let's take a look at an example of a base class and derived classes:

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

class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
};

class CRectangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height);
                }
};

class CTriangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height / 2);
                }
};

int main ()
{
        CRectangle rectangle;
        CTriangle triangle;

        CPolygon * ptr_polygon1 = &rectangle;
        CPolygon * ptr_polygon2 = &triangle;

        ptr_polygon1->setup(2,2);
        ptr_polygon2->setup(2,2);

        cout << rectangle.area () << endl;
        cout << triangle.area () << endl;

        return 0;
}
```

As you can see, we create two pointers (ptr_polygon1 and ptr_polygon2) that point to the objects of class CPolygon. Then we assign to these pointers the address of (using the reference ampersand sign) the objects rectangle and triangle. Both rectangle and triangle are objects of classes derived from CPolygon.

In the cout statement we use the objects rectangle and triangle instead of the pointers ptr_polygon1 and ptr_polygon2. We do this because ptr_polygon1 and ptr_polygon2 are of the type CPolygon. This means we can only use the pointers to refer to members that CRectangle and CTriangle inherit from Cpolygon.

If we want to use the pointers to class CPolygon then area() should be declared in the class CPolygon and not only in the derived classes CRectangle and Ctriangle.

The problem is that we use different versions of area() in the derived classes – CRectangle and Ctriangle – so we can't implement one version of area() in the base class CPolygon. (If they were the same we had no problem.)

We can fix this by using **virtual members.**

Virtual Members

A virtual member is a member of a base class that we can redefine in its derived classes. To declare a member as virtual we must use the keyword virtual.

Let's change our previous example:

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

class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area()
                {
                        return (0);
                }
};

class CRectangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height);
                }
};

class CTriangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height / 2);
                }
};

int main ()
{
        CRectangle rectangle;
        CTriangle triangle;
        CPolygon polygon;

        CPolygon * ptr_polygon1 = &rectangle;
        CPolygon * ptr_polygon2 = &triangle;
        CPolygon * ptr_polygon3 = &polygon;

        ptr_polygon1->setup(2,2);
        ptr_polygon2->setup(2,2);
        ptr_polygon3->setup(2,2);

        cout << ptr_polygon1->area () << endl;
        cout << ptr_polygon2->area () << endl;
        cout << ptr_polygon3->area () << endl;

        return 0;
}
```

Because of the change – adding area() as a virtual member of CPolygon – now all the three classes have all the same members (width, height, setup() and area().)

A class that declares or inherits a virtual function is called a **polymorphic class.**

Abstract Base Classes:

At the design level, an abstract base class (ABC) corresponds to an abstract concept. For instance: if you ask to draw a shape, I will probably ask what kind of shape. The term "shape" is an abstract concept, it could be a circle, a square, etc, etc. You could say in C++ that class CShape is an abstract base class (ABC) and class circle (etc) could be a derived class.

As we look at the C++ language we could say that an abstract base class has one or more **pure virtual** member functions.

In the example above we put an implementation (return (0);) in the virtual member function area(). If we want to change it into a pure virtual member function we use =0; instead of the return (0). So the class will look like this:

```
class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area() = 0;
};
```

This pure virtual function area() makes CPolygon an abstract base class. But you have to remember the following: by adding a pure virtual member to the base class, you are forced to also add the member to any derived class.

So our example should now look like this:

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

class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area() = 0;
};

class CRectangle: public CPolygon
{
        public:
                int area(void)
                {
                        return (width * height);
                }
};
```

```
class CTriangle: public CPolygon
{
        public:
                int area(void)
                {
                        return (width * height / 2);
                }
};

int main ()
{
        CRectangle rectangle;
        CTriangle triangle;

        CPolygon * ptr_polygon1 = &rectangle;
        CPolygon * ptr_polygon2 = &triangle;

        ptr_polygon1->setup(2,2);
        ptr_polygon2->setup(2,2);

        cout << ptr_polygon1->area () << endl;
        cout << ptr_polygon2->area () << endl;

        return 0;
}
```

Note: there is also an extra void in the derived classes CRectangle and CTriangle.

Using a unique type of pointer (CPolygon*) we can point to objects of different but related classes. We can make use of that. For instance: we could implement an extra function member in the abstract base class CPolygon that can print the result of the area() function. (Remember that CPolygon itself has no implementation for the function area() and still we can use it, isn't it cool.) After implementation of such a function the example will look like this:

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

class CPolygon
{
        protected:
                int width, height;
        public:
                void setup (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area(void) = 0;
                void onscreen(void)
                {
                        cout << this->area() << endl;
                }
};
```

```
class CRectangle: public CPolygon
{
        public:
                int area(void)
                {
                        return (width * height);
                }
};

class CTriangle: public CPolygon
{
        public:
                int area(void)
                {
                        return (width * height / 2);
                }
};

int main ()
{
        CRectangle rectangle;
        CTriangle triangle;

        CPolygon * ptr_polygon1 = &rectangle;
        CPolygon * ptr_polygon2 = &triangle;

        ptr_polygon1->setup(2,2);
        ptr_polygon2->setup(2,2);

        ptr_polygon1->onscreen();
        ptr_polygon2->onscreen();

        return 0;
}
```

Note: Before doing your lab exercises run the examples.

Exercises will be provided by the instructors in the lab.