



CL-217 OBJECT ORIENTED PROGRAMMING LAB

LAB MANUAL 11

INSTRUCTOR: MUHAMMAD HAMZA

SEMESTER SPRING 2020

Virtual Functions and Abstract Classes

Before going through the content of the manual, Watch the links below.

1. <https://www.youtube.com/watch?v=JU8DbwBvOWE>
2. <https://www.youtube.com/watch?v=wWY7I5S7A7w>
3. <https://www.youtube.com/watch?v=SvesRBYu65k>

Virtual Functions and Polymorphism:

Virtual means *existing in appearance but not in reality*. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call.

Program # 01

```
#include<iostream>

using namespace std;

class shape{
    int size;
    public:

    void draw()
    {
        cout<<"Shape  Class"<<endl;
    }

};

class circle: public shape{

    int raduis;
    public:
```

```
void draw ()
{
    cout<<"Circle"<<endl;
}

};

class rectangle: public shape{

    int radius;
public:

    void draw ()
    {
        cout<<"Rectangle"<<endl;
    }
};

void main()
{

    shape* ptrarr[2];
    ptrarr[0]=new circle;
    ptrarr[1]=new rectangle;

    for(int i=0;i<3;i++)
        ptrarr[i]->draw();

    system("pause");
}
```

Program#02

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

class shape{

    int size;
    public:

    virtual void draw()
    {
        cout<<"Shape  Class"<<endl;
    }
};

class circle: public shape{

    int raduis;
    public:
    void draw()
    {
        cout<<"Circle"<<endl;
    }
};

class rectangle: public shape{
    int raduis;
    public:

    void draw()
    {
        cout<<"Rectangle"<<endl;
    }
};
```

```
void main()
{
    shape* ptrarr[2];
    ptrarr[0]=new circle;
    ptrarr[1]=new rectangle;

    for(int i=0;i<3;i++)
        ptrarr[i]->draw();

    system("pause");
}
```

Program # 03

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

class shape
{
    int size;
    public:

    virtual void draw()
    {
        cout<<"Shape  Class"<<endl;
    }
};

class circle: public shape{
    int radius;
    public:

    void draw()
    {
        cout<<"Circle"<<endl;
    }
};

class rectangle: public circle{

    int raduis;

    public:
    void draw()
    {
        cout<<"Rectangle"<<endl;
    }
};
```

```
void main()
{

    shape* ptrarr[2];
    ptrarr[0]=new circle;
    ptrarr[1]=new rectangle;

    for (int i=0; i<2; i++)
    {

        ptrarr[i]->draw();

    }
    system("pause");
}
```

This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in ptrarr points to a rectangle, the function of the rectangle is called; if it points to a circle, the circle-drawing function is called. This is called *polymorphism*, which means *different forms*.

Abstract Classes and Pure Virtual Functions:

Think of the shape class in the above programs. We'll never make an object of the shape class; we'll only make specific shapes such as circles and rectangles. When we will never want to instantiate objects of a base class, we call it an *abstract class*. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.

How can we make it clear to someone using our family of classes that we don't want anyone to instantiate objects of the base class? How can we do that? By placing at least one *pure virtual function* in the base class. A pure virtual function is one with the expression `=0` added to the declaration.

Here the virtual function `draw ()` is declared as

```
virtual void draw() = 0; // pure virtual function
```

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The `=0` syntax is simply how we tell the compiler that a virtual function will be pure. Now if in `main ()` you attempt to create objects of class `Base`, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class.

Once you've placed a pure virtual function in the base class, you must override it in all the derived classes from which you want to instantiate objects. If a class doesn't override the pure virtual function, it becomes an abstract class itself, and you can't instantiate objects from it (although you might from classes derived from it).

Program # 04

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

class shape{

    int size;
    public:

    virtual void draw()=0;
};
```



```
class circle: public shape{

    int radius;
    public:

    void draw()
    {
        cout<<"Circle"<<endl;

    }
};

class rectangle: public shape{
    int radius;

    public:
    void draw()
    {
        cout<<"Rectangle"<<endl;

    }
};

void main()
{
    shape* ptrarr[2];
    ptrarr[0]=new circle;
    ptrarr[1]=new rectangle;

    for(int i=0;i<2;i++)
    {
        ptrarr[i]->draw();
    }

    system("pause");
}
```