Pure Virtual Member Functions

In this lesson, we'll be learning about a very important concept of polymorphism, i.e., Pure Virtual Member Functions.

WE'LL COVER THE FOLLOWING

Abstract Class

As we've seen in the Inheritance chapter, we can only make derived class's objects to access their functions, and 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 to Write a Pure Virtual Function?

It may also provide an interface for the class hierarchy 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.

`=0` Sign

The equal sign `=` here has nothing to do with the 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.

Overriding Virtual Function

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). For consistency, you may want to make all the virtual functions in the base class pure.

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

// A simple Shape interface which provides a method to get the Shape's area
class Shape {
  public:
  virtual float getArea() = 0;
};
// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
  }
  float getArea(){
    return width * height;
  }
};

// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};

// A Square is a Shape with a specific length
class Square : public Shape {
  private:
  float length;

  public:
  Square(float len){
    length = len;
  }
  float getArea(){
    return length * length;
  }
};
```

```
int main() {
  Shape * shape[3];   // Referencing Shape class to Rectangle object

  Rectangle r(2, 6);    // Creating Rectangle object
  shape[0] = &r;   // Referencing Shape class to Rectangle object

  Circle c(5);    // Creating Circle object
  shape[1] = &c;   // Referencing Shape class to Circle object

  Square s(10);   // Creating Square object
  shape[2] = &s;     // Referencing Shape class to Circle object

  for(int i=0; i<3; i++)
    cout << shape[i]->getArea() << endl;
}
```

Explanation

Now in `main()` you attempt to create objects of a `Shape` class, 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. Notice that, although this is only a declaration, you never need to write a definition of the `Shape` class **getArea()**. Initialize the `Shape` class pointer and point it to objects of derived classes to access the **getArea()** function of respective classes.

Let's move on to a quick quiz to test your understanding of polymorphism.