

Lab 6

Due No Due Date **Points** 15 **Submitting** a file upload

Available after Apr 25, 2021 at 7pm

Implementing Inheritance and Operator Overloading

For this lab we will be working with shapes. We'll create several classes to represent different shapes using inheritance, and then overload some operators for them.

(2 pt) Implementing the Shape class

The first class we'll write is one to represent a generic shape with a name and a color.

Create two new files, shape.h and shape.cpp, and in them define a Shape class. Here's the start of the class definition you should use:

```
class Shape {  
    private:  
        string name;  
        string color;  
    public:  
        float area();  
        ...  
};
```

Your class should also have constructors, accessors, and mutators as appropriate. In addition, your class should have an area() member function for computing the shape's area. For this generic Shape class, the area() member function can simply return 0, since we aren't actually defining the shape itself.

(1 pt) Testing the Shape class

In addition to your files shape.h and shape.cpp, create a new file called application.cpp. In this file, write a simple main() function that instantiates some Shape objects and prints out their information. Write a Makefile to specify compilation of your program.

(4 pts) Implementing the Rectangle and Circle classes

Create new files rectangle.h, rectangle.cpp, circle.h, and circle.cpp. Implement a Rectangle class and a Circle class in them. Both of these classes should be derived from your Shape class. The Rectangle class should have a width and a height, and the Circle class should have a radius. Here's the start of the class definitions you should use:

```
class Rectangle : public Shape {  
    private:
```

```

    float width;
    float height;
public:
    float area();
    ...
};

class Circle : public Shape {
private:
    float radius;
public:
    float area();
    ...
};


```

Both of these class should have constructors, accessors, and mutators as needed, and each one should override the Shape class's area() member function to compute areas appropriate for rectangles and circles.

(2 pt) Testing the Rectangle and Circle classes

Add some code to your application.cpp file to instantiate and print out some Rectangle and Circle objects, and update your Makefile to include the new source files appropriately.

(2 pt) Implementing the Square class

 create new files square.h and square.cpp and implement a Square class that derives from your Rectangle class within them. Your Square class should not contain any new data members, and you cannot change any members of the Rectangle class to protected or public access. Instead, figure out how to implement a public interface for your Square class by appropriately using the width and height of your Rectangle class via its public interface (i.e. via the Rectangle class' constructors, accessors, and mutators). Specifically, the public interface to your Square class should use the public interface of your Rectangle class while enforcing the constraint that a square's width and height are equal. Here's the start of a class definition you could use:

```

class Square : public Rectangle {
public:
    ...
};

```

(1 pt) Testing the Square class

Once your Square class is written, add some lines to your application.cpp to instantiate and print out some Square objects, and update your Makefile to include the new source file appropriately.

(3 pts) Operator Overloading

Make an operator overload member function for comparing rectangles based on their area. For example, you want to have the ability to compare two rectangles using the > and < operators, such as

```
if (rect1 > rect2) {  
    //do something  
}
```

In order to do this, you need to make two operator overloads (one for $>$ and one for $<$) that support the Rectangle Class.

```
bool operator> (const Rectangle &, const Rectangle &);  
bool operator< (const Rectangle &, const Rectangle &);
```

Next week we will learn about how to make this work for **any** two shapes—not just rectangles.

Convince your TA that you can create different types of shapes and the inheritance and operator overloading is working for comparing rectangles.

