EECS 211 Lab 8

Inheritance

Winter 2018

In this week's lab, we will be going over inheritance, and doing some more practice with classes.

If you have any lingering questions during the lab, don't hesitate to ask your peer mentor!

Getting the code

Download the zip file from the course site:

http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab08.zip

After you have downloaded the zip file onto your laptop, extract the zip file into its own folder. Make sure you keep track of which folder it's in! Next, open up CLion and Click on File –> Open Project, and click on the Lab 8 project that you just unzipped.

Once you open the project, try building the lab and then running the lab8 executable. You should see some output printed in your output subwindow. If you need a reminder on how to build and run code in CLion, consult lab 3/4 or ask your TA. Once this works, you're ready to start the lab!

Inheritance

General Idea

Inheritance is an incredibly important idea, central to object oriented design. The main concept to understand for inheritance is the parent-child relationship. This is represented in C++ with a *base* class which contains the general information that a set of child classes, called *derived* classes, inherit from. In situations where you have different variants of something which share several common required functions and data members, inheriting from one base class can not only help you abstract your code, but also help you have better code organization.

An Example

Let's say we are creating a geometry program which has a bunch of different shapes we want to use. We could create a shape base class that has circles, rectangles, and triangles all inheriting from our

shape parent, or *base* class. As you can imagine, some common fields we may want would be things like the size of the perimeter of the shape. However, for the circles we may also want to add a radius field, and for the rectangles and triangles we may want to add length and width fields.

The Protected Access Modifier

So far you had only seen public and private access modifiers. With public data members and functions, both you and any other class can access those fields. With private data members and functions, only other member functions of your class can access those private data members and functions. However, there is a third access modifier called *protected*. With protected access modifiers, like private you and your member functions are able to access the data member or function. However, your derived classes now also are able to access that same field! This basically becomes a way to make things private to only you and your derived classes, and is extremely common and important to object oriented design.

Representing This All in C++

In order to represent this in C++, we would first need to make the Shape base class, then create the derived classes from the Shape base.

This looks something like this:

```
class Shape{
    protected:
        double perimeter_;
    public:
        void setPerimeter(double perimeter){
            perimeter_ = perimeter;
        }
        // Virtual function:
        virtual void print(){
            cout<<"I am a Shape! \n";
        }
};

class Rectangle: public Shape {
    private:
        double length_;
        double width_;
    public:
        // Basic setters and getters:
```

```
        void setLength(double length){
            length_ = length;
        }
        void setWidth(double width){
            width_ = width;
        }
        double length(){
            return length_;
        }
        double width(){
            return width_;
        }

        // Area function:
        double area(){
            return width_ * length_;
        }
        void print(){
            cout<<"I am a Rectangle! \n";
        }

};

class Circle: public Shape {
    private:
        double radius_;
    public:
        void setRadius(double radius){
            radius_ = radius;
        }
        double getRadius(){
            return radius_;
        }
        // Area function:
        // Notice how we are able
        // to access the perimeter data member

        double area(){
            // area for circle =
            // .5 * perimeter * radius
            return .5 * perimeter_ * radius_;
        }
        void print(){
            cout<<"I am a Circle! \n";
```

```
        }
};
```

Notice how for the area function of the Circle derived class, we are able to access the perimeter_ data member from the parent shape. If the perimeter_ data member of Shape had a private access modifier as opposed to a protected access modifier, we wouldn't be able to access it.

Virtual Functions

Virtual functions are functions defined in a base class that will give the derived class's function with the same name, return types, and arguments precedent over the virtual function. Let's say we have our *print* function for our Shape base class as defined above, with one difference - let's suppose that the *print* isn't virtual. If that function was not virtual, and you were to write the following code (assuming we have the shapes from above, where both the Shape and each of the derived classes have a *print* function defined):

```
Circle circ;
Shape* shp = &circ;
shp->print();
// --> Outputs:
// "I am a Shape!"
```

Since shp is using its *print* function, ***which is not a virtual function***, even though shp points to a Circle, we are still using the base, Shape class's *print* function.

Now, if our Shape defined its *print* function as a virtual function as we do actually have above, we would instead be using the derived, Circle print function. This is illustrated below:

```
Circle circ;
Shape* shp = &circ;
shp->print();
// Outputs:
// "I am a Circle!"
```

Notice how with the exact same code besides changing the print function to virtual, we are now able to have our derived class's function take priority! As you get more practice with inheritance, this will become increasingly useful!

Pure Virtual Functions

A pure virtual function is very similar to a virtual function. A pure virtual function is a virtual function that is required to be implemented by a derived class that is not abstract. This means that a derived class which is not abstract ***must*** implement the pure virtual function. We denote a pure virtual function by placing "= 0" in its declaration. For example, in the Shape class we used above, we could make *print* a pure virtual function as follows:

A non-abstract class doesn't have any pure virtual methods, like Circle in our example above.

```
class Shape{
    public:
        // Pure virtual function:
        virtual void print() = 0;
};
```

Abstract Base Classes

You want abstract base classes in situations where the parent is a class that you can't create without getting more specific. For example, you can't just have a generic vehicle. Instead, you would need to have a specific type of a vehicle, like a car or a boat. In C++, abstract base classes are defined simply as classes which have pure virtual functions. In the example above, Shape is an abstract base class because it has a pure virtual function. Abstract base classes, along with virtual functions, as you'd suspect, allow you to easily abstract concepts out into more generic ideas!

Practice Problems

For this section, we will be working with an abstract base class called Vehicle and creating functions for its derived classes.

For our Vehicle class, its *Drive* function is a pure virtual function, making it an abstract base class. However, we do need to define *Drive* for our 3 inherited classes, our Boat, our Car, and our Plane.

A function to calculate distance looks something like distance = velocity * time. In Vehicle.h we defined a DELTA_TIME variable to signify the time that is supposed to pass in between each movement call.

Implement the Drive Function

For our Car class, make your car move forward proportional to it's movement speed data member. This should be multiplied with

DELTA_TIME then added to its position data member inherited from the Vehicle class.

For the Plane class, assume that each engine gives you an additional 100 mph of speed. Use this to calculate the plane's speed, then from there you can find the distance moved, and add that to its inherited position data member.

For the Boat class, assume the boat's speed to be 20 mph if it's a sailboat, or 50 mph if it's a motor boat, as determined by its movementType_ data member. Use this to determine a speed, and again move the boat's inherited position.

Implement move_for_distance and move_for_time

Now, once you define these functions, go to your lab8_functions.cpp File, and fill in two functions: *move_for_distance* and *move_for_time*.

For *move_for_distance*, move the Vehicle passed in using its *Drive* function until the position has changed by the specified distance.

For *moveForTime*, move the Vehicle passed in using its *Drive* function until the desired amount of time has passed.

Race Time!

Implement the function *race*, which takes in a vector of shared pointers to Vehicles and a distance. the function should move each Vehicle over the specified distance and return the shared pointer to the Vehicle that moved over the distance in the least amount of time.