CS115 Lab: C++ Inheritance

Highlights of this lab:

In this lab, we will discuss one of the cornerstones of object-oriented programming – **inheritance** (the other two being encapsulation and polymorphism). Specifically, the following topics will be covered:

- What is inheritance
- How does inheritance work
- Access control
- Function overriding

Lab Exercise:

- Answer some questions on inheritance
- Complete a program that uses inheritance .

Click the little computer above for a detailed description.
NOTE: Your lab instructor will tell you what will be marked for this lab.

1. What is inheritance?

Inheritance is one of the most important concepts in object-oriented programming languages. One of the purposes of inheritance is to promote code reuse, because with inheritance, it is possible to define a class with some general characteristics (often referred to as the "base" or "parent" class) and then let other classes ("derived", "inherited", or "child" class) inherit these general characteristics from the base class. The derived class can then add more things that are unique to their own purposes. "That's quite a mouthful", you are saying..., but it's a common trait of human beings to make simple things seem complicated, right? Actually, this concept, like many other seemingly intimidating concepts, can be clearly explained with a simple example.

Here it is: suppose we would like to manipulate points in a 2D space. It is natural for us to define a class for this purpose. Let's call this class `TwoD`. The following is the definition for this class:

```
class TwoD // base class
{
   protected:
        double x, y; // x and y  coordinates
   public:
        // inline implementation of constructor
        TwoD(double i, double j):x(i), y(j){}

        // inline implementation of member functions
        void setX(double NewX){x = NewX;}
        void setY(double NewY){y = NewY;}
        double getX() const {return x;}
        double getY() const {return y;}
};
```

Suppose later on we decide to implement a class to deal with points in a 3D place. An intuitive way of implementing this `ThreeD` class would be:

```
// definition of ThreeD without using inheritance
class ThreeD
{
   protected:
        double x, y, z; // x, y, z coordinates
   public:
        // inline implementation of constructor
        ThreeD(double i, double j, double k):x(i), y(j), z(k){}

        // inline implementation of member functions
        void setX(double NewX){x = NewX;}
        void setY(double NewY){y = NewY;}
        void setZ(double NewZ){z = NewZ;}
        double getX() const {return x;}
        double getY() const {return y;}
        double getZ() const {return z;}
};
```

The code marked with red color shows the differences between the definition of the two classes. Actually, the only differences are that the `ThreeD` class has an additional member variable `z` and two additional member functions: `setZ()` and `getZ()` to set and print out the values of the `z` axis. In other words, the definition of the `ThreeD` class adds little new code to the definition of the `TwoD` class.

2. How does inheritance work?

In the definition of the `ThreeD` class, we have written some repetitive code and that's not wise. This can be fixed by using ***inheritance***. The syntax for inheritance is:

```
class derived-class:access base-class
{
      body of new class
};
```

`access` is the access control specifier which we will discuss shortly. For the time being, it suffices to know that `access` should be either `public`, `private`, or `protected`.

Here is the definition for the `ThreeD` class using inheritance:

```
// definition of ThreeD class using inheritance
class ThreeD:public TwoD
{
   private:
        double z;
   public:
        // Inline implementation of constructor.
        // Constructor of the base class is not inherited.
        // The following constructor of ThreeD class reuses the
        // constructor of the TwoD class and the only way values
        // can be passed to the TwoD constructor is via the use
        // of a member initialization list.
        ThreeD(double i, double j, double k):TwoD(i,j){z = k;}

        void setZ(double NewZ){z = NewZ;}
        double getZ() {return z;}
};
```

Notice that in the above definition of the `ThreeD` class, the `setX(), setY(), getX(), getY()` functions are not defined again, because these functions are inherited from the `TwoD` class. The result of inheritance in this case is these four functions do not need to be defined again in the `ThreeD` class.

Following is the complete program that you can compile and run:

```
// File name: ~ftp/pub/class/170/ftp/cpp/inheritance/Points.cpp
// Purpose:   Demonstrating the idea of inheritance
//

#include <iostream>
using namespace std;

class TwoD
{
   protected:
        double x, y; // x and y  coordinates
   public:
        // inline implementation of constructor
        TwoD(double i, double j):x(i), y(j){}

        // inline implementation of member functions
        void setX(double NewX){x = NewX;}
        void setY(double NewY){y = NewY;}
        double getX() const {return x;}
        double getY() const {return y;}
};

class ThreeD:public TwoD
{
   private:
        double z;
   public:
        // Inline implementation of constructor.
        // Constructor of the base class is not inherited.
        // The following constructor of ThreeD class reuses the
        // constructor of the TwoD class and the only way values
        // can be passed to the TwoD constructor is via the use
        // of a member initialization list.
        ThreeD(double i, double j, double k):TwoD(i,j){z = k;}

        void setZ(double NewZ){z = NewZ;}
        double getZ() {return z;}
};


int main()
{
        TwoD obj2(1, 1);
        ThreeD obj3(1, 2, 3);
        cout << "Coordinates for the 3d object are: " <<endl;
        cout << obj3.getX() << ", " << obj3.getY() << ", " << obj3.getZ()<< endl;
        return 0;
}
```

And here is a capture of the running result:

```
hercules[86]% g++ -o Points Points.cpp
hercules[87]% Points
Coordinates for 3d object are:
1, 2, 3
hercules[88]%
```

3. Access control

You probably have noticed that in the definition for `TwoD` and `ThreeD` classes, the member variables and member functions have ***access specifiers*** before them: `public`, `private`, or `protected`. These access specifiers control how these member variables and member functions can be accessed inside and outside the classes where they are declared and defined.

As mentioned previously, the access specifiers can also appear before the base class in the case of inheritance: here, again, is the syntax of inheritance:

```
class derived-class:access base-class
{
      body of new class
};
```

The access specifier used in inheritance, controls the way variables and functions in the base class are accessed by the derived class.

In the following section, we will discuss these two cases separately: 1) access specifiers within a class and 2) access specifiers in inheritance.

3.1 Access specifiers within a class

When a class member (variable and function) is declared as ***public***, it can be accessed by any other part of a program, including its derived class.

When a member is declared as ***private***, it can be accessed only by members of its class. It cannot be accessed by its derived class.

When a member is declared as ***protected***, it can be accessed only by members of its class. However, derived classes also have access to protected members of the base class.

When a member is declared **without** an access specifier, it is ***private*** by default.

3.2 Access specifiers in inheritance

The syntax for inheritance is:

```
class derived-class:access base-class
{
      body of new class
};
```

When a base class is inherited using ***public***, its ***public*** members become ***public*** members of the derived class, and its ***protected*** members become ***protected*** members of the derived class.

When a base class is inherited by use of ***protected***, its ***public*** and ***protected*** members become ***protected*** members of the derived class.

When a base class is inherited by use of ***private***, its ***public*** and ***protected*** members become ***private*** members of the derived class.

You might be wondering: what about the ***private*** members of the base class? Well, ***private*** members of the base class are private to that base class and therefore, ***not inherited***. You forgot it already. Didn't you? There are a lot of definitions to remember here.

By using access specifiers, a programmer can control exactly how each member should be accessed inside and outside of the class where they are defined.

In the next section, we introduce ***function overriding*** which looks similar to overloading, but should never be confused with overloading.

4. Function Overriding

Unlike overloading, overriding is used in case of inheritance. You have learned that a derived class can inherit both public and protected members (both variable and functions) from a base class. However, the derived class can also redefine the inherited member function. If the derived class defines a member function with has the same signature (number and type of parameters) as the base class, then the derived class is ***overriding*** the base class's member function.

Using the 2-D and 3-D program above, the `ThreeD` class inherits `setX(), setY(), getX(), getY()` from the base class. That's why in the definition for the ThreeD class, these functions do not have to be redefined. However, they CAN be redefined using the ***same*** function signature if necessary

In the following definition for the ThreeD class, we deliberately ***override*** the `setX()` function.

```
// definition of ThreeD class using inheritance
// Overriding the setX() function
class ThreeD:public TwoD
{
   private:
        double z;
   public:
        // Inline implementation of constructor.
        // Constructor of the base class is not inherited.
        // The following constructor of ThreeD class reuses the
        // constructor of the TwoD class and the only way values
        // can be passed to the TwoD constructor is via the use
        // of a member initialization list.
        ThreeD(double i, double j, double k):TwoD(i,j){z = k;}

        void setX(double NewX) {x = 2 * NewX;}
             // overriding the setX() function which was defined
             // in the TwoD class, because for some strange reasons
             // we would like to set x to be twice as large as the
             // value provided.
        void setZ(double NewZ){z = NewZ;}
        double getZ() {return z;}
};
```

If a derived class overrides a function that is overloaded in the base class all the overloaded functions with that name will be hidden, even if they have different signatures. This means you have to redefine all the overloaded functions if you want to use them. Click here for an example of this problem. Fixing this code is left to you as an exercise.

5. Lab Exercise -- C++ Inheritance

Question 1 - Access Levels

Fill in the blanks in the following table which describes the access levels in a derived class's members. i.e. State whether the member's access level is public, private, or protected - or, if it is not accessible!

Class Access specifier	Base Class Member Access Level	Derived Class Member Access Level
public	private public protected	⋮
protected	private public protected	⋮
private	private public protected	⋮

Question 2 - Programming Exercise

Start with the repl code provided to you.

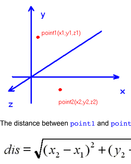
Finish the program so that it compiles and runs. The instructions are contained in the C++ file.

Your completed program should generate output similar to the following:

```
TwoD default constructor
This program asks for the coordinates of two points
in 3D space and calculates their distance.
Please enter the xyz coordinates for the first point:
1 1 1
Please enter the xyz coordinates for the second point:
2 3 4
TwoD constructor with two arguments
Distance is: 3.74166
```

The following refreshes your memory on how to calculate the distance between two points in 3D place.

Suppose we have the following two points in a 3D space:



The distance between `point1` and `point2` is:

$$dis = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

More details:

- In `main`:
  1. Create one `ThreeD` object using the default constructor.
     Use the setters to set x, y, and z.
  2. Create the second `ThreeD` object using the constructor with three arguments.
- in the `TwoD` class:
  1. Add a `cout` statement to the `TwoD` default constructor with a message "TwoD default constructor"
  2. Add a `cout` statement to other `TwoD` constructor with a message "TwoD constructor with two arguments"