Lab 9 – Inheritance

Overview

In this assignment, you're going to create a series of classes to represent simple 2D and 3D shapes. While the classes themselves are all simple, you will be using *inheritance* & the concept of *polymorphism* to store and use a group of these separate classes. You will also create classes which make use of multiple inheritance, as well as private inheritance.

A note about Pi and math functions

While many languages have math libraries of varying size and complexity built into them, C++ includes little, if any, such functionality. The value of PI, for example, is not officially defined in any C++ standard, although some header files provided by various creators of compilers do contain them.

For a standard in this assignment, you should use the following definition of PI, in one of two different ways:

```
#define PI 3.14159f // Do a simple find-and-replace of PI everywhere in your code
const float PI = 3.14159f; // Create an actual variable with PI as its value.
```

Also, in this assignment you should use the `float` data type instead of `double` to ensure proper results.

Lastly, there exists a header file called <math.h> which you can include to get access to some useful functions such as the pow() function, which lets you raise a value to an exponent and return the result.

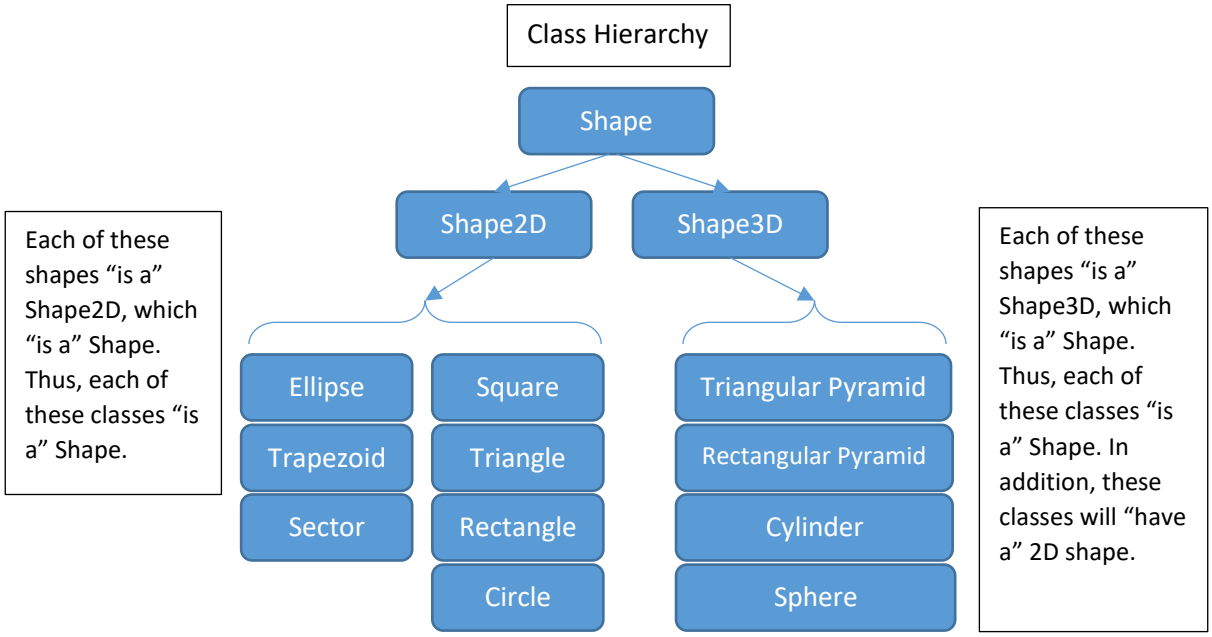
Description

The basic idea of inheritance is that you have a **base** class, and you create new classes which **derive** from that base class. One common use of inheritance is to reuse data and functions: if a base class defines those things, then the derived class doesn't have to; it inherits them, so in effect, you get "free" code in that class.

Another use of inheritance is to define a class which serves as what other programming languages would call an **interface**. In C++, the keyword interface doesn't exist, but some of the classes you create here will serve the same purpose. An interface defines how a class should look on the outside.

The first three classes shown below will serve as these interfaces; all the other classes will derive, ultimately, from those 3 base classes. In this way, all classes in this assignment are Shapes, even if their type is Triangle, Sphere, Circle, etc. Some types of inheritance model the "Is-a" relationship—a triangle IS A shape. A circle IS A shape as well. The "Is-a" model is heavily used in relational databases.

Other types of inheritance model a "has-a" relationship. A car, for example, has an engine—but it isn't an engine. This type of relationship can still be created, but the process is a bit different.



Class Description

Let's look at the first level of these classes: the Shape, Shape2D, and Shape3D:

```
class Shape
{
public:
    virtual void Scale(float scaleFactor) = 0;
    virtual void Display() const = 0;
};
```

Shape is an abstract base class. Because at least one of the functions in this class are **pure virtual functions** (i.e. virtual functions with = 0 at the end of the prototype, and no definition). This base says that all Shapes can scale, and display their data. Each derived class can define HOW they scale or display. (Each class should multiply all of its components by the passed-in scaleFactor variable).

```
class Shape2D : virtual public Shape
{
public:
    virtual float Area() const = 0;
    virtual void ShowArea() const = 0;
    bool operator>(const Shape2D &rhs) const;
    bool operator<(const Shape2D &rhs) const;
    bool operator==(const Shape2D &rhs) const;
};
```

Shape2D derives from the Shape class (a 2D shape IS A shape). Like the Shape class, it is also an abstract base class.

All Shape2D objects have an Area() & ShowArea() function. In addition, comparison operators are overloaded so that a Shape2D can compare itself to other shapes to test different relations, in terms of the shape's area. These functions will be defined only once, in this class. All classes deriving from Shape2D will inherit them.

The definition of these functions will simply be comparing the Area() of "this" to the Area() of the incoming object (greater than, less than, or equal to, respectively). Due to polymorphism, you will not have to define one of these comparisons for every possible combination of shapes, but instead you could compare the area of a circle against a triangle, square, hexagon, etc… any class deriving from Shape2D will be able to use these 3 functions as-is.

```
class Shape3D : virtual public Shape
{
public:
    virtual float Volume() const = 0;
    virtual void ShowVolume() const = 0;
    bool operator>(const Shape3D &rhs) const;
    bool operator<(const Shape3D &rhs) const;
    bool operator==(const Shape3D &rhs) const;
};
```

The same is true for the Shape3D class—it's an abstract base class that does not define two functions relating to volume, but instead requires child classes to define them. The overall concept is still the same though.

The comparison operators will have to be defined in terms of the Volume() function of these shapes. Like the Shape2D class, these can be defined once in this class, and all Shape3D objects can use them as-is.

2D Shapes

The next classes, derived from Shape2D, are going to be based on calculating the area of common shapes. The shapes and their respective area formulae are:

Shape	Area Formula
Square	Area = s^2 s = length of side
Rectangle	Area = w x h w = width h = height
Triangle	Area = $\frac{1}{2}bh$ b = length of base h = height
Circle	Area = πr^2 r = radius
Ellipse	Area = πab a = semi-minor axis b = semi-major axis
Trapezoid	Area = $\frac{1}{2}(a+b)*h$ a = side b = side h = height
Sector	Area = $\frac{1}{2}r^2\theta$ r = radius θ = angle in radians

Reference: https://www.mathsisfun.com/area.html (It also has an area calculator to test against your code)

Because they inherit from Shape2D, you will have to implement the four functions required by the abstract base classes. In addition, each class should have:

- A default constructor – initialize all member variables to 0

- A constructor which takes in parameters for each of the main components, like this (all variables are **floats**):
  `Square`(length of each side)
  `Rectangle`(width, height) [Note: these would be interchangeable in most, but not necessarily all scenarios]
  `Triangle`(base, height)
  `Circle`(just a radius)
  `Ellipse`(length of the major axis, length of minor axis)
  `Trapezoid` (length of one parallel side, length of the other, height)
  `Sector`(radius, angle in degrees)

Special Requirement: Sector

The Sector class will have a special requirement regarding handling of degrees/radians. The constructor should take in an angle in **degrees** and convert it to radians. This could be done using radians instead (or in addition to), but many math libraries will do this one of two ways, possibly both. There are **2π radians in a circle**, or 1 degree is equal to π/180.

3D Shapes

The 3D shapes will be a little different, partly due to how they will derive from the base classes. They will utilize **multiple inheritance**, and one of the base classes will derive using **private inheritance**.

Shape	Base classes	Volume
TriangularPyramid	`public Shape3D, private Triangle` It *IS A* Shape3D, and it *HAS A* triangle (the triangle is at its base) It has a private height variable	Volume = $\frac{1}{3}Ah$ A = area of base (you can get this from the Triangle you inherit!) h = height
RectangularPyramid	It **IS A** Shape3D, and it **HAS A** rectangle It also has a private height variable Note: length * width sounds like Area() too!	Volume = $\frac{1}{3}lwh$ l = length of base w = width of base h = height
Cylinder	It is a Shape3D, and it has a circle It has a height (private) variable too!	Volume = $\pi r^2 h$ r = radius h = height
Sphere	It is a Shape3D, and it has a circle It only needs the radius (private) of its circle to define its volume	Volume = $\frac{4}{3}\pi r^3$ r = radius

Because they inherit from Shape3D, you will have to implement the four functions required by the abstract base classes. In addition, each class should have:

- A default constructor
- A constructor which takes in a parameter for its main component (if it has one), and then anything its base class would need:
  `TriangularPyramid`(height of the pyramid, length of its triangular base, height of its base [which is on the ground])
  `RectangularPyramid`(pyramid height, length of its base, width of its base)
  `Cylinder`(height of the cylinder, radius of its circle)
  `Sphere`(just a radius)

Abstract Base Class

An **A**bstract **B**ase **C**lass (*ABC*) is a class you never want to create an instance of; it doesn't have enough information to be useful **on its own**. An ABC could have any number of functions or variables, whether public, protected, or private. You cannot create an instance of an abstract base class.

Perhaps you are creating a program that needs to store data about students and faculty members. They share similar data (name, age, email address, etc.), but have their own **unique** data as well. You might create 3 classes:

```
// Abstract class. "Just a Person" isn't enough data
class Person
class Student : public Person
class Faculty : public Person
```

To create an abstract base class, at least one function in the class must be a **pure virtual function**. A pure virtual function is one that:

1. Have the virtual keyword before the return type
2. Has = 0 at the end of the prototype
3. Has no function definition (i.e. no body) – the only exception to this is the destructor. You MUST have a body on a destructor, pure virtual or otherwise

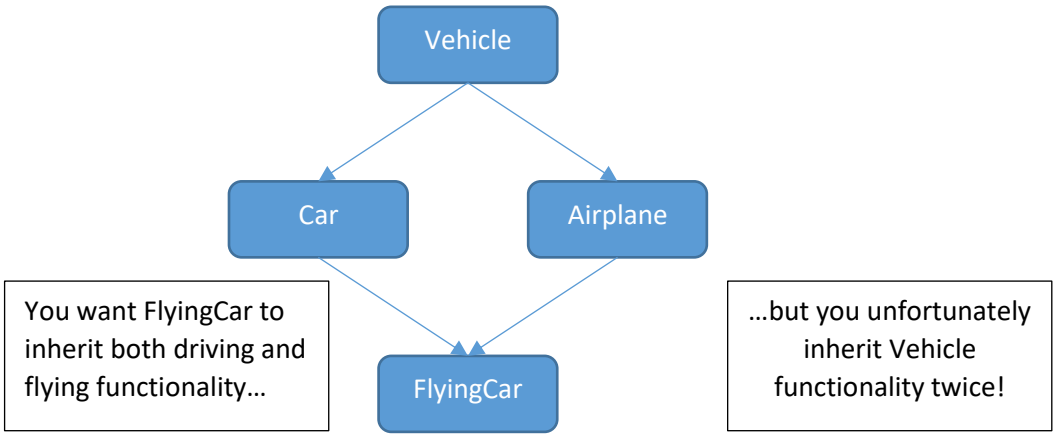
```
virtual void Display() const = 0;
```

Multiple Inheritance

While many languages do not allow multiple inheritance, C++ does! The basic concept is the same. You have multiple base classes, and a single class can derive from some or all of them, inheriting the data members and functions of each base class.

The Diamond Problem

An issue with multiple inheritance is the dreaded "diamond problem." This comes about when a derived class inherits from two base classes, which each derive from the same parent base class. Consider this:



How to resolve the diamond problem

You can fix the issue of the diamond problem by using **virtual** inheritance. In the above example, the class declarations might look like this:

```
class Vehicle
class Car       : public Vehicle
class Airplane  : public Vehicle
class FlyingCar : public Car, public Airplane
```

With virtual inheritance, the "in-between" classes would add the virtual keyword to the inheritance:

```
class Car      : virtual public Vehicle
class Airplane : virtual public Vehicle
```

This ensures that FlyingCar would only inherit ONE instance of anything Car and Airplane inherited from Vehicle.

For more information:

https://en.wikipedia.org/wiki/Virtual_inheritance
https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-where

Public and Private Inheritance

Perhaps the most common type of inheritance is public inheritance. It represents an "is-a" relationship between classes. The derived class is a base class.

For example:

```
class Car            : public Vehicle
class SavingsAccount : public BankAccount
class Button         : public UIControl
```

We would say that the Car (derived class) IS A Vehicle (base class). A SavingsAccount IS A BankAccount. A Button IS A UIControl, etc. If class inherits from another class using **private** inheritance, however, it models a "has a" relationship. This is also sometimes referred to as **composition**. For example:

```
// Private inheritance -- the car "has a" Engine
class Car : private Engine
```

The only thing the Car class can access of the Engine object is *public* members and functions — here's where the *accessor* functions of a class would come into play.

Accessing members of base class(es)

To access the functions or variables of a base class, you must specify the name of the class, and then the thing you are trying to access. So in the case of the car and its engine, you would do something like this:

```
void Car::SomeFunction()
{
      string maker = Engine::GetManufacturer();
}
```

Reference: https://isocpp.org/wiki/faq/private-inheritance

Polymorphism

One important concept in object-oriented programming is that a **base-class pointer** can point to any object which **derives** from that base class. For example:

```
class Car{};                        // Base class
class Mustang : public Car{};    // Derived class

// Set a base class pointer to an instance of the derived class
Car *someCar = new Mustang; // Valid due to inheritance
```

Following that same concept, we could extend that to something like the following:

```
Car *cars[3];
cars[0] = new Car;
cars[1] = new SportsCar;
cars[2] = new LuxurySedan;
```

What polymorphism allows you to do is call a function from one of those objects and **depending on what the pointer is pointing to** the correct function will be called.

```
class Car {
public:
      virtual void Identity()
      { cout << "I'm just a car!"; }
};
class SportsCar : public Car {
public:
      void Identity()
      { cout << "I'm a sports car!"; }
};
class LuxurySedan : public Car {
public:
      void Identity()
      { cout << "I'm a luxury car!"; }
};
```

Given the previous array, a line of code like the following would print out “I’m a sports car!”:

```
cars[1]->Identity(); // cars[1] points to a SportsCar
```

Polymorphism is the concept that allows you to write code like that, without knowing exactly what type of object you are pointing to. Since cars[1] points to a SportsCar, the call to Identity() will determine that the SportsCar version of the function should be used. The **virtual** keyword is required on the base class version of the function in order to make this functionality possible.

For more on polymorphism: http://www.cplusplus.com/doc/tutorial/polymorphism/

Bringing it all together

While it may seem like a lot of work to create such simple classes, once this framework is in place, you can very easily modify the class hierarchy with minimal effort (such as changing code elsewhere). Consider the following code:

```
Shape *shapes[] =
{
      new Square(2.4f),
      new Rectangle(3, 5),
      new Triangle(4.2f, 6),
      new Circle(2),
      new Trapezoid(2, 6, 2.3f),
      new Sector(2, 35)
};
for (int i = 0; i < 6; i++)
{
      shapes[i]->Display();
      cout << endl;
}
```

```
// Mix 2D and 3D shapes
Shape *shapes[] =
{
   new Circle(3.02f),
   new TriangularPyramid(4.5f, 1, 4),
   new Sector(3, 18),
   new RectangularPyramid(3.23f, 2, 3)
   new Trapezoid(2, 6, 2.3f),
};
for (int i = 0; i < 5; i++)
{
      shapes[i]->Display();
      cout << endl;
}
```

Nowhere in the loops is there any code checking the types of the objects (Shape2D, Shape3D, it doesn't matter). It simply uses a function, and because of inheritance, polymorphism, and virtual functions, each object behaves as it should. In this particular case the behavior of each object is simple, and very similar, but this concept allows us to change any individual object's behavior, without modifying the code you see here. And if you were to write a function like this:

```
void SomeFunction(vector<Shape *> shapes)
{
      for (unsigned int i = 0; i < shapes.size(); i++)
            shapes[i]->Display();
}
```

You can further reduce the amount of information you need to have in order to write your code. How many objects are in the shapes vector? The size() function will tell us. What shapes, exactly, are in that vector? Don't need to know, don't need to care. The importance of this concept, of working with objects, **WITHOUT KNOWING ALL OF THAT OBJECT'S DETAILS**, cannot be overstated. Many things you will work on later in your programming career will rely on this concept.

Tips

A few tips about this assignment:

- Start with one class at a time. Since Square derives from Shape2D, which derives from Shape… start with the Shape class first. Then Shape2D, then Square, then all the other 2D shapes.
- Similarly, if a class like TriangularPyramid derives from both Shape3D and Triangle, it's probably a good idea to complete those classes first.
- If you want to add any other functionality to these classes in order to help you get the job done, do it! Helper functions, constant values (pi, degrees to radian conversions, etc), go for it! The interface of the class indicates
- **what** it can do. It says nothing about **how** that gets done.

Sample Output

2D Shapes:

```
The area of the Square is : 1.210
Length of side: 1.100

The area of the Rectangle is : 7.260
Length: 2.200
Width: 3.300

The area of the Triangle is : 12.100
Base: 4.400
Height: 5.500

The area of the Circle is : 136.848
Radius: 6.600

The area of the Ellipse is : 212.874
Length of semi-major axis: 7.700
Length of semi-minor axis: 8.800

The area of the Trapezoid is : 111.100
Length of side A: 9.900
Length of side B: 10.100
Height: 11.110

The area of the Sector is : 16.831
Radius: 12.120
Angle in radians: 0.229
Angle in degrees: 13.130
```

3D Shapes:

```
The volume of the Triangular Pyramid is : 1.331
The height is: 1.100
The area of the Triangle is : 3.630
Base: 2.200
Height: 3.300

The volume of the Rectangular Pyramid is : 53.240
The height is: 4.400
The area of the Rectangle is : 36.300
Length: 5.500
Width: 6.600

The volume of the Cylinder is : 1873.294
The height is: 7.700
The area of the Circle is : 243.285
Radius: 8.800

The volume of the Sphere is : 4064.379
The area of the Circle is : 307.907
Radius: 9.900
```