Object-Oriented Programming  
Tutorial 07 - Inheritance

## Introduction

In an earlier tutorial you looked at the concept of **class associations**, that is classes that contain other classes. These associations, in the forms of **composition** and **aggregation**, are one way to build complex classes from simpler ones. However, this way of constructing classes is not the only option available when designing object oriented systems. Another, very powerful, object oriented feature of C++ is called **inheritance**. Inheritance allows you to **extend** an existing class to create an entirely new class. The newly **derived** class contains all of the attributes and methods of the original class plus any additional ones that are defined for the new one. Inheritance allows you to create a hierarchy of classes that start out relatively non-specific, then become increasingly specialised. This not only gives the advantages of code reuse, as specialised classes reuse the code in the classes they extend from, but also allows the use of **polymorphism**. This will be discussed in the next tutorial.

### Inheritance

Class inheritance simply allows you to create a new class from an existing one. The new class is said to be **derived** from the existing one and **inherits** all its members and attributes. The original class is called the **parent class, a**lternatively **base** class or **super** class. The new class is called the **child class**, alternatively **derived** class or **sub** class. A child class has all of the attributes and methods of its parent class and usually the implementation of a child class generally extends and/or specialises the capabilities of the parent class.

Usually, the design of a set of classes that uses inheritance starts with a general class that has methods and attributes that can apply to **many** different sorts of classes. Then new classes are derived from the general class further to create more **specialised** classes.

An example would be to consider creating a set of classes that can represent any kind of 2D shape that you might want to draw, e.g. circle, rectangle, etc. *One* way to do this would be to create a separate class for each type of shape, however, this would involve a lot of duplication of code. A much better way to define the classes would be first to consider what are the **common properties and operations** that you require for **all** shapes and then create a **Shape** class that contains them. A partial **Shape** class diagram might look like this:

**# m\_x\_position : int**

**# m\_y\_position : int**

**# m\_colour : string**

**+ Shape()**

**+ Shape(x: int, y: int, colour: string)**

**+ SetPosition(x: int, y:int): void**

**Shape**

The Shape class contains attributes and methods that might be useful in representing and manipulating any kind of shape, and of course there are likely many more than the simple example above.

You may have noticed the new # symbol in the above class diagram. This represents the **protected** **access specifier** which will discussed later.

Now that a Shape class has been defined you can now use inheritance to specialise into specific shapes that you require. For example, here is a class diagram for a **Rectangle**, along with the inheritance relationship to the Shape class. This is represented by an arrow with an **unfilled** arrowhead:

**-m\_width : int**

**-m\_height : int**

**+Rectangle(x:int, y:int, w:int,**

**h:int, c:string)**

**+SetHeight(height: int): void**

**+SetWidth(width: int): void**

**+DisplayArea(): void**

**Rectangle**

**# m\_x\_position : int**

**# m\_y\_position : int**

**# m\_colour : string**

**Shape**

**+ Shape()**

**+ Shape(x: int, y: int, colour: string)**

**+ SetPosition(x: int, y:int): void**

The specialisation of the Rectangle class is the addition of m\_width and m\_height attributes that are specific to a rectangle and a **DisplayArea**() method that can use the properties of a Rectangle object to display the area of the rectangle. An important point to remember is that since the child class Rectangle is derived from its parent Shape, the Rectangle class **contains all the attributes and methods that are in the Shape class**. Therefore the Rectangle class has an **m\_x\_position** attribute and **SetPosition()**method, for instance. The real power of using inheritance starts becoming obvious when you add another shape to the class diagram, this time a **Circle** class:

**-m\_radius : int**

**+Circle(x:int, y:int,**

**r: int, c:string)**

**+SetRadius(height: int): void**

**+DisplayArea(): void**

**Circle**

**-m\_width : int**

**-m\_height : int**

**+Rectangle(x:int, y:int, w:int,**

**h: int, c:string)**

**+SetHeight(height: int): void**

**+SetWidth(width: int): void**

**+DisplayArea(): void**

**Rectangle**

**# m\_x\_position : int**

**# m\_y\_position : int**

**# m\_colour : string**

**Shape**

**+ Shape()**

**+ Shape(x: int, y: int, colour: string)**

**+ SetPosition(x: int, y:int): void**

A Circle shape is distinctly different from a Rectangle as it is being represented by a **radius** instead of width and height and the **DisplayArea**()method will be different because of this. However, the class diagram inheritance relationship makes it clear which are the common elements between the two different classes, and that both Rectangle and Circle share these by inheriting them from the parent class Shape.

### Implementing Inheritance

In order to show how inheritance is implemented let's start by creating a Shape class from the class diagram in the previous section:

// Shape.h

#pragma once

#include <string>

using namespace std;

class Shape

{

protected:

int m\_x\_position;

int m\_y\_position;

string m\_colour;

public:

Shape(void);

Shape(int x, int y, string colour);

void SetPosition(int x, int y);

};

// Shape.cpp

#include "Shape.h"

Shape::Shape(void)

{

m\_x\_position = 0;

m\_y\_position = 0;

m\_colour = "none";

}

Shape::Shape(int x, int y, string colour)

{

m\_x\_position = x;

m\_y\_position = y;

m\_colour = colour;

}

void Shape::SetPosition(int x, int y)

{

m\_x\_position = x;

m\_y\_position = y;

}

This looks like any other class definition, except with the addition of a new keyword, the protected: access specifier, which was denoted on the class diagram using the # symbol. This differs from the private specifier in that it **allows child classes that inherit from Shape to access these elements as if they were public**, whilst still remaining **private for everything else**. The real benefit of this type of access will become more obvious when a child class that inherits from Shape is shown later.

You *could* create and use a Shape object just like any other object, e.g.

// in main()

Shape my\_shape(6, 8, "green");

my\_shape.SetPosition(2, 1);

but on its own a Shape object doesn't really mean anything. Let's create a **Rectangle child class** using the **Shape class as a parent**, starting with the class declaration:

// Rectangle.h

#pragma once

#include "Shape.h"

class Rectangle : public Shape

{

private:

int m\_width;

int m\_height;

public:

Rectangle(int x, int y, int w, int h, string colour);

void SetHeight(int height);

void SetWidth(int width);

void DisplayArea(void);

};

First of all, you can see that the Rectangle header file includes the Shape header file, which makes sense as the Rectangle class wants to inherit from the Shape class so it needs to know what a Shape is. Secondly, notice the new syntax after the class name - class Rectangle : public Shape. This sets the inheritance relationship to say that **Rectangle inherits from Shape.** The word **public** is an **access specifier**.

*Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you see or use the other types of inheritance. When you inherit a base class publicly, inherited public members stay public, and inherited protected members stay protected. Inherited private members, which were inaccessible because they were private in the base class, stay inaccessible.*

*With Private inheritance all members from the base class are inherited as private. With protected inheritance, which is hardly ever used, private is inaccessible and all other members are inherited as protected.*

Once you have declared the Rectangle class to inherit from Shape here is how you *might* implement the methods:

// Rectangle.cpp

#include "Rectangle.h"

#include <iostream>

using namespace std;

Rectangle::Rectangle(int x, int y, int w, int h, string colour)

{

// initialise attributes declared in Shape -

**// NOTE don't do this in practice, use initialisation list (see next section)**

m\_x\_position = x;

m\_y\_position = y;

m\_colour = colour;

// initialise attributes declared in Rectangle

m\_height = h;

m\_width = w;

}

void Rectangle::SetHeight(int height)

{

m\_height = height;

}

void Rectangle::SetWidth(int width)

{

m\_width = width;

}

void Rectangle::DisplayArea(void)

{

cout << m\_height \* m\_width << endl;

}

As you can see this looks just like any other class definition, but the important thing to realise is the usage of the attributes from the Shape class in the constructor, highlighted in the above code. Because Rectangle inherits from Shape **and they have been given protected access**, these attributes are essentially now part of the Rectangle class and can be used just like any other attribute. If the Shape class had any public attributes then Rectangle could also access them directly, however Rectangle would not be able to directly access private attributes.

The same applies to inherited public and protected methods:

// main.cpp

#include <iostream.h>

#include "Rectangle.h"

using namespace std;

int main()

{

Rectangle my\_rectangle(2, 4, 11, 22, "blue");

my\_rectangle.DisplayArea(); // Rectangle method

my\_rectangle.SetPosition(2, 1); // Shape method

my\_rectangle.SetHeight(32);

my\_rectangle.DisplayArea();

cin.get();

}

The above code shows how you can use the public method SetPosition() from Shape to set the position values of the my\_rectangle object.

### Parent Class Attribute Initialisation

When an object from a child class is declared its constructor is called just like a normal object. However, part of the derived object is made up of its parent class, so where does the constructor for the parent portion of the object get called? For instance, in the Rectangle child class definition there is no reference to any of the Shape constructors, so how and when does the Shape portion of the object get initialised? In this case what happens is the compiler can tell that the class is a child class, so first of all it reserves memory for the parent class and then automatically calls the default constructor to initialise this memory. This is why a default constructor was declared in Shape, otherwise the code would not compile. Then memory is reserved for the Rectangle object and its constructor is called to initialise its values.

The above process leaves you with a fully and correctly initialised object, but if you look at the Rectangle constructor you will see that you have had to initialise the Shape attributes with the values passed in to the Rectangle constructor. This is usually not the best way to go about this as you are likely duplicating initialisation code that is already in a Shape constructor. A better way to do this is to use the **class initialisation list**, an example of which is below for Rectangle:

Rectangle::Rectangle(int x, int y, int w, int h, string colour) : Shape(x, y, colour)

{

**// don't need to initialise Shape attributes, done in Shape constructor**

**// initialise Rectangle specific attributes**

m\_height = h;

m\_width = w;

}

The highlighted code shows the syntax for this type of initialisation, it is similar to the syntax used to create an object using a constructor. Essentially, what this does is tell the compiler to use the given Shape constructor to initialise the Shape attributes of the Rectangle class, using the relevant parameters from the Rectangle constructor parameters - in this case x, y and colour. You should usually use this method of initialising parent class attributes instead of explicitly setting them in a child class constructor. The other way will work just fine but it is inefficient because the attributes are being set to the same value more than once.

### Inheritance Chains

Inheritance is not limited to a single parent-child relationship, a child class can itself be the parent of another class, that new class can be a parent of another, and so on. As a simple example of this imagine that you have a situation where you want to specialise the Rectangle class further to create a specific Square class. This could be achieved as follows:

// Square.h

#pragma once

#include "Rectangle.h"

class Square : public Rectangle

{

private:

int side\_length

public:

Square(int x, int y, int side\_length, string colour);

};

// Square.cpp

Square::Square(int x, int y, int side, string colour) : Rectangle(x, y, colour)

{

side\_length = side;

}

This Square class is very simple, and not that useful, but it does illustrate the concept of how an inheritance chain works. In this case the class has only one additional attribute and the constructor simply sets the side length before using the initialisation list to call its parent’s constructor which in turn calls its Shape parent constructor. Here is how the overall class relationships would look after the addition of the Square child class:

**Shape**

**Rectangle**

**Circle**

**Square**

Obviously, most class hierarchies would be more extensive than this.

### When to use Inheritance, Composition and Aggregation

Inheritance can be thought of as an **'is a'** relationship between classes, so, for instance, the Rectangle class **is a** Shape, and the Square class **is a** Rectangle. The type of relationships that you identify when designing a set of classes is extremely important, as this will eventually dictate how well your classes operate together. A poor choice of class relationships will likely end up with poorly written, ineffective or non-functional code.

As you saw in a previous tutorial, you can create classes that contain other classes using composition and aggregation. Inheritance and the class associations of composition and aggregation may seem very similar, but they do in fact represent distinctly different types of class relationships. A good way to differentiate between them is to determine if the relationship is based on **identity** or **properties**, i.e. an **inheritance 'is a'** relationship, or if the relationship between classes is based on **containment**, i.e. a **composition** **'part of'** or **'contains a'** relationship, or if the relationship is a **reference** to another object, i.e. an **aggregate 'refers to'** or **'linked to'** relationship. To demonstrate the difference:

* First consider the Rectangle and Shape class relationship - a Rectangle doesn't **contain a** Shape or **refer to a** Shape, it **is a** Shape, hence **inheritance**.
* Alternatively consider the Car and Engine classes that were discussed when introducing the concept of composition and aggregation - the Engine isn't a type of Car, nor, in the given example, is the Engine **linked to** a Car, the Engine is **part of** **a** Car, hence **composition**.
* Finally, a Person **isn't a** Car, nor is a Person a **part of a** Car, a Person is **linked to a** Car, hence **aggregation**.

Careful consideration of the correct type of relationships between classes is crucial to a good design, which almost always leads to a more straightforward implementation.

## Exercises

**In order to aid understanding of how the exercises work, use the debugger to step through all of the code you write for the exercises.**

**You are advised to write notes on all aspects of the tutorial and exercises in your notebooks. This can then be used to help with your assignments.**

#### Exercise 01a

1. Create a new project and implement the Shape and Rectangle classes. Use the class initialisation syntax to initialise the Shape attributes in the Rectangle constructor.
2. In main() create a Rectangle object, then call the DisplayArea() method. Check the area is correct for the given width and height of the rectangle.
3. Change the width of the rectangle and output the area again. Check that it is correct.
4. Change the height of the rectangle and output the area again. Check that it is correct.
5. Create a breakpoint just after the creation of the Rectangle object and run the program. Inspect the watch window for the object and see how the attributes are organised. Write in your notebook how this differs from an object that has been added to a class using composition or aggregation.
6. Put breakpoints in all the constructors for both objects and run the program. Note down the order the constructors are called.

#### Exercise 01b

Create a Circle class that inherits from Shape and test that it works correctly.

#### Exercise 01c

Create a Square class that inherits from Rectangle and test that it works correctly. Use breakpoints in all the constructors to see the order in which they are called. Record this in your notebook.

#### Exercise 01d

1. Design a Triangle class that inherits from Shape. The attributes added should be m\_base\_width and m\_height, and it should also have a DisplayArea() method that outputs (m\_base\_width \* m\_height) / 2.
2. Draw the class diagram and inheritance relationship. Implement, and then test that it works correctly.

#### Exercise 02

1. Design an Animal class that represents some of the general properties of animals and draw the class diagram.
2. Design Dog and Cat classes that inherit from Animal, adding attributes and methods that are unique to both. You may find you need to redesign the Animal class, if so document and explain the changes.
3. Design a Fish class that inherits from Animal. The considerations for this new class may require further design changes to the Animal class, and possibly the Dog and Cat classes. Document and explain any changes.
4. Implement and test the classes you have designed. Document and explain any changes to the design that you need to make in order to get a fully functional system.

#### Exercise 03

1. Design a basic Person class, including attributes and methods for name, gender, height, weight and NI number.
2. Design a FactoryWorker class that inherits from the Person class, including attributes job title and hourly wage, and a method WeeklyWage() that returns the pay for a 40 hour week.
3. Design a Footballer class that inherits from the Person class, including yearly wage, goals scored, matches played, shirt number and position. Add methods, including one to return the average number of goals scored per game.
4. Implement and test the classes you have designed. Document and explain any changes to the design that you need to make in order to get a fully functional system.

#### Exercise 04

1. Design an Astronaut class, with extra attributes such as number of flight, hours in space, etc. Use the Person class from the previous exercise.
2. Design a Rocket Engine class that has attributes such as fuel level, thrust, etc., and methods such as start(), stop(), fuel\_left()**,** get\_thrust() and others that you think might be useful.
3. Design a SpaceCraft class, that has several Astronauts and several Rocket Engines. Consider what the correct relationships should be. Add other attributes such as current payload weight, speed, height, number of orbits etc. Add methods launch(), land(), fall() and any others you can think of.
4. Ensure your class diagrams show all class associations.
5. Implement and test all the classes, assume you are launching the SpaceCraft to a specific height which is a successful orbit. Once started the Rockets should produce thrust until they are out of fuel. The SpaceCraft cannot launch without a pilot, or if there are no engines with fuel. The launch should continue until all Rockets are out of fuel. Each Rocket should increase the speed of the SpaceCraft based on the thrust. If the SpaceCraft doesn't reach orbit it should fall to the ground. If it does reach orbit, it should complete a number of orbits and then land.