Object-Oriented Programming  
Tutorial 08 - Overriding Methods & Introduction to Polymorphism

### Introduction

The last tutorial looked at how to create a hierarchy of classes using inheritance. This tutorial extends this concept by introducing the technique of **overriding methods**, which is a mechanism for child classes to rewrite methods from the parent class. Additionally, the tutorial introduces the **basics of** **polymorphism** which is an object oriented methodology that allows the manipulation of objects from different classes in an inheritance hierarchy as if they were the same type of object.

### Overriding Class Methods

If there is a class inheritance hierarchy where the child class extends the capabilities of the parent class, it is often the case that there are methods in the parent class that require some form of additional functionality in the child class. As an example, the Shape class from the previous tutorial could have a new method called DisplayInfo(), which outputs all of the attributes of a Shape:

class Shape

{

...

public:

...

void DisplayInfo(void);

};

void Shape::DisplayInfo(void)

{

// output attributes from attributes declared in Shape

cout << "Shape Info - X: " << m\_x\_position << ", Y: " << m\_y\_position;

cout << ", Colour: " << m\_colour << endl;

}

This is a useful method, so the same could be done for the derived class Rectangle:

class Rectangle

{

...

public:

...

void DisplayInfo(void);

};

void Rectangle::DisplayInfo(void)

{

// output attributes from attributes declared in Rectangle

cout << "Rectangle Info - Height: "<< m\_height << ", Width: " << m\_width << endl;

// output attributes from attributes declared in Shape

cout << "Shape Info - X: " << m\_x\_position << ", Y: " << m\_y\_position;

cout << ", Colour: " << m\_colour << endl;

}

Notice that the return type, the method name and the parameter list are the same for both the parent and child class versions of the method. This technique of using a method in a child class that has the same declaration as a method in its parent class is called **overriding**. Don't confuse overriding of methods with function and method **overloading**:

* Overriding is the **replacement of** a parent class method in a child class. Both methods have the same name, parameters and return types, but belong to and are defined in different classes.
* Overloading is the **addition** **of new** functions or methods with the same name, but with differing parameter types. Which function/method is called is determined by the types in the parameter list, e.g. constructors

Here is some code that calls the DisplayInfo()method for a Shape object and a Rectangle object:

Shape my\_shape(1, 3, "red");

cout << endl << "my\_shape info:" << endl;

my\_shape.DisplayInfo();

Rectangle my\_rectangle(2, 4, 10, 12, "blue");

cout << endl << "my\_rectangle info:" << endl;

my\_rectangle.DisplayInfo();

This code would output the following text:

my\_shape info:

Shape Info - X: 1, Y: 3, Colour: red

my\_rectangle info:

Rectangle Info - Height: 12, Width: 10

Shape Info - X: 2, Y: 4, Colour: blue

Depending on the type of object the appropriate DisplayInfo()method is called.

### Calling Overridden Parent Class **Methods**

There is clearly duplication of code to output the Shape class attributes in the Rectangle DisplayInfo()method. As noted before, most recently during parent class initialisation in a previous tutorial, it is usually not a good idea to have duplicated code for memory usage and code maintenance reasons. It would seem like good sense to use the Shape version of DisplayInfo() to display the attributes from Shape when the Rectangle class version is called. Although it looks like this might not be possible, as Rectangle overrides the Shape version of DisplayInfo(), it is in fact very easy to do. All that is needed is the **scope resolution operator (**::**)** in order to **call the parent class version** of the method. The Rectangle DisplayInfo()method can now be rewritten as:

void Rectangle::DisplayInfo(void)

{

// output attributes from attributes declared in Rectangle

cout << "Rectangle Info - Height: "<< m\_height << ", Width: " << m\_width << endl;

// call parent class version of method to output the atributes from Shape

Shape::DisplayInfo();

}

### Polymorphism

Polymorphism essentially means "the ability to assume many forms". In the context of object oriented programming polymorphism allows the management of objects from different classes derived from the same parent class(es) as if they were the same type. Simply put, it means that a parent class pointer can be used to point to an object of any of its child classes. As an example, assuming that a Circle class has been created that inherits from Shape with an m\_radius attribute, the following code creates a Rectangle object and a Circle object, then uses a Shape pointer to first point at the Rectangle and then at the Circle object. After both pointer assignments, the SetPosition() method is called:

Rectangle my\_rectangle(2, 4, 10, 12, "blue");

Circle my\_circle(4, 8, 22, "green"); // parameters x, y, radius, colour

Shape \*p\_any\_shape = &my\_rectangle;

p\_any\_shape->SetPosition(1, 1);

p\_any\_shape = &my\_circle;

p\_any\_shape->SetPosition(2, 2);

If this code is executed and a breakpoint set just after the final SetPosition() then the debugger Watch window will show that the Rectangle and Circle objects positions have correctly been set to the values set **via the Shape pointer method calls**. It is important to note that no Shape object has been created and in this instance the Shape pointer does not point at a Shape object; it only gets the address of objects from child classes. This is a simple example of how polymorphism works, but it will be useful to see it in action in a more likely real-world scenario. In this case, a small array that can store a list of Shapes:

Rectangle my\_rectangle(2, 4, 10, 12, "blue");

Circle my\_circle(4, 8, 22, "green");

Rectangle my\_rectangle2(3, 5, 11, 13, "purple");

Circle my\_circle2(5, 6, 2, "yellow");

const int NUM\_SHAPES = 4;

Shape \*p\_Shapes[NUM\_SHAPES];

p\_Shapes[0] = &my\_rectangle;

p\_Shapes[1] = &my\_circle;

p\_Shapes[2] = &my\_rectangle2;

p\_Shapes[3] = &my\_circle2;

cout << "Initial values:" << endl << endl;

for(int i = 0; i < NUM\_SHAPES; i++) p\_Shapes[i]->DisplayInfo();

for(int i = 0; i < NUM\_SHAPES; i++) p\_Shapes[i]->SetPosition(i, i);

cout << endl << "Final values:" << endl << endl;

for(int i = 0; i < NUM\_SHAPES; i++) p\_Shapes[i]->DisplayInfo();

The above code creates two Rectangle objects, two Circle objects and an array of Shape pointers. Each element of the array is assigned the address of one of the Rectangle or Circle objects. It's worth pointing out that there is now an array that contains **two different types of objects**. Finally several for loops are called which loop through each element of the array and call DisplayInfo() and SetPosition() for each one. The output from the above code would look like this:

Initial values:

Shape Info - X: 2, Y: 4, Colour: blue

Shape Info - X: 4, Y: 8, Colour: green

Shape Info - X: 3, Y: 5, Colour: purple

Shape Info - X: 5, Y: 6, Colour: yellow

Final values:

Shape Info - X: 0, Y: 0, Colour: blue

Shape Info - X: 1, Y: 1, Colour: green

Shape Info - X: 2, Y: 2, Colour: purple

Shape Info - X: 3, Y: 3, Colour: yellow

The yellow highlights show the changes to the values of the objects. The crucial thing to understand here is that the code in the for loops is using, accessing and manipulating two different types of objects, Rectangles and Circles, using the same piece of code and without having to know in advance what specific type of object it has to deal with beforehand. Even better, if a new Shape were to be created, e.g. a Triangle class, then a Triangle object could be added to this list without having to change any code in Shape or in the loops that manipulate the Shape array.

This ability to store and manipulate different types of objects from the same inheritance hierarchy, and the ability to use code that can treat them as the same type of objects, is fundamental to polymorphism. Polymorphism is enormously useful when developing complex systems, in terms of code simplicity and understanding, ease of use, and cohesive data organisation and processing.

### Virtual Methods

The code in the preceding section demonstrates the basics of how polymorphism works, but in its present form it is somewhat limited. The array of Shape objects stored both Rectangle and Circle objects and the DisplayInfo() method was called for each object, but it was the **Shape class version** of the method which was called. In most situations this is probably not what was wanted, it's more likely that the specific overridden method from the child object should be called. In order to do this the parent class needs to declare the method as a **virtual method**. In simple terms, this means that any call to a method **via a parent class pointer** will check to see if a child class has overridden the method, and if so the child class method will be called instead.

This is best illustrated by example:

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);

virtual void DisplayInfo(void);

};

The only change needed to the Shape class is the addition of the virtual keyword immediately before the return type of the Shape DisplayInfo()method.

If the code from the previous section is executed with the addition of the virtual keyword as shown above, and assuming the Circle class has a DisplayInfo() method of its own, then the following would be output:

Initial values:

Rectangle Info - Height: 12, Width: 10

Shape Info - X: 2, Y: 4, Colour: blue

Circle Info - radius: 22

Shape Info - X: 4, Y: 8, Colour: green

Rectangle Info - Height: 13, Width: 11

Shape Info - X: 3, Y: 5, Colour: purple

Circle Info - radius: 2

Shape Info - X: 5, Y: 6, Colour: yellow

Final values:

Rectangle Info - Height: 12, Width: 10

Shape Info - X: 0, Y: 0, Colour: blue

Circle Info - radius: 22

Shape Info - X: 1, Y: 1, Colour: green

Rectangle Info - Height: 13, Width: 11

Shape Info - X: 2, Y: 2, Colour: purple

Circle Info - radius: 2

Shape Info - X: 3, Y: 3, Colour: yellow

The addition of the virtual keyword to the Shape parent class means that, instead of the Shape version of the method being called when the method is called via a Shape pointer, the child class version is called. This means that, as long as there is a common method or set of methods, collections of similar objects from an inheritance hierarchy can be stored and manipulated together with the processing being specific to each type of object.

### Virtual Destructors

If there is a destructor in a child class, when an object of that class type goes out of scope then the destructor for the child class is called first, and then any destructor for the parent class is called. The same thing happens if a dynamically created object is **deleted via a pointer of that same class type.** However, when using polymorphism, i.e. referencing a child class object **via a parent class pointer**, there is a problem when calling delete on the object via the parent class pointer. This problem is that the delete call will only invoke the parent class destructor, as the access to destructors works in the same way as for normal methods. This can of course lead to memory leaks as any dynamically created data in the child part of the class will not be correctly freed.

This problem can be demonstrated with the following code:

// in Shape.cpp, assuming Destructor declared in Shape.h

Shape::~Shape()

{

cout << " Shape Destructor called" << endl;

}

// in Rectangle.cpp, assuming Destructor declared in Rectangle.h

Rectangle::~Rectangle()

{

cout << " Rectangle Destructor called" << endl;

}

// main.cpp

Rectangle \*p\_rect = new Rectangle (8, 9, 55, 33, "orange");

Rectangle \*p\_rect2 = new Rectangle (11, 129, 525, 343, "cyan");

Shape \*p\_shape = p\_rect2;

cout << "Deleting via Rectangle pointer" << endl;

delete p\_rect;

cout << endl << "Deleting via Shape pointer" << endl;

delete p\_shape;

The output would be as follows, which shows that the Rectangle destructor isn't called when deleting via a Shape pointer:

Deleting via Rectangle pointer

Rectangle Destructor called

Shape Destructor called

Deleting via Shape pointer

Shape Destructor called

This problem can be resolved using the virtual keyword in the same way as for virtual methods, by placing the virtual keyword before the parent class destructor declaration and creating a **virtual destructor**.

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);

virtual void DisplayInfo(void);

virtual ~Shape();

};

Now if delete is called on a child object referenced via a parent class pointer the compiler will check to see if there is a child class destructor and call it first, and only then call the parent class destructor. With the addition of the virtual keyword to the destructor the output would now correctly be:

Deleting via Rectangle pointer

Rectangle Destructor called

Shape Destructor called

Deleting via Shape pointer

Rectangle Destructor called

Shape Destructor called

It's important to be clear that this only matters if an object is being deleted **via a parent class pointer**. If a static object is going out of scope, or a dynamic object that is referenced by a pointer that points to the same class type is being deleted via that pointer, then the child class destructor is correctly called first and then the parent class destructor.

## 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 01

1. Create a new project and save it. **Copy** the **.cpp** and **.h** files from a working version of Tutorial 07 Exercise 01 into the new project directory. Drag these files from the **new project directory** into the Solution Explorer tab in Visual Studio ( be **VERY** careful **NOT** to drag them from the Tutorial 08 Exercise 01 project folder, otherwise your new project will use the old files and overwrite them). Test that the project works by building and running it. It should work the same as the previous exercise. If not, delete the new project folder and begin again.
2. Implement the overloaded methods from the tutorial.
3. Add the other child classes from the inheritance tutorials, and add the same overloaded methods.
4. Test all the methods using the debugger to step through and help understand which methods are called and why.
5. Try to think of another method that could be added to Shape that would need to be overridden in its child classes. Add this method and the overridden versions, and test that it works. If you have trouble thinking of something for the purposes of this exercise it doesn't have to be a useful method.

#### Exercise 02

1. Create a new project and save it. Add the **.cpp** and **.h** files from Tutorial 08 Exercise 01 as done in the previous exercise.
2. Add the code from the tutorial that initially demonstrates polymorphism. Test your understanding as usual using the debugger.
3. Add the virtual keyword to the Shape DisplayInfo() class declaration, and again step through the code.
4. Create destructors for all your classes and add the virtual keyword to the Shape destructors in the class declaration. Add code to delete all types of objects via their own type pointers and via Shape pointers Test as usual.