# Lesson 10 - Inheritance
# Extending Class Functionality

## Summary

In this lesson we explain the concept of Inheritance when applied to C++ classes.

### New Concepts

Inheritance, method overriding, polymorphism.

## Inheritance

In the previous lesson we introduced the notion of classes and defined an "Enemy" class. Now say we wish to declare a new class called an `ArmedEnemy`. Rather than declare a new class from scratch, we can create our `ArmedEnemy` class by 'extending' our original Enemy class. Extending classes brings the possibility of having new classes which 'inherit' methods and fields from their parent classes. For example, we may want our `ArmedEnemy` class to have the same methods as an Enemy class but with additional features which give it extra capabilities. Rather than re-declaring methods in the `ArmedEnemy` class, a repetitious and error-prone task, we can simply inherit them from our Enemy class.

First we will make some modifications to our original Enemy class. First notice on lines 4 to 8 the introduction of the `virtual` keyword. We need this keyword to allow method-overriding, as we explain this feature shortly, we'll move on for now. Secondly, notice the use of the `protected` keyword on line 9. This is another access-modifier and replaces the `private` keyword that was previously here.

The `protected` access modifier states that any sub-class that extends this class also has access to these fields and methods. As we want our `ArmedEnemy` class to have access to these fields, we need to change `private` to `protected`.

```
1  class Enemy {
2  public:
3     Enemy(int hps);
4     virtual ~Enemy();
5     virtual int get_hit_points() const;
6     virtual int get_score() const;
7     virtual void set_hit_points(int new_hit_points);
8     virtual  void set_score(int new_score);
9  protected:
10    int hit_points;
11    int* score;
12 };
```

inheritance.cpp

Now let's take a look at our `ArmedEnemy` class. See line 13, and notice that we declare the class as usual but we follow the name by a colon (`:`) followed by `public Enemy`, the name of this class' super-class. This syntax stipulates that the `ArmedEnemy` class inherits (the `public` and `protected`) functionality of the `Enemy` class. Notice that we provide a Constructor and Destructor as usual and two more methods: `set_score` and `shoot`. We also declare an integer field `ammo_level`.

```
13 class ArmedEnemy : public Enemy
14 {
15 public:
16    ArmedEnemy(int hps, int ammo);
17    virtual ~ArmedEnemy();
18    virtual void set_score(const int new_score);
19    virtual void shoot();
20 protected:
```

```
21    int ammo_level;
22 };
```

inheritance.cpp

Now let's examine the new method definitions. We start with the constructor on line 23. Notice the initialiser list defined on lines 24 and 25. First we call the super-class constructor passing it the 'hps' parameter on line 24: `Enemy(hps)`. This action will initialise the inherited aspects of this class for us by calling the `Enemy` class constructor with the appropriate parameter. Calling the `Enemy` constructor will also allocate the memory we used for the `score` field. Next we initialise the new `ammo_level` field from the second `ammo` parameter.

The destructor is defined as normal although no actions are necessary. When the `ArmedEnemy` Class is deleted its destructor will be called, followed automatically by the Destructor of its super-class, releasing the memory allocated when this class was constructed.

```
23 ArmedEnemy::ArmedEnemy(int hps, int ammo):
24 Enemy(hps),
25 ammo_level(ammo) {
26 }
27
28 ArmedEnemy::~ArmedEnemy() {
29 }
```

inheritance.cpp

When defining a sub-class, we can 'override' methods in our new class that already exist in their super-class. For instance, as `ArmedEnemy` inherits from `Enemy`, we can call the `set_score` method in our `ArmedEnemy` class and the corresponding `set_score` method in the `Enemy` class will be invoked. However, we might want the `set_score` method to behave differently when called by an `ArmedEnemy` class, hence we say we want to override the `set_score` method to reflect this.

To override the `set_score` method we simply re-declare and re-define this method, making sure we provide the same name, parameters and return type as in the super-class. When we are dealing with pointers or references to classes, we also need to ensure the method in the super-class has been prefixed with the keyword `virtual`. Note on line 31 and 32 we re-define the `set_score` method adding an extra statement that outputs the score to screen using `cout`.

```
30 void ArmedEnemy::set_score(const int new_score) {
31    *score = new_score;
32    cout << "score is now " << *score << "\n";
33 }
```

inheritance.cpp

Finally, for our `ArmedEnemy` class we define the `shoot` method (see lines 34-41).

```
34 void ArmedEnemy::shoot() {
35    if(ammo_level > 0) {
36       cout << "bang!\n";
37       --ammo_level;
38    } else {
39       cout << "out of ammo\n";
40    }
41 }
```

inheritance.cpp

Now we use the `main` function to demonstrate inheritance in our new `ArmedEnemy` class. Note on line 47 we create a new `ArmedEnemy` object on the heap, referred to by the pointer `ae`. On line 48 we call the `set_hit_points` method on our `ArmedEnemy` object. As we inherited this method from the super-class `Enemy`, this statement executes as normal. We display this fact on the following line by calling the `get_hit_points` method and displaying the result to screen.

Now we introduce another aspect of Inheritance, the power of Polymorphism. Note on line 53 we call `some_function` and pass the `ArmedEnemy` Object as the parameter. However, if you examine the function declaration on line 42, you'll see that it expects a reference to an object of class `Enemy`. This is one of the major benefits of inheritance; although `some_function` expects a reference to an `Enemy` object we can pass an `ArmedEnemy` reference because `ArmedEnemy` inherits from `Enemy`. We can say that `ArmedEnemy` has a "is-a" relationship with `Enemy` and in the `some_function` function, we can call the `set_score` method because `ArmedEnemy` inherits this method from `Enemy`.

Remember that we had previously overridden the `set_score` method when the `ArmedEnemy` method was defined. The great thing about Polymorphism is that the overridden method in `ArmedEnemy` will be called on line 43, not the original method in `Enemy`. Polymorphism provides a versatile method of exploiting a range of functionality in sub-classes, without the calling methods (like `some_function`) having to know the specifics of those sub-classes. The `some_function` function only needs to know about the `Enemy` class, Polymorphism handles the rest at run-time. You must ensure you mark a method as `virtual` however if you want to use this behaviour, otherwise the program won't be able to call the overridden method.

```
42 void some_function(Enemy& enemy) {
43    enemy.set_score(6);
44 }
45
46 int main(void) {
47    ArmedEnemy* ae = new ArmedEnemy(2, 5);
48    ae->set_hit_points(3);
49    cout << "hit points = " << ae->get_hit_points() << "\n";
50
51    ae->shoot();
52
53    some_function(*ae);
54
55    delete ae;
56    ae = NULL;
57    return 0;
58 }
```

inheritance.cpp

**Under the Hood**

Whenever you declare a `virtual` method something special takes place in the run-time environment of your C++ program. We have seen how `virtual` methods provide polymorphism so that it is possible to determine at run-time which particular method should be invoked depending on the class or sub-class type of the calling object. In order to support this functionality, a `virtual` table (or `vtable`) is created for every class which either has a `virtual` method or inherits from a class with a `virtual` method. The `vtable` is then used to locate the correct implementation of a particular method depending on the sub-class type.
Bugs can arise from forgetting to mark a method as `virtual` because a non `virtual` method does not cause the `vtable` to be consulted. Some books on C++, therefore, advise programmers to always mark a method as `virtual` if there exists the chance that the owning class will be subclassed. As game programmers, however, you have to keep in mind that liberal use of `virtual` may incur a performance penalty caused by consulting the `vtable` (a `virtual` method may incur a costly data cache miss when consulting the `vtable`). The `virtual` functionality is a apt example of why it often makes sense to write our software in stages. In the early stages, don't refrain from making use of `virtual` as you identify the overall class design for your game/engine. Then, once most of your classes are written, revisit them individually to identify portions of code where the use of `virtual` methods can be removed. Optimisations are best reserved until late in a software's life (or as Donald Knuth said *"premature optimisation is the root of all evil!"*).

## New Feature

C++11 provides two new features to make the use of inheritence less error-prone, namely the `override` and `final` keywords. One common mistake occurs when a programmer wishes to override the functionality of a method from a base-class but accidentally writes the overriding method with a different method signature, as can be seen in on lines 61 and 66 with the `update` method (note the `float` and `int` arguments to the `update` methods).

```
59 class BaseClass {
60 protected:
61    virtual void update(float value);
62 };
63
64 class DerivedClass : BaseClass {
65 protected:
```

```
    virtual void update(int value); // unintentionally created another 'update' method
};
```

inheritance.cpp

To make your intentions more explicit, you can use the `override` modifier (see line 76). The compiler will see that you want to override a method in the base class but will not find the 'update' method because of the different signature. The compiler will alert you to the mistake rather than allowing you to overlook the error.

```
68 class BaseClass {
69 protected:
70     virtual void update(float value);
71     virtual void draw(float value) final;
72 };
73
74 class DerivedClass : BaseClass {
75 protected:
76     virtual void update(int value) override; // compiler will spot the mistake in the new signature
77     void draw(float value); // cannot override 'draw' because it is final
78 };
```

inheritance.cpp

You may also mark a method as `final` in `BaseClass` to signal to the compiler that a method cannot be overridden. The `draw` method on line 71, for instance, has been marked `final`. The attempt to override `draw` on line 77 will now prompt an alert to the compiler that such behaviour has been disallowed.

## Exercises

1. If we did not make the `Enemy` Class Destructor `virtual`, how might a memory leak be introduced into our program when the `ArmedEnemy` class Destructor is called?

2. How does giving a class one or more virtual methods, affect the memory requirements for the C++ compiler compared to a class which possesses no virtual methods?

3. Create a `Boss` class which inherits from the `Armed Enemy` class. Implement an additional armour level field for the `Boss` class and provide suitable getter and setter methods. Create a `Boss` Object in the main function to test the functionality of your `Boss` class.

4. By taking advantage of Inheritance and Polymorphism we can group together objects of different classes and perform common operations on them. Make the `Enemy` class abstract. Create an array of 10 `ArmedEnemy` objects and a `Boss` object. With the help of Polymorphism, create a single array of pointers to the `ArmedEnemy` and `Boss` objects. Iterate through the array decrementing the `hit_point` value of each pointer; use a single function for decrementing which accepts a `Enemy` as its argument.

5. Create an abstract `Comparable` class. Any class which inherits from `Comparable` should implement a `compare_to` method which has the following header:

```
/** returns 1 if this class is greater than rhs, 0 if equal
* and (-1) if this class is less than rhs.
*/
int compare_to(const Comparable& rhs);
```

inheritance.cpp

Create a new Binary Search Tree class which stores `Comparable` objects rather than integers. Create some `Comparable` objects and test your new Binary Search Tree class.