11.6 — Adding new functionality to a derived class

BY ALEX ON JANUARY 17TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the **introduction to inheritance** lesson, we mentioned that one of the biggest benefits of using derived classes is the ability to reuse already written code. You can inherit the base class functionality and then add new functionality, modify existing functionality, or hide functionality you don't want. In this and the next few lessons, we'll take a closer look at how each of these things is done.

First, let's start with a simple base class:

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Base\n"; }
};
```

Now, let's create a derived class that inherits from Base. Because we want the derived class to be able to set the value of m_value when derived objects are instantiated, we'll make the Derived constructor call the Base constructor in the initialization list.

```cpp
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }
};
```

Adding new functionality to a derived class

In the above example, because we have access to the source code of the Base class, we can add functionality directly to Base if we desire.

There may be times when we have access to a base class but do not want to modify it. Consider the case where you have just purchased a library of code from a 3rd party vendor, but need some extra functionality. You could add to the original code, but this isn't the best solution. What if the vendor sends you an update? Either your additions will be overwritten, or you'll have to manually migrate them into the update, which is time-consuming and risky.

Alternatively, there may be times when it's not even possible to modify the base class. Consider the code in the standard library. We aren't able to modify the code that's part of the standard library. But we are able to inherit from those classes, and then add our own functionality into our derived classes. The same goes for 3rd party libraries where you are provided with headers but the code comes precompiled.

In either case, the best answer is to derive your own class, and add the functionality you want to the derived class.

One obvious omission from the Base class is a way for the public to access m_value. We could remedy this by adding an access function in the Base class -- but for the sake of example we're going to add it to the derived class instead. Because m_value has been declared as protected in the Base class, Derived has direct access to it.

To add new functionality to a derived class, simply declare that functionality in the derived class like normal:

```
class Derived: public Base
{
public:
    Derived(int value)
        :Base(value)
    {
    }

    int getValue() { return m_value; }
};
```

Now the public will be able to call getValue() on an object of type Derived to access the value of m_value.

```
int main()
{
    Derived derived(5);
    std::cout << "derived has value " << derived.getValue() << '\n';

    return 0;
}
```

This produces the result:

```
derived has value 5
```

Although it may be obvious, objects of type Base have no access to the getValue() function in Derived. The following does not work:

```
int main()
{
    Base base(5);
    std::cout << "base has value " << base.getValue() << '\n';

    return 0;
}
```

This is because there is no getValue() function in Base. Function getValue() belongs to Derived. Because Derived is a Base, Derived has access to stuff in Base. However, Base does not have access to anything in Derived.

11.6a -- Calling inherited functions and overriding behavior

Index

11.5 -- Inheritance and access specifiers