11.6 — Adding, changing, and hiding members in a derived class

BY ALEX ON JANUARY 17TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2016

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 lesson, we'll take a closer look at how this is done.

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

```
#include <iostream>
using namespace std;

class Base
{
protected:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

    void Identify() { cout << "I am a Base" << endl; }
};
```

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

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

We'll develop Derived over the course of this lesson.

Adding new functionality

Because we have access to the source code of the Base class, we could add functionality directly to Base. However, there may be times when we do not want to, or can not. 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. It's also common for developers to release header files containing class definitions, but release the implementation code precompiled -- this means you can use the code, but you won't have the ability to modify it directly.

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_nValue. Normally we'd write 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_nValue 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 nValue)
        :Base(nValue)
    {
    }

    int GetValue() { return m_nValue; }
};
```

Now the public will be able to call GetValue() in order to access the value of m_nValue.

```
int main()
{
    Derived cDerived(5);
    cout << "cDerived has value " << cDerived.GetValue() << endl;

    return 0;
}
```

This produces the result:

```
cDerived 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 cBase(5);
    cout << "cBase has value " << cBase.GetValue() << endl;

    return 0;
}
```

This is because there is no GetValue() function in Base. 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.

Redefining functionality

When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the inherited classes. It uses the first one it finds.

Consequently, take a look at the following example:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

This prints

```
I am a Base
I am a Base
```

When cDerived.Identify() is called, the compiler looks to see if Identify() has been defined in the Derived class. It hasn't. Then it starts looking in the inherited classes (which in this case is Base). Base has defined a Identify() function, so it uses that one. In other words, Base::Identify() was used because Derived::Identify() doesn't exist.

However, if we had defined Derived::Identify() in the Derived class, it would have been used instead. This means that we can make functions work differently with our derived classes by redefining them in the derived class!

In our above example, it would be more accurate if `cDerived.Identify()` printed "I am a Derived". Let's modify Identify() so it returns the correct response when we call Identify() with a Derived object.

To modify a function the way a function defined in a base class works in the derived class, simply redefine the function in the derived class.

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

    int GetValue() { return m_nValue; }

    // Here's our modified function
    void Identify() { cout << "I am a Derived" << endl; }
};
```

Here's the same example as above, using the new Derived::Identify() function:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

```
I am a Base
I am a Derived
```

Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can be redefined as public in the derived class, or vice-versa!

Adding to existing functionality

Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it. In the above example, note that Derived::Identify() completely hides Base::Identify()! This may not be what we want. It is possible to have our Derived function call the Base function of the same name (in order to reuse code) and then add additional functionality to it.

To have a derived function call a base function of the same name, simply do a normal function call, but prefix the function with the scope qualifier (the name of the base class and two colons). The following example redefines Derived::Identify() so it first calls Base::Identify() and then does it's own additional stuff.

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

    int GetValue() { return m_nValue; }

    void Identify()
    {
        Base::Identify(); // call Base::Identify() first
        cout << "I am a Derived"; // then identify ourselves
    }
};
```

Now consider the following example:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

```
I am a Base
I am a Base
I am a Derived
```

When `cDerived.Identify()` is executed, it resolves to Derived::Identify(). However, the first thing Derived::Identify() does is call Base::Identify(), which prints “I am a Base”. When Base::Identify() returns, Derived::Identify() continues executing and prints “I am a Derived”.

This is all pretty straightforward. The real lesson to take away from this is that if you want to call a function in a base class that has been redefined in the derived class, you need to use the scope resolution operator (::) to explicitly say which version of the function you want.

If we had defined Derived::Identify() like this:

```
    void Identify()
    {
        Identify(); // Note: no scope resolution!
        cout << "I am a Derived"; // then identify ourselves
    }
```

Identify() without a scope resolution qualifier would default to the Identify() in the current class, which would be Derived::Identify(). This would cause Derived::Identify() to call itself, which would lead to an infinite loop!

Hiding functionality

In C++, it is not possible to remove functionality from a class. However, it is possible to hide existing functionality.

As mentioned above, if you redefine a function, it uses whatever access specifier it’s declared under in the derived class. Therefore, we could redefine a public function as private in our derived class, and the public would lose access to it. However, C++ also gives us the ability to change a base member’s access specifier in the derived class without even redefining the member! This is done by simply naming the member (using the scope resolution operator) to have it’s access changed in the derived class under the new access specifier.

For example, consider the following Base:

```
class Base
{
private:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

protected:
    void PrintValue() { cout << m_nValue; }
};
```

Because Base::PrintValue() has been declared as protected, it can only be called by Base or it’s derived classes. The public can not access it.

Let’s define a Derived class that changes the access specifier of PrintValue() to public:

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

    // Base::PrintValue was inherited as protected, so the public has no access
    // But we're changing it to public by declaring it in the public section
    Base::PrintValue;
};
```

This means that this code will now work:

```
int main()
{
    Derived cDerived(7);

    // PrintValue is public in Derived, so this is okay
    cDerived.PrintValue(); // prints 7
    return 0;
}
```

Note that Base::PrintValue does not have the function call operator (()) attached to it.

We can also use this to make public members private:

```
class Base
{
public:
    int m_nValue;
};

class Derived: public Base
{
private:
    Base::m_nValue;

public:
    Derived(int nValue)
    {
         m_nValue = nValue;
    }
};

int main()
{
    Derived cDerived(7);

    // The following won't work because m_nValue has been redefined as private
    cout << cDerived.m_nValue;

    return 0;
}
```

Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class. (Alternatively, instead of inheriting Base's members publicly and making m_nValue private by overriding it's access specifier, we could have inherited Base privately, which would have caused all of Base's member to be inherited privately in the first place).

One word of caution: you can only change the access specifiers of base members the class would normally be able to access. Therefore, you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

11.7 -- Multiple inheritance

Index

11.5 -- Inheritance and access specifiers

Share this:

39 comments to 11.6 — Adding, changing, and hiding members in a derived class

Zafer
January 31, 2008 at 3:03 pm · Reply

I think the following statement about the last example is confusing: "Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class (alternatively, we could have