

11.6a — Calling inherited functions and overriding behavior

BY ALEX ON JUNE 27TH, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

By default, derived classes inherit all of the behaviors defined in a base class. In this lesson, we'll examine in more detail how member functions are selected, as well as how we can leverage this to change behaviors in a derived class.

Calling a base class function

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 parent classes. It uses the first one it finds.

Consequently, take a look at the following example:

```
1  class Base
2  {
3  protected:
4      int m_value;
5
6  public:
7      Base(int value)
8          : m_value(value)
9      {
10     }
11
12     void identify() { std::cout << "I am a Base\n"; }
13 };
14
15 class Derived: public Base
16 {
17 public:
18     Derived(int value)
19         : Base(value)
20     {
21     }
22 };
23 int main()
24 {
25     Base base(5);
26     base.identify();
27
28     Derived derived(7);
29     derived.identify();
30
31     return 0;
32 }
```

This prints

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

When `derived.identify()` is called, the compiler looks to see if function `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 an

identify() function, so it uses that one. In other words, Base::identify() was used because Derived::identify() doesn't exist.

This means that if the behavior provided by a base class is sufficient, we can simply use the base class behavior.

Redefining behaviors

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 derived.identify() printed "I am a Derived". Let's modify function identify() in the Derived class so it returns the correct response when we call function identify() with a Derived object.

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

```

1  class Derived: public Base
2  {
3  public:
4      Derived(int value)
5          : Base(value)
6      {
7      }
8
9      int getValue() { return m_value; }
10
11     // Here's our modified function
12     void identify() { std::cout << "I am a Derived\n"; }
13 };

```

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

```

1  int main()
2  {
3      Base base(5);
4      base.identify();
5
6      Derived derived(7);
7      derived.identify();
8
9      return 0;
10 }

```

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!

```

1  class Base
2  {
3  private:
4      void print()
5      {
6          std::cout << "Base";
7      }

```

```
8   };
9
10  class Derived : public Base
11  {
12  public:
13      void print()
14      {
15          std::cout << "Derived ";
16      }
17  };
18
19
20
21  int main()
22  {
23      Derived derived;
24      derived.print(); // calls derived::print(), which is public
25      return 0;
26  }
```

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 version of the 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 its own additional stuff.

```
1   class Derived: public Base
2   {
3   public:
4       Derived(int value)
5           : Base(value)
6       {
7       }
8
9       int GetValue() { return m_value; }
10
11      void identify()
12      {
13          Base::identify(); // call Base::identify() first
14          std::cout << "I am a Derived\n"; // then identify ourselves
15      }
16  };
```

Now consider the following example:

```
1   int main()
2   {
3       Base base(5);
4       base.identify();
5
6       Derived derived(7);
7       derived.identify();
8
9       return 0;
10  }
```

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

When `derived.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 should be pretty straightforward. Why do we need to use the scope resolution operator (`::`)? If we had defined `Derived::identify()` like this:

```
1  class Derived: public Base
2  {
3  public:
4      Derived(int value)
5          : Base(value)
6      {
7      }
8
9      int GetValue() { return m_value; }
10
11     void identify()
12     {
13         identify(); // Note: no scope resolution!
14         cout << "I am a Derived";
15     }
16 };
```

Calling function `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!

There’s one bit of trickiness that we can run into when trying to call friend functions in base classes, such as `operator<<`. Because friend functions of the base class aren’t actually part of the base class, using the scope resolution qualifier won’t work. Instead, we need a way to make our `Derived` class temporarily look like the `Base` class so that the right version of the function can be called.

Fortunately, that’s easy to do, using `static_cast`. Here’s an example:

```
1  class Base
2  {
3  private:
4      int m_value;
5
6  public:
7      Base(int value)
8          : m_value(value)
9      {
10     }
11
12     friend std::ostream& operator<< (std::ostream &out, const Base &b)
13     {
14         out << "In Base\n";
15         out << b.m_value << '\n';
16         return out;
17     }
18 };
19
20 class Derived : public Base
21 {
```

```
22 public:
23     Derived(int value)
24         : Base(value)
25     {
26     }
27
28     friend std::ostream& operator<< (std::ostream &out, const Derived &d)
29     {
30         out << "In Derived\n";
31         // static_cast Derived to a Base object, so we call the right version of operator<<
32         out << static_cast<Base>(d);
33         return out;
34     }
35 };
36
37 int main()
38 {
39     Derived derived(7);
40
41     std::cout << derived;
42
43     return 0;
44 }
```

Because a Derived is-a Base, we can static_cast our Derived object into a Base, so that the appropriate version of operator<< that uses a Base is called.

This prints:

```
In derived
In base
7
```



11.6b -- Hiding inherited functionality



Index



11.6 -- Adding new functionality to a derived class

 [C++ TUTORIAL](#) |  [PRINT THIS POST](#)

26 comments to 11.6a — Calling inherited functions and overriding behavior



hellmet

November 23, 2019 at 10:17 am · Reply

How do I make sure I'm not accidentally masking a method inside the parent of a class? Is making sure I don't the only way?