24.7 — Calling inherited functions and overriding behavior

learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/

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.

In other words, it uses the most-derived version of the function that it can find.

Consequently, take a look at the following example:

```
#include <iostream>

class Base
{
protected:
    int m_value {};

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

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

class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }
};
int main()
{
    Base base { 5 };
    base.identify();

    Derived derived { 7 };
    derived.identify();

    return 0;
}
```

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.

```
#include <iostream>

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

    int getValue() const { return m_value; }

    // Here's our modified function
    void identify() const { std::cout << "I am a Derived\n"; }
};
```

Here’s the same example as above, using the new Derived::identify() function:

```
int main()
{
    Base base { 5 };
    base.identify();

    Derived derived { 7 };
    derived.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!

```
#include <iostream>

class Base
{
private:
        void print() const
        {
                std::cout << "Base";
        }
};

class Derived : public Base
{
public:
        void print() const
        {
                std::cout << "Derived ";
        }
};


int main()
{
        Derived derived;
        derived.print(); // calls derived::print(), which is public
        return 0;
}
```

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.

```
#include <iostream>

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

    int getValue() const  { return m_value; }

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

Now consider the following example:

```
int main()
{
    Base base { 5 };
    base.identify();

    Derived derived { 7 };
    derived.identify();

    return 0;
}
```

```
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:

```
#include <iostream>

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

    int getValue() const { return m_value; }

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

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 recursion!

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:

```
#include <iostream>

class Base
{
private:
	int m_value {};

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

	friend std::ostream& operator<< (std::ostream& out, const Base& b)
	{
		out << "In Base\n";
		out << b.m_value << '\n';
		return out;
	}
};

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

	friend std::ostream& operator<< (std::ostream& out, const Derived& d)
	{
		out << "In Derived\n";
		// static_cast Derived to a Base object, so we call the right version
of operator<<
		out << static_cast<const Base&>(d);
		return out;
	}
};

int main()
{
	Derived derived { 7 };

	std::cout << derived << '\n';

	return 0;
}
```

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
```