24.7 — Calling inherited functions and overriding behavior

ALEX[1] JUNE 9, 2024

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.

When a member function is called on a derived class object, the compiler first looks to see if any function with that name exists in the derived class. If so, all overloaded functions with that name are considered, and the function overload resolution process is used to determine whether there is a best match. If not, the compiler walks up the inheritance chain, checking each parent class in turn in the same way.

Put another way, the compiler will select the best matching function from the most-derived class with at least one function with that name.

Calling a base class function

First, let's explore what happens when the derived class has no matching function, but the base class does:

```
#include <iostream>

class Base
{
public:
    Base() { }

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

class Derived: public Base
{
public:
    Derived() { }
};

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

    Derived derived {};
    derived.identify();

    return 0;
}
```

This prints:

```
Base::identify()
Base::identify()
```

When `base.identify()` is called, the compiler looks to see if a function named `identify()` has been defined in class `Base`. It has, so the compiler looks to see if it is a match. It is, so it is called

When `derived.identify()` is called, the compiler looks to see if a function named `identify()` has been defined in the `Derived` class. It hasn't. So it moves to the parent class (in this case, `Base`), and tries again there. `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!

For example, let's say we want `derived.identify()` to print `Derived::identify()`. We can simply add 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 Base
{
public:
    Base() { }

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

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

    void identify() const { std::cout << "Derived::identify()\n"; }
};

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

    Derived derived {};
    derived.identify();

    return 0;
}
```

This prints:

```
Base::identify()
Derived::identify()
```

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!

```cpp
#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 when called with a derived object. 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 of the base class. For example:

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

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

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

    void identify() const
    {
        std::cout << "Derived::identify()\n";
        Base::identify(); // note call to Base::identify() here
    }
};

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

    Derived derived {};
    derived.identify();

    return 0;
}
```

This prints:

```
Base::identify()
Derived::identify()
Base::identify()
```

When `derived.identify()` is executed, it resolves to `Derived::identify()`. After printing `Derived::identify()`, it then calls `Base::identify()`, which prints `Base::identify()`.

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

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

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

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

    void identify() const
    {
        std::cout << "Derived::identify()\n";
        identify(); // no scope resolution results in self-call and infinite recursion
    }
};

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

    Derived derived {};
    derived.identify();

    return 0;
}
```

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:

```cpp
#include <iostream>

class Base
{
public:
    Base() { }

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

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

    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 {};

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

    return 0;
}
```

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

This prints:

```
In Derived
In Base
```

Overload resolution in derived classes

As noted at the top of the lesson, the compiler will select the best matching function from the most-derived class with at least one function with that name.

First, let's take a look at a simple case where we have overloaded member functions:

```cpp
#include <iostream>

class Base
{
public:
    void print(int)    { std::cout << "Base::print(int)\n"; }
    void print(double) { std::cout << "Base::print(double)\n"; }
};

class Derived: public Base
{
public:
};


int main()
{
    Derived d{};
    d.print(5); // calls Base::print(int)

    return 0;
}
```

For the call `d.print(5)`, the compiler doesn't find a function named `print()` in `Derived`, so it checks `Base` where it finds two functions with that name. It uses the function overload resolution process to determine that `Base::print(int)` is a better match than `Base::print(double)`. Therefore, `Base::print(int)` gets called, just like we'd expect.

Now let's look at a case that doesn't behave like we might expect:

```cpp
#include <iostream>

class Base
{
public:
    void print(int)    { std::cout << "Base::print(int)\n"; }
    void print(double) { std::cout << "Base::print(double)\n"; }
};

class Derived: public Base
{
public:
    void print(double) { std::cout << "Derived::print(double)"; } // this function added
};


int main()
{
    Derived d{};
    d.print(5); // calls Derived::print(double), not Base::print(int)

    return 0;
}
```

For the call `d.print(5)`, the compiler finds one function named `print()` in `Derived`, therefore it will only consider functions in `Derived` when trying to determine what function to resolve to. This function is also the best matching function in `Derived` for this function call. Therefore, this calls `Derived::print(double)`.

Since `Base::print(int)` has a parameter that is a better match for int argument `5` than `Derived::print(double)`, you may have been expecting this function call to resolve to `Base::print(int)`. But because `d` is a `Derived`, there is at least one `print()` function in `Derived`, and `Derived` is more derived than `Base`, the functions in `Base` are not even considered.

So what if we actually want `d.print(5)` to resolve to `Base::print(int)`? One not-great way is to define a `Derived::print(int)`:

```cpp
#include <iostream>

class Base
{
public:
    void print(int)    { std::cout << "Base::print(int)\n"; }
    void print(double) { std::cout << "Base::print(double)\n"; }
};

class Derived: public Base
{
public:
    void print(int n) { Base::print(n); } // works but not great, as we have to define
    void print(double) { std::cout << "Derived::print(double)"; }
};

int main()
{
    Derived d{};
    d.print(5); // calls Derived::print(int), which calls Base::print(int)

    return 0;
}
```

While this works, it's not great, as we have to add a function to `Derived` for every overload we want to fall through to `Base`. That could be a lot of extra functions that essentially just route calls to `Base`.

A better option is to use a using-declaration in `Derived` to make all `Base` functions with a certain name visible from within `Derived`:

```cpp
#include <iostream>

class Base
{
public:
    void print(int)    { std::cout << "Base::print(int)\n"; }
    void print(double) { std::cout << "Base::print(double)\n"; }
};

class Derived: public Base
{
public:
    using Base::print; // make all Base::print() functions eligible for overload
resolution
    void print(double) { std::cout << "Derived::print(double)"; }
};


int main()
{
    Derived d{};
    d.print(5); // calls Base::print(int), which is the best matching function visible
in Derived

    return 0;
}
```

By putting the using-declaration `using Base::print;` inside `Derived`, we are telling the compiler that all `Base` functions named `print` should be visible in `Derived`, which will cause them to be eligible for overload resolution. As a result, `Base::print(int)` is selected over `Derived::print(double)`.

[Next lesson](#)
24.8 [Hiding inherited functionality](#)

2

[Back to table of contents](#)

3

[Previous lesson](#)
24.6 [Adding new functionality to a derived class](#)

4

5

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[7] are connected to your provided email address.

89 COMMENTS

Newest

James

May 9, 2025 4:56 am PDT

This is probably not very relevant in practice, but interesting nonetheless:

A function in the derived class will be preferred by the compiler even if it's private and a matching overload in the base class is public, leading to a potential compiler error. On a similar note, the compiler will choose a better-matching private overload over a worse public overload in the same class and refuse to compile.

0 Reply

Nidhi Gupta

May 6, 2025 9:07 am PDT

If the derived class doesn't define a function, the base version is used.

If the derived class defines the function, it hides all base versions with the same name—even if the overloads differ.

2. Function Hiding

Even if the overloads in the base class are better matches, once a derived class defines a function of the same name, the base versions are hidden unless explicitly reintroduced.

3. Scope Resolution

Use Base::function() to call the base version inside a derived override to extend rather than replace behavior.

0 Reply

Cal

August 8, 2024 2:56 pm PDT

When we use a using-declaration in `Derived` to make all `Base` functions with a certain name visible within `Derived`, are there any consequences to having multiple definitions of similar function overloads?

Referring to `void print(double)` in the final code of the lesson where we made all `Base::print()` functions eligible for overload resolution, will the compiler remove all parent versions of the overload with the same signature so that someone using the Derived public interface that calls `print(double)` will only resolve to the version that belongs to the `Derived` class? In other words, function overloads of the same signature from the parent class will be lost to (only to) the user calling the `Derived` public interface?

Edit: I guess not, I realised `Derived.Base::print(double)` still works so the compiler did not remove it. Anyhow are there side effects we should be wary of?

Last edited 10 months ago by Cal

1 Reply

Alex Author

Reply to Cal [8] August 10, 2024 5:01 pm PDT

Only that it may result in a better matching overload being pulled in.

```
#include <iostream>

class Base
{
public:
    void foo(char) { std::cout << "foo(char)\n"; }
    void foo(int) { std::cout << "foo(int)\n"; }
};

class Derived: public Base
{
public:
//    using Base::foo;
    void foo(double) { std::cout << "foo(double)\n"; }
};


int main()
{
    Derived d{};
    d.foo(5); // consider this line

    return 0;
}
```

With `using Base::foo` commented out, `d.foo(5)` calls `Derived::foo()`. If we uncomment `using Base::foo` because we want `Base::foo(char)` to be accessible, `d.foo(5)` now calls `Base::foo(int)` as it is a better match.

1 Reply

Seb

Reply to Alex [9] January 17, 2025 7:55 am PST

When there is the same overload for a function (like in the example in the lesson, where there are two print(double) functions), how do we know which one gets called? Does the Derived function get priority over the Base?

2 Reply

Block3r

Reply to Seb [10] May 3, 2025 9:22 am PDT

Methods from derived classes shadow the methods from the superclasses, as it was stated in the article - redefinition of the function in derived class is superior when called from derived class. Consider following:

```
#include <iostream>

class Base
{
public:
    void foo(double) { std::cout << "Base foo(double)\n"; }
};

class Derived: public Base
{
public:
    using Base::foo;
    void foo(double) { std::cout << "Derived foo(double)\n"; }
};


int main()
{
    Derived d{};
    d.foo(5.0);

    return 0;
}
```

The output is of course `Derived foo(double)` because if **derived class reimplements a method of base class with the same signature**, this method will shadow the method from base class, even if it was made visible via using.

Last edited 1 month ago by Block3r

0 Reply

Sanderson

Reply to Seb [10] April 18, 2025 1:30 am PDT

I have the same question. I asked AI and it told me that the function in the Derived class will override the ones with same signature in the Base class. Only all the functions in the Derived and those in the Base which are not hided will participate in the resolution of overloading.

Hope Alex will give us an answer.

0 Reply

Karl

June 7, 2024 11:01 pm PDT

"So what if we actually want d.print(5) to resolve to Base::print(int)? One not-great way is to define a Derived::print(double):"

I think you meant "... is to define a `Derived::print(int)`" as `Derived::print(double)` is already defined.

This is definitely superior to my work around. Thanks!!

1 Reply

Alex Author

Reply to Karl 11 June 9, 2024 8:05 pm PDT

Thanks for the correction! Integrated.

1 Reply

yousef elsayed

June 23, 2023 5:39 pm PDT

Very helpful thank you for your efforts

4 Reply

shaan

March 16, 2023 10:54 pm PDT

why do i have to declare pri_base as a friend in both classes in order to access m_base

```cpp
#include <cassert> // for assert()
#include <initializer_list> // for std::initializer_list
#include <iostream>


class base
{

    int m_base{10};

    public:
    friend void pri_base(base& obj);

};


class derived:public base
{
    int  m_der{20};
    public:
 friend void  pri_base(base& obj)
   {
    std::cout<<obj.m_base;
   }

};



int main()
{
    derived d;
    pri_base(d);
}
```

0 Reply

Alex Author

Reply to shaan [12] March 18, 2023 1:32 pm PDT

The first friend declaration gives pri_base access to the private members of base. The second one implements the function as a non-member (friend functions aren't members) that has access to the private members of derived.

Since pri_base doesn't need access to the private members of derived, it shouldn't be defined there. Either define it in base, or make it a non-member.

1 Reply

Jannik Becker

February 14, 2023 12:30 pm PST

Searched the web how to call the base classes operator<< for 10 minutes just to see it appearing in the next scroll down :(

3 Reply

Polo

Reply to Jannik Becker [13] October 16, 2024 10:54 pm PDT

pat pat

1 Reply

Tcorn

February 7, 2023 5:21 am PST

I have a question .When we call Base::identify() , what is the argument for the hidden "this" pointer ????

Last edited 2 years ago by Tcorn

0 Reply

Tcorn

Reply to Tcorn [14] February 8, 2023 5:45 am PST

Ohh sorry Alex , I realized it now , the "this" pointer of the caller ("identify") is the argument .

0 Reply

Alex Author

Reply to Tcorn [15] February 8, 2023 7:53 am PST

Yes, the value of "this" is the address of the object the member function is being called on.

0 Reply

GMJ

November 17, 2022 1:37 pm PST

There's a comment above:

"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!"

This isn't an infinite loop, but an infinite recursion, and would likely cause the program to crash from blowing the stack. Since recursion was covered in an earlier section, this can probably be clarified.

Thank you for all of the effort you put into this site -- this has been a gold mine for me.

6 Reply

Alex Author

Reply to GMJ [16] November 21, 2022 3:31 pm PST

I agree, recursion is a better term. Updated. Thanks!

6 Reply

KotoWhiskas

July 15, 2022 2:16 am PDT

Sorry if it's a stupid question but how static_cast does convert from Derived to Base? Is it just reference to a fenced Derived object which doesn't show non-Base members and is seen as Base?

0 Reply

Alex Author

Reply to KotoWhiskas [17] July 18, 2022 11:28 am PDT

A Derived object static_cast to a Base& uses the Base class interface (but is actually a Derived object, so virtual functions will still resolve to Derived functions). This means Derived portions of the class are inaccessible except through virtual function resolution.

4 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/adding-new-functionality-to-a-derived-class/
5. https://www.learncpp.com/calling-inherited-functions-and-overriding-behavior/
6. https://www.learncpp.com/cpp-tutorial/introduction-to-c14/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-600650
9. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-600726
10. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-606666
11. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-598116
12. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-578466
13. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-577468
14. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-577200

15. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-577234
16. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-574728
17. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/#comment-570647
18. https://g.ezoic.net/privacy/learncpp.com