24.8 — Hiding inherited functionality

ALEX[1] JULY 17, 2024

Changing an inherited member's access level

C++ gives us the ability to change an inherited member's access specifier in the derived class. This is done by using a *using declaration* to identify the (scoped) base class member that is having its access changed in the derived class, under the new access specifier.

For example, consider the following Base:

```
#include <iostream>

class Base
{
private:
    int m_value {};

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

protected:
    void printValue() const { std::cout << m_value; }
};
```

Because Base::printValue() has been declared as protected, it can only be called by Base or its 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 value)
        : Base { value }
    {
    }

    // Base::printValue was inherited as protected, so the public has no access
    // But we're changing it to public via a using declaration
    using Base::printValue; // note: no parenthesis here
};
```

This means that this code will now work:

```
int main()
{
    Derived derived { 7 };

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

You can only change the access specifiers of base members the derived 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.

Hiding functionality

In C++, it is not possible to remove or restrict functionality from a base class other than by modifying the source code. However, in a derived class, it is possible to hide functionality that exists in the base class, so that it can not be accessed through the derived class. This can be done simply by changing the relevant access specifier.

For example, we can make a public member private:

```
#include <iostream>

class Base
{
public:
    int m_value{};
};

class Derived : public Base
{
private:
    using Base::m_value;

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

int main()
{
    Derived derived{ 7 };
    std::cout << derived.m_value; // error: m_value is private in Derived

    Base& base{ derived };
    std::cout << base.m_value; // okay: m_value is public in Base

    return 0;
}
```

This allowed us to take a poorly designed base class and encapsulate its data in our derived class. Alternatively, instead of inheriting Base's members publicly and making m_value private by overriding its 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.

However, it is worth noting that while m_value is private in the Derived class, it is still public in the Base class. Therefore the encapsulation of m_value in Derived can still be subverted by casting to Base& and directly accessing the member.

For advanced readers

For the same reason, if a Base class has a public virtual function, and the Derived class changes the access specifier to private, the public can still access the private Derived function by casting a Derived object to a Base& and calling the virtual function. The compiler will allow this because the function is public in Base. However, because the object is actually a Derived, virtual function resolution will resolve to (and call) the (private) Derived version of the function. Access controls are not enforced at runtime.

```
#include <iostream>

class A
{
public:
    virtual void fun()
    {
        std::cout << "public A::fun()\n";
    }
};

class B : public A
{
private:
    virtual void fun()
    {
         std::cout << "private B::fun()\n";
   }
};

int main()
{
    B b {};
    b.fun();                    // compile error: not allowed as B::fun() is private
    static_cast<A&>(b).fun(); // okay: A::fun() is public, resolves to private
B::fun() at runtime

    return 0;
}
```

Perhaps surprisingly, given a set of overloaded functions in the base class, there is no way to change the access specifier for a single overload. You can only change them all:

```cpp
#include <iostream>

class Base
{
public:
    int m_value{};

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

class Derived : public Base
{
private:
    using Base::getValue; // make ALL getValue functions private

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

int main()
{
    Derived derived{ 7 };
    std::cout << derived.getValue();  // error: getValue() is private in Derived
    std::cout << derived.getValue(5); // error: getValue(int) is private in Derived

    return 0;
}
```

Deleting functions in the derived class

You can also mark member functions as deleted in the derived class, which ensures they can't be called at all through a derived object:

```
#include <iostream>
class Base
{
private:
    int m_value {};

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

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

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


    int getValue() const = delete; // mark this function as inaccessible
};

int main()
{
    Derived derived { 7 };

    // The following won't work because getValue() has been deleted!
    std::cout << derived.getValue();

    return 0;
}
```

In the above example, we've marked the getValue() function as deleted. This means that the compiler will complain when we try to call the derived version of the function. Note that the Base version of getValue() is still accessible though. We can call Base::getValue() in one of two ways:

```
int main()
{
    Derived derived { 7 };

    // We can call the Base::getValue() function directly
    std::cout << derived.Base::getValue();

    // Or we can upcast Derived to a Base reference and getValue() will resolve to
Base::getValue()
    std::cout << static_cast<Base&>(derived).getValue();

    return 0;
}
```

If using the casting method, we cast to a Base& rather than a Base to avoid making a copy of the Base portion of `derived`.

Next lesson
24.9 Multiple inheritance

2

Back to table of contents

3

Previous lesson
24.7 Calling inherited functions and overriding behavior

4

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

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.

124 COMMENTS

Newest

Nidhi Gupta
May 6, 2025 9:10 am PDT

using Base::someMember; // Changes access level to current section (e.g., public)
✅ Allowed: protected → public or private
❌ Not allowed: private → protected or public (because derived class can't access private base members)

2. Hiding Base Class Members

You can restrict access to inherited members:

cpp
Copy

Edit
private:
using Base::someMember; // Makes it inaccessible through Derived
Effective in Derived, but:

Not truly private — accessible via static_cast<Base&> or direct scope resolution.

3. Virtual Function Access and Runtime Behavior

Changing a virtual function's access in a derived class doesn't stop runtime dispatch:

cpp
Copy
Edit
class A { public: virtual void fun(); };
class B : public A { private: void fun() override; };

static_cast<A&>(b).fun(); // Will still call B::fun() despite being private

0 Reply

Asicx
September 18, 2024 8:05 am PDT

I have two questions:

1. In this line `Base& base{ derived };`, you didn't use a C style cast or static_cast..., does this mean that all derived classes can be implicitly converted to base/parent even without a constructor (using implicit/default constructor) ?
2. The las line of this lesson threw me off: "to avoid making a copy of the Base portion of derived". Can you give us more details about how inherited class object's memory is allocated ?

Is it one block of memory with a uniques address for both base and derived class objects or is it something similar to containers (multiple contiguous addresses for each parent...).

Last edited 9 months ago by Asicx

3 Reply

Alex Author
Reply to Asicx [8] September 22, 2024 8:35 pm PDT

1. Assuming public inheritance, yes, since the Derived is a Base.
2. Yes. Yes, when you create a Derived, a single block is allocated for the entire object, and the object's address is the first byte of that object in memory. Let's say Base has a size of 8 and Derived has a size of 16 (excluding Base). When allocating a Derived, the compiler will allocate 24 bytes (assuming no padding). The first 8 bytes are for the Base portion, and the next 16 bytes are for the Derived portion. The Derived constructor initializes the Derived portion, and the Base constructor initializes the Base portion.

9 Reply

bettingbear
July 29, 2024 8:35 am PDT

i have a feeling it is, but is it possible to have code write itself some code? (1 way or another)

Last edited 11 months ago by bettingbear

1 Reply

Alex Author
Reply to bettingbear [9] July 30, 2024 4:32 pm PDT

Code is just a text file, so not directly. But code that has been compiled or interpreted can produce code as an output.

1 Reply

Ruchika
July 27, 2024 10:10 am PDT

In the previous lesson you changed the access specifier of a private member function print (in base) to public in derived , and in this lesson you are saying it's not possible to change the access of private member, and you are using - using approach here to change access. I am confused.

0 Reply

Alex Author
Reply to Ruchika [10] July 27, 2024 12:22 pm PDT

In the previous lesson, the derived class defined a public member function that happened to have the same name as a private member function in the base class. It doesn't matter that the base member function is private here since it is never accessed from the derived class.

In this lesson, we're talking about changing the access level of an inherited member. This can only be done to non-private members since the base member function is being referenced.

2 Reply

Ph3r0X
July 18, 2024 5:56 am PDT

Hey Alex and Nascardriver,

in the last code snippet of this section you are using derived.Base::getvalue(), to call the getvalue method of the Base class.

What I don't understand is the use of the member selection operator:

I am aware that the method is not static and therefore requires that it is called by an instance via the member selection operator (.) or the member pointer operator (->), to provide the hidden this pointer. However, this basically means, that Base::getvalue (scope resolution operator resolves prior to member

selection operator and function call operator) is basically a publicly accessible member function of the Derived class...

Am I understanding this correctly?

Thanks for your answer in advance!

0 Reply

Alex Author

Reply to Ph3r0X [11] July 18, 2024 9:03 pm PDT

No. Base::getValue() is a member function of Base. Derived::getValue() shadows the Base version within the scope of Derived. Using the `Base::` qualifier tells the compiler to use the Base version rather than the shadowing Derived version.

2 Reply

Karl

June 5, 2024 3:28 pm PDT

Near as I can tell, it is not possible to **overload** a function across an inheritance boundary. For example:

```
1  class Base
2  {
3  public:
4      bool greaterThanZero(double d)
5      {
6          std::cout << "In base class.\n";
7          return d > 0.0;
8      }
9  };
10
11 class Derived: public Base
12 {
13 public:
14     bool greaterThanZero(int i)
15     {
16         std::cout << "In derived class.\n";
17         return i > 0;
18     }
19 };
20
21
22 int main()
23 {
24     Derived derived;
25     derived.greaterThanZero(3.14159);
26
27     return 0;
28 }
```

This will print "In derived class." **not** "In base class." even though the provided argument (a double) is a better match for the base version of the function. This is because as soon as the compiler sees the function in the derived class, it will not look beyond the derived class for other implementations.

If you do want to overload functions that are partially defined in a parent class, you would have to do something like this in the `Derived` class:

```
class Derived: public Base
{
public:
    bool greaterThanZero(double d)
    {
        return Base::greaterThanZero(d);
    }

    bool greaterThanZero(int i)
    {
        std::cout << "In derived class.\n";
        return i > 0;
    }
};
```

Just putting this here as an FYI.

2 Reply

Alex Author

Reply to Karl [12] June 7, 2024 1:58 pm PDT

Thanks for identifying this gap. I added a discussion of this to the bottom of https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/, including a better method for doing the same thing.

4 Reply

Hossein

January 2, 2024 4:56 pm PST

Is there a way to change the visibility of the whole base class?

```
//base structure from an external lib
struct A{...};

class B: protected A
{
public:
//non-templated stuff need to see A here
private:
...
};

template<typename T>
class C: public B
{
public:
//templated stuff
//templated stuff need to see A as well, but derived classes from C shouldn't.
};
```

Last edited 1 year ago by Hossein

1 Reply

Alex Author

Reply to Hossein [13] January 5, 2024 4:28 pm PST

I don't think so.

1 Reply

D D

December 8, 2023 10:55 am PST

Hello.

1. Why do you use here `ref`, not a copy?

```
Derived derived{ 7 };
    std::cout << derived.m_value; // error: m_value is private in Derived

    Base& base{ static_cast<Base&>(derived) };
```

and here

```
std::cout << static_cast<Base&>(derived).getValue();
```

2. Mark your methods `const`

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

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

0 Reply

Alex Author

Reply to D D [14] December 11, 2023 9:33 am PST

1. To show that we can access the Base member of `derived` directly via a Base reference. If we make a copy of `derived` into `base`, then we slice `derived`.
2. Fixed, thanks!

0 Reply

Zoltan

October 27, 2023 12:42 pm PDT

Maybe this is a silly question, but how can I test that the access specifier of a function has indeed been changed e.g. from protected to public?

What can a public one do that a protected can't?
For a void function I couldn't think of anything

0 Reply

Alex Author

Reply to Zoltan [15] October 28, 2023 6:07 pm PDT

A public function can be accessed from outside the class, whereas a protected one can only be accessed by members of the class or derived members.

So if you want to verify that a member is now public, instantiate an object of the class type from main() and see if you can call the member function on that object.

0 Reply

...

September 10, 2023 1:14 pm PDT

Hi alex I have 2 questions:

1. what does upcast means in the context of this lesson ?, do I understand it right ? (cast the derived object to the parent)

2.in this lesson and the last one u was doing `static_cast` to `Base&` how its possible to cast it to reference (instead of `Base`) and how does the compiler knows how to convert it to the base, is it because we inherited from it is that right ?

sry if my questions a bit vauge

0 Reply

Alex Author

Reply to ... [16] September 14, 2023 8:47 am PDT

1. Yes, upcast is derived to base.
2. Yes. If we upcast to Base&, then we have a Base reference to a Derived object. If we upcast to Base then we get a sliced object.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/
5. https://www.learncpp.com/hiding-inherited-functionality/
6. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-602087
9. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-600233
10. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-600166
11. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-599773
12. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-598034
13. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-591687
14. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-590677
15. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-589167
16. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/#comment-586937
17. https://g.ezoic.net/privacy/learncpp.com