25.3 — The override and final specifiers, and covariant return types

ALEX[1] DECEMBER 29, 2024

To address some common challenges with inheritance, C++ has two inheritance-related identifiers: `override` and `final`. Note that these identifiers are not keywords -- they are normal words that have special meaning only when used in certain contexts. The C++ standard calls them "identifiers with special meaning", but they are often referred to as "specifiers".

Although final isn't used very much, override is a fantastic addition that you should use regularly. In this lesson, we'll take a look at both, as well as one exception to the rule that virtual function override return types must match.

The override specifier

As we mentioned in the previous lesson, a derived class virtual function is only considered an override if its signature and return types match exactly. That can lead to inadvertent issues, where a function that was intended to be an override actually isn't.

Consider the following example:

```
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName1(int x) { return "A"; }
    virtual std::string_view getName2(int x) { return "A"; }
};

class B : public A
{
public:
    virtual std::string_view getName1(short x) { return "B"; } // note: parameter is a
short
    virtual std::string_view getName2(int x) const { return "B"; } // note: function
is const
};

int main()
{
    B b{};
    A& rBase{ b };
    std::cout << rBase.getName1(1) << '\n';
    std::cout << rBase.getName2(2) << '\n';

    return 0;
}
```

Because rBase is an A reference to a B object, the intention here is to use virtual functions to access B::getName1() and B::getName2(). However, because B::getName1() takes a different parameter (a short instead of an int), it's not considered an override of A::getName1(). More insidiously, because B::getName2() is const and A::getName2() isn't, B::getName2() isn't considered an override of A::getName2().

Consequently, this program prints:

```
A
A
```

In this particular case, because A and B just print their names, it's fairly easy to see that we messed up our overrides, and that the wrong virtual function is being called. However, in a more complicated program, where the functions have behaviors or return values that aren't printed, such issues can be very difficult to debug.

To help address the issue of functions that are meant to be overrides but aren't, the `override` specifier can be applied to any virtual function to tell the compiler to enforce that the function is an override. The `override` specifier is placed at the end of a member function declaration (in the same place where a function-level `const` goes). If a member function is `const` and an override, the `const` must come before `override`.

If a function marked as `override` does not override a base class function (or is applied to a non-virtual function), the compiler will flag the function as an error.

```
#include <string_view>

class A
{
public:
    virtual std::string_view getName1(int x) { return "A"; }
    virtual std::string_view getName2(int x) { return "A"; }
    virtual std::string_view getName3(int x) { return "A"; }
};

class B : public A
{
public:
    std::string_view getName1(short int x) override { return "B"; } // compile error,
function is not an override
    std::string_view getName2(int x) const override { return "B"; } // compile error,
function is not an override
    std::string_view getName3(int x) override { return "B"; } // okay, function is an
override of A::getName3(int)

};

int main()
{
    return 0;
}
```

The above program produces two compile errors: one for B::getName1(), and one for B::getName2(), because neither override a prior function. B::getName3() does override A::getName3(), so no error is produced for that line.

Because there is no performance penalty for using the override specifier and it helps ensure you've actually overridden the function you think you have, all virtual override functions should be tagged using the

override specifier. Additionally, because the override specifier implies virtual, there's no need to tag functions using the override specifier with the virtual keyword.

Best practice

Use the virtual keyword on virtual functions in a base class.

Use the override specifier (but not the virtual keyword) on override functions in derived classes. This includes virtual destructors.

Rule

If a member function is both `const` and an `override`, the `const` must be listed first. `const override` is correct, `override const` is not.

The final specifier

There may be cases where you don't want someone to be able to override a virtual function, or inherit from a class. The final specifier can be used to tell the compiler to enforce this. If the user tries to override a function or inherit from a class that has been specified as final, the compiler will give a compile error.

In the case where we want to restrict the user from overriding a function, the **final specifier** is used in the same place the override specifier is, like so:

```
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B : public A
{
public:
    // note use of final specifier on following line -- that makes this function not
able to be overridden in derived classes
    std::string_view getName() const override final { return "B"; } // okay, overrides
A::getName()
};

class C : public B
{
public:
    std::string_view getName() const override { return "C"; } // compile error:
overrides B::getName(), which is final
};
```

In the above code, B::getName() overrides A::getName(), which is fine. But B::getName() has the final specifier, which means that any further overrides of that function should be considered an error. And indeed, C::getName() tries to override B::getName() (the override specifier here isn't relevant, it's just there for good practice), so the compiler will give a compile error.

In the case where we want to prevent inheriting from a class, the final specifier is applied after the class name:

```cpp
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B final : public A // note use of final specifier here
{
public:
    std::string_view getName() const override { return "B"; }
};

class C : public B // compile error: cannot inherit from final class
{
public:
    std::string_view getName() const override { return "C"; }
};
```

In the above example, class B is declared final. Thus, when C tries to inherit from B, the compiler will give a compile error.

Covariant return types

There is one special case in which a derived class virtual function override can have a different return type than the base class and still be considered a matching override. If the return type of a virtual function is a pointer or a reference to some class, override functions can return a pointer or a reference to a derived class. These are called **covariant return types**. Here is an example:

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
    // This version of getThis() returns a pointer to a Base class
    virtual Base* getThis() { std::cout << "called Base::getThis()\n"; return this; }
    void printType() { std::cout << "returned a Base\n"; }
};

class Derived : public Base
{
public:
    // Normally override functions have to return objects of the same type as the base function
    // However, because Derived is derived from Base, it's okay to return Derived* instead of Base*
    Derived* getThis() override { std::cout << "called Derived::getThis()\n";  return this; }
    void printType() { std::cout << "returned a Derived\n"; }
};

int main()
{
    Derived d{};
    Base* b{ &d };
    d.getThis()->printType(); // calls Derived::getThis(), returns a Derived*, calls Derived::printType
    b->getThis()->printType(); // calls Derived::getThis(), returns a Base*, calls Base::printType

    return 0;
}
```

This prints:

```
called Derived::getThis()
returned a Derived
called Derived::getThis()
returned a Base
```

One interesting note about covariant return types: C++ can't dynamically select types, so you'll always get the type that matches the actual version of the function being called.

In the above example, we first call d.getThis(). Since d is a Derived, this calls Derived::getThis(), which returns a `Derived*`. This `Derived*` is then used to call non-virtual function Derived::printType().

Now the interesting case. We then call b->getThis(). Variable b is a Base pointer to a Derived object. Base::getThis() is a virtual function, so this calls Derived::getThis(). Although Derived::getThis() returns a `Derived*`, because Base version of the function returns a `Base*`, the returned Derived* is upcast to a `Base*`. Because Base::printType() is non-virtual, Base::printType() is called.

In other words, in the above example, you only get a `Derived*` if you call getThis() with an object that is typed as a Derived object in the first place.

Note that if printType() were virtual instead of non-virtual, the result of b->getThis() (an object of type `Base*`) would have undergone virtual function resolution, and Derived::printType() would have been called.

Covariant return types are often used in cases where a virtual member function returns a pointer or reference to the class containing the member function (e.g. Base::getThis() returns a `Base*`, and Derived::getThis() returns a `Derived*`). However, this isn't strictly necessary. Covariant return types can be used in any case where the return type of the override member function is derived from the return type of the base virtual member function.

Quiz time

Question #1

What does the following program output?

```cpp
#include <iostream>

class A
{
public:
    void print()
    {
        std::cout << "A";
    }
    virtual void vprint()
    {
        std::cout << "A";
    }
};
class B : public A
{
public:
    void print()
    {
        std::cout << "B";
    }
    void vprint() override
    {
        std::cout << "B";
    }
};


class C
{
private:
    A m_a{};

public:
    virtual A& get()
    {
        return m_a;
    }
};

class D : public C
{
private:
    B m_b{};

public:
    B& get() override // covariant return type
    {
        return m_b;
    }
};

int main()
{
    // case 1
    D d {};
    d.get().print();
    d.get().vprint();
    std::cout << '\n';

    // case 2
    C c {};
    c.get().print();
    c.get().vprint();
    std::cout << '\n';

    // case 3
    C& ref{ d };
    ref.get().print();
    ref.get().vprint();
```

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

    return 0;
}
```

Show Solution (javascript:void(0))[2]

Question #2

When would we use function overloading vs function overriding?

Show Solution (javascript:void(0))[2]

Next lesson

25.4 Virtual destructors, virtual assignment, and overriding virtualization

3

Back to table of contents

4

Previous lesson

25.2 Virtual functions and polymorphism

5

6

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/[9] are connected to your provided email address.

159 COMMENTS

Newest

Jay

February 14, 2025 8:16 am PST

Do you think there would be any benefit to readability if you added virtual AND override to intermediary objects meant to be a base and a child? For example A -> B -> C, A's methods would just have the virtual keyword to indicate that it can be overridden, B's methods would have virtual and override to indicate it can be overridden and act as an override, and C's methods would just have override to indicate its the most derived object. This would purely be for documentation because as was stated override implies virtual.

0 Reply

Alex Author

Reply to Jay [10] February 15, 2025 3:05 pm PST

Since you can only override a virtual function, override implies virtual. Having both is redundant.

Of course, nothing says you can't be redundant if you prefer it.

1 Reply

Swaminathan

July 14, 2024 7:27 am PDT

But why is override only allowed on Virtual function and not on normal function overrides? As an example,

```
class Parent
{
protected:
    void myFun()
    {
        std::cout << "This does something";
    }
};

class Child : public Parent
{
    void myFun() override // this errors...
    {
        std::cout << "I want to make sure this is an override of a Parent class
function";
    }
};
```

Even in this case, it can serve as a documentation and make sure we're overriding something from parent.

Last edited 11 months ago by Swaminathan

0 Reply

Alex Author

Reply to Swaminathan [11] July 17, 2024 2:00 pm PDT

Because technically `Child::myFun()` isn't an override. `Child::myFun()` hides/shadows `Parent::myFun()`.

I'm guessing they didn't allow for this because it would have required a change to the definition of override.

0 Reply

yourmom

April 26, 2024 8:48 am PDT

The content is great, thank you for providing so <3

2 Reply

MaxRob

January 9, 2024 7:28 am PST

So what's the purpose of convariant return type, if the actually return a reference of a base object type, why changing the type? Why override a function with B& if at hte end it return an A& like the overriden one? No matter if reference returned is binding to it's object or a derived obejct, the behavior will be identical. In fact in the quiz program the output is the same changing the convariant type to A&, and as a base class, through the reference is impossible to access specific derived class function, and virtual ones will behave equally depending of the object binded. What's the pointof all this?

2 Reply

Alex Author

Reply to MaxRob [12] January 10, 2024 3:11 pm PST

The purpose is to allow a function defined in the derived class to return a Derived instead of a Base when it is called with a Derived object.

Changing the return type of `D::get()` from `B&` to `A&` changes the output of `d.get().print()`;

1 Reply

MaxRob

Reply to Alex [13] January 10, 2024 9:45 pm PST

I focused only on the reference case. So when invoking virtual resolution the return is upcasted to the type of the base function, when called directly the return type isn't upcasted. Actually it could be useful in a lot of case. Because the compiler doesn't know if the reference A& (or pointer) is pointing to the base type A or a derived B, in this situation an upcast is made even when not necessay, like when A& is pointing to an A, or there is another way?

1 Reply

MaxRob

Reply to MaxRob [14] January 10, 2024 10:33 pm PST

Edit: actually the upcast is nothing other than how the compiler consider the data type (and so the operation on it). Both an A& and B& are address so there isn't a real data conversion.

1 Reply

Suryaansh

December 18, 2023 9:20 pm PST

I needed to dig further to understand the Covariant return types. Summarising my understanding here for those who might need it.

Function Signatures:
Ideally, the function signatures of a virtual function in the base class and its override in the derived class should be the same.

Covariant Return Types:
The exception to the identical signatures is covariant return types. The derived class can override a virtual function with a return type that is a subclass (or a derived type) of the return type in the base class.

Upcasting:
However, when the derived class overrides a virtual function, you use it through a pointer or reference to the base class, because the return type is upcasted to the type declared in the base class.

Virtual Function Resolution:
During the resolution of virtual functions, the most derived version is invoked. If a derived class overrides a virtual function, and you call that function through a base class pointer or reference, the actual implementation in the most derived class is executed.

Covariant Return Types:
In contrast, when it comes to the return type of a virtual function, covariant return types allow the returned pointer or reference to be upcasted to the type declared in the base class. This ensures a consistent interface when using polymorphism with base class pointers, even though the actual object might be of a more derived type.

So, you can think of it as follows:
Virtual Function Invocation: Goes to the most derived implementation.
Covariant Return Types: Goes up the inheritance hierarchy to the type declared in the base class.

Example -:

```cpp
#include <iostream>

class Animal {
public:
    // Virtual function in the base class
    virtual Animal* create() const {
        std::cout << "Creating an Animal\n";
        return new Animal();
    }

    // Virtual destructor to ensure proper cleanup
    virtual ~Animal() {
        std::cout << "Animal Destructor\n";
    }
};

class Dog : public Animal {
public:
    // Covariant return type: returns a pointer to a Dog
    Dog* create() const override {
        std::cout << "Creating a Dog\n";
        return new Dog();
    }

    // Dog-specific function
    void bark() const {
        std::cout << "Woof! Woof!\n";
    }

    // Destructor to ensure proper cleanup
    ~Dog() override {
        std::cout << "Dog Destructor\n";
    }
};

int main() {
    // Create a Dog object using the base class pointer
    Animal* animalPtr = new Dog();

    // Call the overridden create() function
    // Since create() is a virtual function, due to Covariant Return types the returned pointer is Animal* and not Dog*
    Animal* dogPtr = animalPtr->create();

    // Can't use Dog*
    // Dog* dogPtr = animalPtr->create(); Compile error invalid conversion from ‘Animal*’ to ‘Dog*’

    // We cannot use Dog-specific function
    // dogPtr->bark(); Compile error - ‘class Animal’ has no member named ‘bark’

    // Cleanup
    delete animalPtr;  // Calls the Dog Destructor via the virtual destructor in Animal

    return 0;
}
```

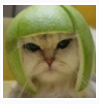

D D
December 10, 2023 12:19 am PST

Hello.

1. You've forgotten yo add the `const` specifier for `getName`
2. When you want to mark your method with `final`, that you may omit `override`. It'll work

```
class B : public A
{
public:
    // note use of final specifier on following line -- that makes this function no
longer overridable
    std::string_view getName() override final { return "B"; } // okay, overrides
A::getName()
};
```

0 Reply



Alex Author

Reply to D D [15] December 11, 2023 11:46 am PST

1. Added
2. `final` and `override` do different things and can be combined. In this example, we use `override` to ensure the function is an override of the version from the base class, and `final` to prevent overrides of the function in derived classes.

0 Reply

atutsasha

November 4, 2023 2:51 am PDT

Use the override specifier (but not the virtual keyword) on override functions in derived classes. This includes virtual destructors.

```
void vprint() override
{
  std::cout << "B" << '\n';
}
```

0 Reply

Zoltan

October 28, 2023 4:24 am PDT

Sorry, ChatGPT doesnt know the answer to this one :(

I'm trying to reproduce the same behavior. I'm trying to make my object get "upcast" to Base, so that I see "B, A" on the screen but no matter how many times I look through this, this seems exactly the same as yours and it keeps printing B,B.
In other words, my code correctly finds the Derived class, not the Base.

What should I change in order to make it print B, A?

```cpp
#include <iostream>

class Base
{
public:
    virtual std::string_view getName() const
    {
        return "A";
    }

    virtual Base* getThis() { return this; }
};

class Derived : public Base
{
public:
    std::string_view getName() const
    {
        return "B";
    }

    Derived* getThis() override { return this; }
};


int main()
{

    Derived d{};
    Base* b{ &d };
    std::cout << d.getThis()->getName() << "\n";
    std::cout << static_cast<Base*>(b)->getThis()->getName();

    return 0;
}
```

0 Reply

Alex Author

Reply to Zoltan [16] October 28, 2023 6:19 pm PDT

Make `Base::getName()` non-virtual.

3 Reply

Zoltan
October 28, 2023 3:56 am PDT

I don't seem to "getThis()" part
I was legit going to ask a question and it dawned on me as I was typing it lol

In case 2 it's not AB, but AA, because the return type of C::get() is A&, which means we are not looking "from A to C" for the most-derived vprint() function. We are only looking at A to find the only vprint() function there is!

Right?!

0 Reply

Pilot
October 15, 2023 9:52 am PDT

In regards to the covariant return type, I can only get the derived class override to return a pointer (this) to a derived class object.
The return value can't be assigned to a derived class pointer(Indicating it is a base class pointer), but any virtual function called on the return value invariably invokes the derived class override.
This seems like a contradiction, what am I missing?

0 Reply

Alex Author
Reply to Pilot [17] October 17, 2023 8:21 pm PDT

What seems like a contradiction?

This behavior is described in the lesson, in the 3 paragraphs starting with "Now the interesting case."

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/virtual-functions/
6. https://www.learncpp.com/the-override-and-final-specifiers-and-covariant-return-types/
7. https://www.learncpp.com/cpp-tutorial/chapter-24-summary-and-quiz/
8. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/

9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-607739
11. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-599586
12. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-592048
13. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-592149
14. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-592164
15. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-590723
16. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-589180
17. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/#comment-588847
18. https://g.ezoic.net/privacy/learncpp.com