12.2a — The override and final specifiers, and covariant return types

BY ALEX ON NOVEMBER 6TH, 2016 | LAST MODIFIED BY ALEX ON MAY 20TH, 2018

To address some common challenges with inheritance, C++11 added two special identifiers to C++: override and final. Note that these identifiers are not considered keywords -- they are normal identifiers that have special meaning in certain contexts.

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:

```
class A
{
public:
    virtual const char* getName1(int x) { return "A"; }
    virtual const char* getName2(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const char* getName1(short int x) { return "B"; } // note: parameter is a short int
    virtual const char* 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 int 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, C++11 introduced the **override specifier**. Override can be applied to any override function by placing the specifier in the same place const would go. If the function does not override a base class function, the compiler will flag the function as an error.

```
class A
{
public:
```

```
    virtual const char* getName1(int x) { return "A"; }
    virtual const char* getName2(int x) { return "A"; }
    virtual const char* getName3(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const char* getName1(short int x) override { return "B"; } // compile error, function is no
t an override
    virtual const char* getName2(int x) const override { return "B"; } // compile error, function is no
t an override
    virtual const char* 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.

There is no performance penalty for using the override specifier, and it helps avoid inadvertent errors. Consequently, we highly recommend using it for every virtual function override you write to ensure you've actually overridden the function you think you have.

Rule: Apply the override specifier to every intended override function you write.

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 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:

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};

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

class C : public B
{
public:
    virtual const char* getName() override { return "C"; } // compile error: overrides B::getName(), wh
ich 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:

```
class A
{
```

```
public:
    virtual const char* getName() { return "A"; }
};

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

class C : public B // compile error: cannot inherit from final class
{
public:
    virtual const char* getName() 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 a class, override functions can return a pointer or a reference to a derived class. These are called **covariant return types**. Here is an example:

```
#include <iostream>

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*
    virtual Derived* getThis() { 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
}
```

This prints:

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

Note that some older compilers (e.g. Visual Studio 6) do not support covariant return types.

One interesting note about covariant return types: C++ can’t dynamically select types, so you’ll always get the type that matches the base 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 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 downcast to a Base*. And thus, 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.

12.3 -- Virtual destructors, virtual assignment, and overriding virtualization

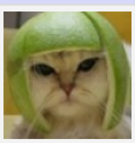
Index

12.2 -- Virtual functions and polymorphism

26 comments to 12.2a — The override and final specifiers, and covariant return types

SandeepD
May 19, 2018 at 6:41 am · Reply

In the text below the last code, there is a type - "Since d is a Derived, this calls Derived::getBase(), which returns a Derived*. "
getBase() should be getThis()
Surprisingly nobody noticed ☺

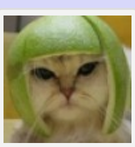


Alex
May 20, 2018 at 1:06 pm · Reply

Thanks for the fix. This code sample was added to the site fairly recently, so it hasn't had as many readers as other parts of the site. Thanks for pointing out the error!

Luhan
November 3, 2017 at 6:32 am · Reply

Would you know how to explain what is exactly a dynamic select type? I found this [https://en.wikipedia.org/wiki/Dynamic_dispatch] but I don't know if is it.



Alex
November 3, 2017 at 8:23 pm · Reply

I don't know what a "dynamic select type" is. Dynamic dispatch is a generic name for virtual function resolution.

Luhan