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

BY ALEX ON NOVEMBER 6TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
1  class A
2  {
3  public:
4      virtual const char* getName1(int x) { return "A"; }
5      virtual const char* getName2(int x) { return "A"; }
6  };
7
8  class B : public A
9  {
10 public:
11     virtual const char* getName1(short int x) { return "B"; } // note: parameter is a short in
12     virtual const char* getName2(int x) const { return "B"; } // note: function is const
13 };
14
15 int main()
16 {
17     B b;
18     A &rBase = b;
19     std::cout << rBase.getName1(1) << '\n';
20     std::cout << rBase.getName2(2) << '\n';
21
22     return 0;
23 }
```

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**. The override specifier 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 (or is applied to a non-virtual 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, funct
    virtual const char* getName2(int x) const override { return "B"; } // compile error, funct
    virtual const char* getName3(int x) override { return "B"; } // okay, function is an overr

};

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 ove
    virtual const char* getName() override final { return "B"; } // okay, overrides A::getName
};
```

```
13
14  class C : public B
15  {
16  public:
17      virtual const char* getName() override { return "C"; } // compile error: overrides B::getN
18  };
```

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:

```
1   class A
2   {
3   public:
4       virtual const char* getName() { return "A"; }
5   };
6
7   class B final : public A // note use of final specifier here
8   {
9   public:
10      virtual const char* getName() override { return "B"; }
11  };
12
13  class C : public B // compile error: cannot inherit from final class
14  {
15  public:
16      virtual const char* getName() override { return "C"; }
17  };
```

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:

```
1   #include <iostream>
2
3   class Base
4   {
5   public:
6       // This version of getThis() returns a pointer to a Base class
7       virtual Base* getThis() { std::cout << "called Base::getThis()\n"; return this; }
8       void printType() { std::cout << "returned a Base\n"; }
9   };
10
11  class Derived : public Base
12  {
13  public:
14      // Normally override functions have to return objects of the same type as the base functio
15      // However, because Derived is derived from Base, it's okay to return Derived* instead of
16      virtual Derived* getThis() { std::cout << "called Derived::getThis()\n";  return this; }
17      void printType() { std::cout << "returned a Derived\n"; }
18  };
19
```

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

    return 0;
}
```

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 upcast 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**

C++ TUTORIAL | PRINT THIS POST

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

Sergey
September 12, 2019 at 10:34 am · Reply

Hi! The author wrote, "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".

I am a little bit confusing, cause it turns out that "override" and "final" is not applicable when overriding normal member functions. The following program does not compile.

```
1  #include <string>
2  #include <iostream>
3
4  class Base
5  {
6  public:
7      int getValue() { return 5; }
8  };
9
10 class Derived: public Base
11 {
12 public:
13     int getValue() override { return 6.78; }
14 };
15
16 int main() {
17     Derived d;
18
19     std::cout << d.getValue();
20     return 0; // compiler error: ‘int Derived::getValue()’ marked ‘override’, but does not
21 }
```

Am I right, that "override" and "final" is only applicable to virtual functions? Thanks in advance!

nascardriver
September 12, 2019 at 11:44 pm · Reply

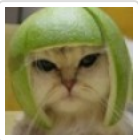
Yes, it only applies to `virtual` functions.

cdecde57
August 3, 2019 at 9:03 am · Reply

Hello! I think I understand everything here except the covariant return types fully. Pretty much what I see is that we have a base and a derived class. Because the derived one is a child of the base class we are able to return different types like instead of returning a base class we can do a derived one because it is a child / inherited traits from the main base like in the previous lessons being able to make an array of animals that consisted of dogs and cats.

Please help me with covariant return types I don't know if I am completely wrong or anything and feedback would be great. Thanks!

And thank you for the tutorials btw this is the ONLY place I have learned c++ so far and I have gone to a couple of other places just to see what they are teaching and you do everything way better, from the detail and clarity to the in-depth reasons behind why you do what you do.



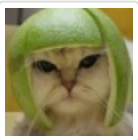
Alex
August 4, 2019 at 5:34 pm · Reply

Yes, covariant return types allow virtual functions in derived classes to return a Derived* if the base version of the function returns a Base*. That's really it.

nascardriver
July 15, 2019 at 7:41 am · Reply

Hi Alex!

The first sentence of this lesson will need adjustment when C++20 is released. I won't list the new special identifiers, as they might still change.
I won't tell you about all changes, but this is one that's easy to miss so I thought I'd point it out.



Alex
July 18, 2019 at 11:46 am · Reply

Hey Nas,

Are you just noting that there there will be additional special identifiers (e.g. audit, axiom, import, module, etc...) for contracts and modules? If so, I'll definitely want to do separate lessons for those two things.

nascardriver
July 18, 2019 at 11:48 am · Reply

Yes, separate lessons for sure. I'm just noting that this lesson will need to be updated as well, as it's easily overseen.

Nirbhay
August 19, 2019 at 6:02 am · Reply