When should your destructor be virtual?

devblogs.microsoft.com/oldnewthing/20040507-00

May 7, 2004

Raymond Chen

When should your C++ object's destructor be virtual?

First of all, what does it mean to have a virtual destructor?

Well, what does it mean to have a virtual method?

If a method is virtual, then calling the method on an object always invokes the method as implemented by the most heavily derived class. If the method is not virtual, then the implementation corresponding to the compile-time type of the object pointer.

For example, consider this:

```
class Sample {
public:
 void f();
 virtual void vf();
};
class Derived : public Sample {
public:
 void f();
 void vf();
}
void function()
{
 Derived d;
 Sample* p = &d;
 p->f();
 p->vf();
}
```

The call to `p->f()` will result in a call to `Sample::f` because `p` is a pointer to a `Sample`. The actual object is of type `Derived`, but the pointer is merely a pointer to a `Sample`. The pointer type is used because `f` is not virtual.

On the other hand, the call to The call to `p->vf()` will result in a call to `Derived::vf`, the most heavily derived type, because `vf` is virtual.

Okay, you knew that.

Virtual destructors work exactly the same way. It's just that you rarely invoke the destructor explicitly. Rather, it's invoked when an automatic object goes out of scope or when you `delete` the object.

```
void function()
{
 Sample* p = new Derived;
 delete p;
}
```

Since `Sample` does not have a virtual destructor, the `delete p` invokes the destructor of the class of the pointer (`Sample::~Sample()`), rather than the destructor of the most derived type (`Derived::~Derived()`). And as you can see, this is the wrong thing to do in the above scenario.

Armed with this information, you can now answer the question.

A class must have a virtual destructor if it meets both of the following criteria:

- You do a `delete p` .
- It is possible that `p` actually points to a derived class.

Some people say that you need a virtual destructor if and only if you have a virtual method. This is wrong in both directions.

Example of a case where a class has no virtual methods but still needs a virtual destructor:

```
class Sample { };
class Derived : public Sample
{
 CComPtr<IStream> m_p;
public:
 Derived() { CreateStreamOnHGlobal(NULL, TRUE, &m_p); }
};
Sample *p = new Derived;
delete p;
```

The `delete p` will invoke `Sample::~Sample` instead of `Derived::~Derived` , resulting in a leak of the stream `m_p` .

And here's an example of a case where a class has virtual methods but does not require a virtual destructor.

```
class Sample { public: virtual void vf(); }
class Derived : public Sample { public: virtual void vf(); }
Derived *p = new Derived;
delete p;
```

Since the object deletion occurs from the pointer type that matches the type of the actual object, the correct destructor will be invoked. This pattern happens often in COM objects, which expose several virtual methods corresponding to its interfaces, but where the object itself is destroyed by its own implementation and not from a base calss pointer. (Notice that no COM interfaces contain virtual destructors.)

The problem with knowing when to make your destructor virtual or not is that you have to know how people will be using your class. If C++ had a "sealed" keyword, then the rule would be simpler: If you do a "`delete p`" where `p` is a pointer to an unsealed class, then that class needs have a virtual destructor. (The imaginary "sealed" keyword makes it explicit when a class can act as the base class for another class.)

Raymond Chen

Follow