Understanding warning C4265: class has virtual functions, but destructor is not virtual, part 2

devblogs.microsoft.com/oldnewthing/20200619-00

June 19, 2020

Raymond Chen

Last time, we learned about what warning C4265 (class has virtual functions, but destructor is not virtual) is trying to say. The fact that it warns you even in the absence of a suspicious `delete` means that you can get quite a lot of false positives.

```
struct Interface
{
    virtual void f() = 0;
    virtual void Destroy() = 0;
};

struct Implementation : Interface
{
    ~Implementation();
    void f() override;
    void Destroy() override;
};
```

This triggers the warning because `Implementation` has a non-virtual destructor and virtual methods `f` and `Destroy`.

What the compiler doesn't realize is that the `Destroy` method is how the object is destructed. And the `Destroy` method is virtual, so everything is just fine. It doesn't realize this even if you make the `~Implementation()` destructor protected or private.

This pattern is common when implementing COM interfaces, where `Release` acts as the virtual destructor-ish method.

```
struct Thing : public IThing
{
  // IUnknown
  HRESULT QueryInterface(REFIID riid, void** ppv) override;

  ULONG AddRef() override { return ++m_refCount; }

  ULONG Release() override
  {
    auto refCount = --m_refCount;
    if (refCount == 0) delete this;
    return refCount;
  }

  // IThing
  HRESULT DoTheThing() override;

private:
  ~Thing();
  std::atomic_ulong m_refCount = 1;
};
```

This is pretty standard boilerplate for a COM object. The object's destructor is private because you are expected to destruct the object by calling `Release`, rather than calling `delete`. But it also generates a spurious warning C4265.

You can make the warning go away by declaring `Thing` as `final`.

But there are cases where you have a class that is meant to be a base class, so you don't want to mark it as `final`. If you're using C++/WinRT, you get to choose whether your implementation classes are final or not, and maybe you don't want them to be final, but you do need them to be COM objects, and when you do, warning C4265 will spuriously appear.

In WRL, an example of a non-final object with virtual methods is the `FtmBase`. This is a base class that implements `IMarshal` by forwarding all methods to the standard COM free-threade marshaler, thereby making your object free-threaded. (If you don't know what this means, don't worry. It's not important to the main point of this article, though it's a good thing to understand when you're working with COM.)

The point of the `FtmBase` is to be a base class for other COM objects. And when you do that, it is the `Release` method that triggers the destruction. Since `Release` is a virtual method, the `FtmBase::Release` will go to the most-derived object, and that implementation will perform the `delete` on the most-derived object, and everything will be fine.

But the compiler doesn't know that.

The compiler sees a class like `FtmBase` that has the potential for being used incorrectly. If somebody did, say,

```
class Weird : public FtmBase
{
};

void bad_idea()
{
  FtmBase* p = new Weird; // bad idea
  delete p; // even worse idea
}
```

they would stumble across the problem that warning C4265 is trying to tell you about.

But nobody should be writing code like that in the first place. WRL objects should be created with `Make`, and C++/WinRT objects should be created with `make`. In both cases, the object lifetimes are managed by `IUnknown`, which means that you `Release` the object when you're done. The smart pointer classes in WRL and C++/WinRT take care of this for you (similarly to `std::shared_ptr`), so you rarely see an explicit call to `Release`.

In C++/WinRT, the implementation template classes use CRTP, and the `Release` methods cast the `this` pointer to the most-derived type (the first template type parameter) before deleting, thereby accomplishing the same result in a different way.

So yes, if you use WRL or C++/WinRT, or more generally COM, you will see false positives of warning C4265. That may be one reason why warning C4265 is off by default.

Raymond Chen

Follow