Getting a strong reference from the this pointer too soon

devblogs.microsoft.com/oldnewthing/20240117-00

January 17, 2024

Raymond Chen

A bunch of us were investigating a crash that appeared to be due to the premature destruction of an object while they were still active COM references to it. After some investigation, we tracked it down: An object handed out a COM reference to itself in its constructor, even though it couldn't fulfill all the requirements of COM strong references.

Here's a simplified version of the mistake:

```
// C++/WinRT - this code is wrong
namespace winrt::Contoso::implementation
{
    struct MyPage : MyPageT<MyPage>
    {
        MyPage()
        {
            Application::LoadComponent(*this, blah, blah);
            something_that_might_throw();
        }

        [[ ... ]]
    };
}

// C++/WRL - this code is wrong
struct MyPage : RuntimeClass<IPage>
{
    MyPage()
    {
        Application::LoadComponent(this, blah, blah);
        something_that_might_throw();
    }

    [[ ... ]]
};

// Manually implemented - this code is wrong
struct MyPage : IPage
{
    MyPage() : m_refCount(1)
    {
        Application::LoadComponent(this, blah, blah);
        something_that_might_throw();
    }

    LONG m_refCount;
    [[ ... ]]
};

CComPtr<IPage> page;
page.Attach(new MyPage());
```

The C++/WinRT library convention for producing a COM reference to the current object is to use the * operator, so the line

```
// C++/WinRT
Application::LoadComponent(*this, blah, blah);
```

creates an `IInspectable` that refers to the current object and passes it to `LoadComponent`. If you use WRL or some other library that operates at the ABI level, then you just pass `this`, and the compiler will automatically convert it to a pointer to the `IInspectable` base class.

No matter how you slice it, what happens is that you passed a COM pointer to `Application::LoadComponent`, and therefore `LoadComponent` is within its rights to call `AddRef` on that COM pointer to extend the pointer's lifetime, and retain the pointer for later use.

When `Application::LoadComponent` returns, the object's reference count has been incremented to 2.

Unfortunately, what might happen next is that an exception or other error is encountered in the next part of the constructor. An exception in the constructor forces the object to destruct, and that means that the `MyPage` destructs without regard for its reference count.[1]

Later, some other code tries to use the COM pointer that `Application::LoadComponent` saved, and it crashes because it now points to freed memory.

Your code made a promise it couldn't keep. It told `Application::LoadComponent`, "Here's a COM pointer," and one of the things that COM pointers can do is extend their lifetime with the `AddRef` method. But this particular COM pointer doesn't honor that `AddRef` if an exception occurs.

The traditional solution to this problem is to have a rule (enforced by code review) that constructors may not throw exceptions if they also hand out COM references to themselves.

If you do hand out COM references, and you want to do things that might throw exceptions, then use two-phase construction, where all of the potentially-throwing operations are put in the second phase. WRL implements two-phase construction with the `RuntimeClass-Initialize()` method,[2] and starting in C++/WinRT version 2.0.220331.4, C++/WinRT uses the name `InitializeComponent()` for the second phase.

If the second phase of two-phase construction fails, then the caller is told that the object failed to construct, but if there are any outstanding COM references to the object, the object won't destruct until those outstanding references are released. You do have to be careful to leave your object in a harmless state after the exception is thrown, because those components which still have COM references will try to use them, and they should get some sort of reasonable behavior out of the zombie object. (What counts as "reasonable" is a domain-specific decision.)

In the above cases, we would do something like this:

```
// C++/WinRT, non-XAML class
namespace winrt::Contoso::implementation
{
    struct MyPage : MyPageT<MyPage>
    {
        void InitializeComponent()
        {
            Application::LoadComponent(*this, blah, blah);
            something_that_might_throw();
        }

        [[ ... ]]
    };
}

// C++/WinRT, XAML class
namespace winrt::Contoso::implementation
{
    struct MyPage : MyPageT<MyPage>
    {
        void InitializeComponent()
        {
            MyPageT::InitializeComponent(); // let base class initialize
            Application::LoadComponent(*this, blah, blah);
            something_that_might_throw();
        }

        [[ ... ]]
    };
}

// C++/WRL
struct MyPage : RuntimeClass<IPage>
{
    HRESULT RuntimeClassInitialize() try
    {
        Application::LoadComponent(this, blah, blah);
        something_that_might_throw();
        return S_OK;
    }
    CATCH_RETURN();

    [[ ... ]]
};

// Manually implemented
struct MyPage : IPage
{
    MyPage() : m_refCount(1)
    {
    }
```

```
    // You can call this method anything you want.
    void Phase2Initialize()
    {
        Application::LoadComponent(this, blah, blah);
        something_that_might_throw();
    }

    LONG m_refCount;
    ⟦ ... ⟧
};

CComPtr<IPage> page;
page.Attach(new MyPage());
page->Phase2Initialize();
```

“What about ATL?”

Yeah, what about ATL?

ATL gets its own discussion next time.

Bonus chatter: This problem doesn’t exist for C++ standard library `shared_ptr` because you cannot obtain a `shared_ptr` to yourself from your constructor. If you call `shared_from_this()` from the constructor, you will get a `std::bad_weak_ptr` exception. That’s because, as we saw some time ago, the weak pointer hiding inside `enable_shared_from_this` is not initialized until after the object is constructed.

[1] This is basically a variant of getting a strong reference from the this pointer too late: We are destructing an object before its reference count goes to zero.

[2] WRL much prefers that you report failure to initialize the class by returning a failure `HRESULT` from `RuntimeClassInitialize()` rather than throwing an exception. WRL does nothing to translate the exception, and if the `MakeAndInitialize()` call came from `DllGetActivation-Factory` or from the autogenerated implementation of `IActivationFactory::Create-Instance()`, the exception will go uncaught and escape the ABI boundary.

WRL does catch exceptions that come from the constructor, but it blindly translates all such exceptions into `E_OUTOFMEMORY`, even if the exception was not `std::bad_alloc`.