Gotcha: C++/WinRT weak_ref.get() doesn't get the weak reference; it gets the strong reference

devblogs.microsoft.com/oldnewthing/20220202-00

February 2, 2022

Raymond Chen

If you have a `winrt::com_ptr<T>`, you can call the `get()` method to obtain the raw COM pointer inside it. This is handy if you need to pass that raw pointer along to another method that wants raw pointers.

The `winrt::weak_ref<T>` also has a `get()` method, but its behavior is different: It tries to resolve the weak reference to a strong reference and returns that strong reference as a `com_ptr<T>`. If it fails to resolve the weak reference to a strong reference, then it returns an empty `com_ptr<T>`. (If `T` is a projected type, then the returned object is a `T` rather than `com_ptr<T>`, since `com_ptr<T>` is redundant.)

I've seen this gotcha bite people who are familiar with `com_ptr`s, but haven't worked much with `weak_ref`. They have a `weak_ref` in their hand and they want to look at the pointer inside it. They'll call `weak_ref.get()` (because that's what works for `com_ptr`), and not only is that not what they want, the result can be downright dangerous.

```
weak_ref<T> saved_weak;

bool IsTheSavedWeakRef(weak_ref<T> const& ref)
{
    return saved_weak.get() == ref.get();
}
```

The idea here is that we want to compare the `IWeakReference` pointers hiding inside the `weak_ref` to see if the weak reference passed in is the same one we had saved earlier. Unfortunately, what actually happens is that we resolve both weak references to strong `T` references, and then see if the strong references match.

The first bug is that this treats all broken weak references as equal, even if they aren't the *same* weak reference. Indeed, they may even have been obtained from different objects entirely!

The second bug is that the temporary materialization of a strong reference means that when the function returns, those strong references are released, and that can result in object destruction at a time you weren't expecting. Suppose we initially have a strong reference to a `T` in a variable called `saved_strong` and a weak reference to the same `T` in a variable called `saved_weak`.

Thread 1	Thread 2	Refcount on `T`
`IsTheSavedWeakRef()`		1
`saved_weak.get()`		2
	`saved_strong.reset()`	1
temporary `com_ptr` destructs		0

Initially, there is one strong reference to the object, held in `saved_strong`. The `IsTheSavedWeakRef()` function tries to promote the `saved_weak` to a strong reference in the form of a `com_ptr` (say), and it succeeds. The number of strong references is now two.

Meanwhile, another thread resets the `saved_strong` strong reference, which would have destructed the object if `IsTheSavedWeakRef()` hadn't created a temporary strong reference. Instead, the reset of `saved_strong` causes the reference count to drop to one.

Back on Thread 1, the `IsTheSavedWeakRef()` function finishes its comparison and destructs the temporary `shared_ptr`, which drops the reference count to zero and destroys the object.

The `IsTheSavedWeakRef()` function is destroying an object!

This can be quite a surprise to the authors of `IsTheSavedWeakRef()`, who thought they were doing completely non-intrusive lightweight operations. In particular, the caller of `IsTheSavedWeakRef()` might be holding a lock, such as one that is designed to protect access to `saved_weak`.

Now you are in the world of destroying an object under a lock, and depending on how those objects are structured, this could be harmless or create a potential deadlock or trigger memory corruption due to unexpected reentrancy.

Bonus chatter: With PR#608, you can now compare weak references directly for equality, so you aren't tempted to try the `get()` trick.

Nevertheless, if you want to get the raw pointer inside a `winrt::weak_ref`, you can use `get_abi`.

Bonus bonus chatter: The WIL library has its own quirk related to COM weak references and COM agile references. The `wil::com_weak_ref` and `wil::com_agile_ref` act like a `wil::com_ptr` in most respects, but the `query` and `copy` methods operate on the underlying object, not the weak reference or agile reference itself.

For weak references, this is normally what you want, since there isn't much you can query from a weak reference. However, if you have a weak reference to a remote object, you may want to query the weak reference for interfaces like `IMarshal` or `IClientSecurity`, and those queries will go not to the weak reference but to the underlying object.

If you want to perform the query against the weak reference itself, you will have to use a Jedi mind trick to make WIL forget that the pointer was ever a weak pointer: Decay it back to the `IUnknown`, and then operate on that.

```
// marshal = saved_weak.query<IMarshal>(); // doesn't work, do this instead
auto marshal = wil::com_query<IMarshal>(static_cast<IUnknown*>(saved_weak.get()));
```

Here, we use the free `com_query` function which does the equivalent of a `com_ptr.query`, but as a free function instead of requiring a `com_ptr` in hand.

(Personally, I think this magic behavior of weak and agile references is a pit of failure. I think the methods should have been called `lock` or `resolve`. Reserve `get` for extracting the raw ABI pointer.)

Raymond Chen

Follow