The dangers of releasing the last strong reference from within its own callback

devblogs.microsoft.com/oldnewthing/20230927-00

September 27, 2023

Raymond Chen

A common callback pattern is that unregistering a callback will wait until any outstanding callbacks have completed. This avoids the problem of freeing the data out from under a running callback.

Thread 1	Thread 2
`RegisterCallback(callback, this);`	
	`callback` begins
Destructor begins	⋮ do stuff with `this`
`UnregisterCallback(callback)`	⋮ do stuff with `this`
Destructor completes	⋮ do stuff with `this`
	⋮ do stuff with `this`
	`callback` returns

In the above example, the `UnregisterCallback` doesn't wait for the outstanding callback to complete, and as a result, the object is destructed while there is code still using it.

Forcing `UnregisterCallback` to wait for outstanding callbacks to complete avoids this problem:

Thread 1	Thread 2
`RegisterCallback(callback, this);`	
	`callback` begins
Destructor begins	⋮ do stuff with `this`

`UnregisterCallback(callback)`	⋮do stuff with `this`
⋮waiting for callback to finish	⋮do stuff with `this`
⋮waiting for callback to finish	⋮do stuff with `this`
⋮waiting for callback to finish	`callback` returns
`UnregisterCallback(callback)` returns	

Destructor completes

This pattern avoids a use-after-free problem, but it creates a new one: Deadlock.

If your callback takes a strong reference to the containing object, then when that strong reference is destructed at the end of the callback, that may end up destructing the last strong pointer to the object, causing the object itself to destruct from inside the callback. That destructor will wait for the callback to complete, and we have deadlocked.

Thread 1	Thread 2	Reference count
Constructor		
⋮`RegisterCallback(callback, this);`		1
Retain strong reference		1
	`callback` begins	
	⋮create strong reference to self	2
	⋮do stuff with `this`	
Release strong reference	⋮do stuff with `this`	1
	⋮do stuff with `this`	
	⋮do stuff with `this`	
	⋮destruct local strong reference	0
	⋮⋮Destructor runs	
	⋮⋮`UnregisterCallback(callback)` hangs	

The call to `UnregisterCallback` hangs because it is waiting for the callback to complete, but the callback is itself waiting for the destructor to finish, and the destructor is stuck in the `UnregisterCallback`.

For thread pool callbacks, we could use `DisassociateCurrentThreadFromCallback` to break the deadlock. But other types of callbacks may not have a way to say, “Act as if I returned (even though I haven’t yet).”

One solution is to forbid taking a strong reference to the containing object from the callback.

```
void MyObject::callback(CallbackData* data, void* context)
{
    auto self = reinterpret_cast<MyObject*>(context);

    // Calling self->get_strong() or self->shared_from_this()
    // is forbidden! Or in fact anything else that might result in
    // a strong reference to MyObject.
    //
    // Also, cannot access any members that are initialized after
    // the RegisterCallback call or cleaned up before the
    // UnregisterCallback call.

    ⟦ do stuff with "self" ⟧
}
```

In addition to forbidding strong references, we must also make sure not to access any members which are initialized in the constructor after the call to `RegisterCallback` or cleaned up in the destructor prior to the `UnregisterCallback` call. In the common case that the callback is managed by an RAII type, this means that you cannot access any members which are declared after the RAII type or manually initialized in the constructor body; nor can you access any members which are cleaned up explicitly in the destructor, or declared after the RAII type. (The “declared after the RAII type” restriction comes from the rule that C++ members are initialized in order of declaration in the class definition, and destructed in reverse order.)

For example, suppose `callback_holder` is an RAII type that calls `UnregisterCallback` at destruction.

```
struct MyObject
{
    std::string m_value1;
    Widget m_widget;
    callback_holder m_callback(&MyObject::callback, this);
    std::string m_value2;

    MyObject()
    {
        m_widget.prime();
    }

    ~MyObject()
    {
        m_widget.disable();
    }

    static void callback(CallbackData* data, void* context);
};
```

Given the above class, the `callback` cannot access `m_value2`, since it is constructed after `m_callback` and destructed before it. It also has to be prepared for running before `m_widget` has been primed or running after `m_widget` has been disabled.

The fact that the callback is still potentially active before the constructor finishes and after the destructor starts means that the usual rule of thumb that “there won’t be any conflicting threads during construction or destruction” does not apply, since there may be an active callback that is racing the constructor or destructor.

You can simplify the rules by explicitly registering the callback at the end of the constructor and unregistering it at the start of the destructor:

```
    MyObject()
    {
        m_widget.prime();
        // Do this last: Don't register for callbacks
        // until all members are ready.
        m_callback.register(&MyObject::callback, this);
    }

    ~MyObject()
    {
        // Do this first: Ensure all outstanding
        // callbacks have completed.
        m_callback.reset();
        m_widget.disable();
    }
```

Now, that's a lot of rules to remember in the callback. Furthermore, enforcing these rules may be difficult if the "do stuff with `self`" is complex and calls other methods: Those other methods now need to know the rules about forbidden strong references and unsafe member variables, and these are the sorts of rules that are easy to break by accident and hard to detect via static analysis.

Another option is to queue further work and return from the callback immediately. Give that future work a *weak* reference, with the promise not to resolve the weak reference to a strong reference until after you have arrived on the other thread.

```
void MyObject::Callback(CallbackData* data, void* context)
{
    auto self = reinterpret_cast<MyObject*>(context);

    [](auto weak, auto important) -> fire_and_forget {
        co_await resume_background();
        if (auto strong = weak.get()) {
            ⟦ do stuff with "strong" ⟧
        }
    }(get_weak(), data->important);
}
```

The data we carry to the background thread are a *weak* reference to ourselves, as well as any event payload that we need. (We probably can't capture the raw pointer `data` since that would result in using the `data` pointer after the callback returns.) Only after safely arriving on the background thread do we try to promote the weak reference to a strong reference, and we bail out if the object has already begun destruction.

Another solution is to break the deadlock by forcing the destructor to run outside of the callback: In C++/WinRT, you can declare your class with a `final_release` and move the destruction to another thread.

```
struct MyObject : implements<MyObject, IInspectable>
{
    static fire_and_forget final_release(std::unique_ptr<MyObject> ptr)
    {
        // Queue continuation on a background thread
        co_await resume_background();

        // allow ptr to destruct on a background thread
    }
};
```

This is probably the simplest solution, assuming you have it available as an option, since it allows the rest of the class to write code "naturally" and not have to worry about all the weird rules.