C++/WinRT gotcha: get_strong() will produce a broken strong reference if destruction has already begun

devblogs.microsoft.com/oldnewthing/20230928-53

September 28, 2023

Raymond Chen

One of the nice properties of C++ and C++/WinRT weak references is that their expiration coincides with the destruction of the last strong reference. You don't have race conditions where one thread manages to promote a weak reference just after the last strong reference is destroyed.

C++ has `enable_shared_from_this` which lets you obtain a strong reference to the current object. If the current object has started destruction, then it throws `std::bad_weak_ptr`. If you don't want an exception if the object has started destruction, you can do `weak_from_this().lock()`, which produces an empty `shared_ptr` if destruction has already begun.

For C++/WinRT, the corresponding `get_strong()` function does *not* check whether destruction has already begun. This means that if you call `get_strong()` after the object has begun destruction, you will get a `com_ptr` that holds a pointer to an object that is being destructed. You think you have extended its lifetime, but you didn't. The object has already started destructing; you can't "roll back" the destructor. Even worse, after the destruction has completed, the resulting strong reference is now a dangling pointer, and when it destructs, it's going to try to decrement the reference count of already-freed memory. This is the sort of memory corruption bug that keeps you up at night.

Suppose we have this C++/WinRT implementation class:

```
using unique_power_setting_register =
    unique_any<HPOWERNOTIFY,
                  decltype(&::PowerSettingUnregisterNotification),
                            ::PowerSettingUnregisterNotification>;
 wil::
struct Widget : winrt::implements<Widget, ⟦ other interfaces ⟧>
{
    unique_power_setting_register m_notify = subscribe();

    unique_power_setting_register subscribe()
    {
        unique_power_setting_register notify;
        DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS params;
        params.Callback = &Widget::OnNotify;
        params.Context = this;
        THROW_IF_WIN32_ERROR(
            PowerSettingRegisterNotification(
                &GUID_ACDC_POWER_SOURCE,
                DEVICE_NOTIFY_CALLBACK,
                &params,
                &notify));
        return notify;
    }

    static ULONG CALLBACK OnNotify(
        void* context,
        ULONG type,
        void* setting)
    {
        Widget* self = static_cast<Widget*>(context);

        // Code in italics is wrong
        auto strongThis = self->get_strong();

        QueueWorkItem([this, strongThis] {
            ⟦ process the notification ⟧
        });

        return NOERROR;
    }
};
```

This class registers for suspend/resume notifications upon construction, and the `unique_power_setting_register` destructor unregisters at destruction. In the notification callback, we extend the lifetime of the `Widget` so that we can continue processing the notification asynchronously.

Unfortunately, we run into the problem that `get_strong()` doesn't extend the lifetime of an object that has started destruction. Instead, we can take inspiration from `enable_shared_from_this` and keep a `weak_ref` to ourselves.

```
struct Widget : winrt::implements<Widget, ⟦ other interfaces ⟧>
{
    winrt::weak_ref<Widget> m_weakThis = get_weak();
    unique_power_setting_register m_notify = subscribe();

    unique_power_setting_register subscribe()
    {
        unique_power_setting_register notify;
        DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS params;
        params.Callback = &Widget::OnNotify;
        params.Context = this;
        THROW_IF_WIN32_ERROR(
            PowerSettingRegisterNotification(
                &GUID_ACDC_POWER_SOURCE,
                DEVICE_NOTIFY_CALLBACK,
                &params,
                &notify));
        return notify;
    }

    static ULONG CALLBACK OnNotify(
        void* context,
        ULONG type,
        void* setting)
    {
        Widget* self = static_cast<Widget*>(context);

        // WARNING! This code is still wrong!
        if (auto strongThis = self->m_weakThis.get()) {
            QueueWorkItem([this, strongThis] {
                ⟦ process the notification ⟧
            });
        }

        return NOERROR;
    }
};
```

When the callback happens, we try to promote the weak reference to a strong reference. Unlike `get_strong()` which always says "Sure!", the `weak_ref::get()` method will fail if destruction has begun, in which case we just return immediately without doing any work.

There are a number of subtleties in this code.

First is that the `PowerSettingUnregisterNotification` function will wait for outstanding callbacks to complete before returning. This avoids a race condition where the callback is at the open-brace of `OnNotify` at the time the destruction occurs. If `PowerSettingUnregister-Notification` did not wait, then the callback could proceed with an already-destructed `Widget`, and that's not going to end well.

Second is that we declare the `m_weakThis` before we declare the `m_notify`. This ensures that the weak reference remains valid as long as the callback is registered, so that the callback can safely read it. If the members had been declared in the other order, then the weak reference would destruct before we unregister the callback, opening a race window where the callback runs and tries to use a destructed object.

Basically, we reinvented `enable_shared_from_this` for C++/WinRT.

But wait, we still have a bug!

We have the problem that we described last time: Creating a strong reference from inside a callback whose cleanup blocks on the callback. The problem now is that some of the recommended solutions contradict our requirements here! For example, one recommendation is to avoid taking any strong references in the callback. But without a strong reference, how can we queue a work item to do processing later?

One solution is to queue the work item with a weak reference, and resolve the weak reference to a strong one in the work item rather than doing so from the callback.

```
static ULONG CALLBACK OnNotify(
    void* context,
    ULONG type,
    void* setting)
{
    Widget* self = static_cast<Widget*>(context);

    QueueWorkItem([this, weakThis = self->m_weakThis] {
        if (auto strongThis = weakThis.get()) {
            ⟦ process the notification ⟧
        }
    });

    return NOERROR;
}
```

As noted last time, enforcing the “no creation of strong references” rule can be difficult, so the best solution is probably to use `final_release` as described in that article. This lets you write callback code in the natural way with no special constraints.

Next time, we’ll look at some complexity in C++/WinRT that made it difficult to allow class to hold a weak reference to itself.