Reminder: When a C++ object fails to construct, the destructor does not run

devblogs.microsoft.com/oldnewthing/20250120-00

January 20, 2025

In C++, if an object's constructor fails (due to an exception), destructors are run for the object's member variables and base classes, but not for the object itself. The principle at play is that you cannot destruct something that was never constructed in the first place.

Consider this pull request:

```
com_timeout_t(DWORD timeoutInMilliseconds)
    : m_threadId(GetCurrentThreadId())
{
    m_cancelEnablementResult = CoEnableCallCancellation(nullptr);
    err_policy::HResult(m_cancelEnablementResult);
    if (SUCCEEDED(m_cancelEnablementResult))
    {
        m_timer.reset(CreateThreadpoolTimer(
            &com_timeout_t::timer_callback, this, nullptr));
        err_policy::LastErrorIfFalse(
            static_cast<bool>(m_timer));
        if (m_timer)
        {
            FILETIME ft = filetime::get_system_time();
            ft = filetime::add(ft, filetime::
                    convert_msec_to_100ns(timeoutInMilliseconds));
            SetThreadpoolTimer(m_timer.get(), &ft,
                    timeoutInMilliseconds, 0);
        }
    }
}

~com_timeout_t()
{
    m_timer.reset();

    if (SUCCEEDED(m_cancelEnablementResult))
    {
        CoDisableCallCancellation(nullptr);
    }
}

⟦ member variables: ⟧

HRESULT m_cancelEnablementResult{};
DWORD m_threadId{};
bool m_timedOut{};
wil::unique_threadpool_timer_nocancel m_timer;
```

The idea is that the constructor first calls `CoEnableCallCancellation` and reports the failure via `err_policy::HResult()`. In the WIL library, the `err_policy` defines how the caller wants errors to be reported.

	`err_returncode_policy`	`err_exception_policy`	`err_failfast_`
`HResult(hr)` with failure	`return hr;`	Throw an exception	Terminate the process
`LastErrorIfFalse(false)`	`return GetLastError();`	Throw an exception	Terminate the process

When writing code in WIL, you pass each result to the error policy so that it can report the error in the manner the caller requested.

The code saves the result of `CoEnableCallCancellation` in `m_cancelEnablementResult` so that it knows whether it needs to call `CoDisableCallCancellation` at destruction to balance it out.

The tricky case here is if `CoEnableCallCancellation` succeeds, but `CreateThreadpool-Timer` fails. If the error policy is `err_exception_policy`, this throws an exception out of the constructor, which *bypasses the destructor*. This means that the `CoDisableCall-Cancellation` never occurs, and we leak a call cancellation.

If we want to clean up things that were done in the constructor, we have to ask a member variable or base class to clean it up for us.

In this case, one solution is to use the `wil::unique_call` to call a function at destruction.

```
namespace details
{
    inline void CoDisableCallCancellationNull()
    {
        ::CoDisableCallCancellation(nullptr);
    }
} // namespace details

com_timeout_t(DWORD timeoutInMilliseconds)
    : m_threadId(GetCurrentThreadId())
{
    const HRESULT cancelEnablementResult = CoEnableCallCancellation(nullptr);
    err_policy::HResult(cancelEnablementResult);
    if (SUCCEEDED(cancelEnablementResult))
    {
        m_ensureDisable.activate();
        m_timer.reset(CreateThreadpoolTimer(
            &com_timeout_t::timer_callback, this, nullptr));
        err_policy::LastErrorIfFalse(
            static_cast<bool>(m_timer));
        if (m_timer)
        {
            FILETIME ft = filetime::get_system_time();
            ft = filetime::add(ft, filetime::
                    convert_msec_to_100ns(timeoutInMilliseconds));
            SetThreadpoolTimer(m_timer.get(), &ft,
                    timeoutInMilliseconds, 0);
        }
    }
}

// ~com_timeout_t()
// {
//     m_timer.reset();
//
//     if (SUCCEEDED(m_cancelEnablementResult))
//     {
//         CoDisableCallCancellation(nullptr);
//     }
// }

⟦ member variables: ⟧

wil::unique_call<decltype(&details::CoDisableCallCancellationNull),
            details::CoDisableCallCancellationNull, false> m_ensureDisable{};
DWORD m_threadId{};
bool m_timedOut{};
wil::unique_threadpool_timer_nocancel m_timer;
```

The `wil::unique_call` calls the function described by its first two template parameters[1] at destruction. The third template parameter (default `true`) specifies whether the object should be initially active. You can put the object into the active state by calling `activate()`, thereby

enabling the call at destruction.

We want to call `CoDisableCallCancellation` only if the `CoEnableCallCancellation` succeeded, so our `unique_call` is initially inactive.

Now, if there is an exception at construction, the member object `m_ensureDisable` will destruct and call `CoDisableCallCancellation` if necessary.

Bonus chatter: Note that for `err_exception_policy` and `err_failfastpolicy`, a failure to enable call cancellation prevents the constructor from running to completion, which means that the corresponding destructor *always* disables call cancellation. This means that the internal `bool` inside the `unique_call` is always `true`, yet we consume data space for it and code space to check it.

If we wanted to optimize further by avoiding the extra `bool` and the code to test it, we could use a different helper class depending on the error policy.

```
template<typename err_policy>
struct WithCallCancellation
{
    WithCallCancellation()
    {
        err_policy::HResult(CoEnableCallCancellation(nullptr));
    }

    WithCallCancellation(const WithCallCancellation&) :
        WithCallCancellation() { }

    ~WithCallCancellation()
    {
        CoDisableCallCancellation(nullptr);
    }

    constexpr bool active() { return true; }
};

template<>
struct WithCallCancellation<err_returncode_policy>
{
    WithCallCancellation() :
        m_active(SUCCEEDED(
                CoEnableCallCancellation(nullptr))) {}

    WithCallCancellation(const WithCallCancellation&) :
        WithCallCancellation() { }

    ~WithCallCancellation()
    {
        if (m_active) {
            CoDisableCallCancellation(nullptr);
        }
    }

protected:
    bool active() { return m_active; }
    const bool m_active;
}


template<typename err_policy>
com_timeout_t : private WithCallCancellation<err_policy>
{
    com_timeout_t(DWORD timeoutInMilliseconds)
        : m_threadId(GetCurrentThreadId())
    {
        if (this->WithCallCancellation::active())
        {
            m_timer.reset(CreateThreadpoolTimer(
                &com_timeout_t::timer_callback, this, nullptr));
```

```
            ⟦ etc ⟧
        }
    }
}
```

The idea here is to take advantage of the empty base optimization (EBO) so that in the case where the error policy is not `err_returncode_policy`, we don't waste space keeping track of something we know will always be true.[2]

This solution is probably overkill for `com_timeout_t`, which is a class that is expected to have a short lifetime, and certainly not expected to have thousands of instances.

[1] WIL was written when C++11 was the new hotness. If it were written today, we would use `template<auto>` to collapse the two parameters into one.

Bonus bonus chatter: Another option would be to reorder the operations so that `CoEnable-CallCancellation` is done last. That way, if the call fails, the `m_timer`'s destructor will clean up the timer. This is quicker, but it is also more fragile because somebody might add more initialization to the constructor later without realizing that the `CoEnableCallCancellation` must come last because we don't have a member or base class to clean it up for us.

Bonus bonus bonus chatter: Removing the destructor from `com_timeout_t` activates the default move constructor and move assignment operator, but we don't want that because the timer callback has already captured the `this` pointer. As part of removing the constructor, the second PR also explicitly deletes the copy constructor and copy assignment operator (which in turn suppress the move constructor and move assignment operator).

[2] If you can assume C++20, then you can make it a member variable with `[[no_unique_address]]`. But note special treatment for the Microsoft Visual C++ compiler due to the ABI-breaking nature of `[[no_unique_address]]`.