Why do smart pointers null out the wrapped pointer before destroying it?

devblogs.microsoft.com/oldnewthing/20210712-00

July 12, 2021

Raymond Chen

When you null out a smart pointer type, the smart pointer type nulls out the old pointer before releasing it, rather than releasing the member and then setting it to null. (All source code samples has been simplified for expository purposes.)

	Actual	Instead of

std::unique_ptr	`void reset(pointer p = nullptr)`
	`noexcept`
	`{`
	`  pointer old =`
	`std::exchange(m_p, p);`
	`  if (old) m_deleter(old);`
	`}`
		`void reset(pointer p = nullptr)`
		`noexcept`
		`{`
		`  if (m_p) m_deleter(m_p);`
		`  m_p = p;`
		`}`

ATL::CComPtr	`void Release() throw()`
	`{`
	`  T* pTemp = p;`
	`  if (pTemp) {`
	`    p = NULL;`
	`    pTemp->Release();`
	`  }`
	`}`
		`void Release() throw()`
		`{`
		
		`  if (p) {`
		`    p->Release();`
		`    p = NULL;`
		`  }`
		`}`

WRL::ComPtr	`unsigned long InternalRelease()`
	`throw()`
	`{`
	`  unsigned long ref = 0;`
	`  T* temp = ptr_;`
	`  if (temp != nullptr) {`
	`    p = nullptr;`
	`    ref = temp->Release();`
	`  }`
	`  return ref;`
	`}`
		`unsigned long InternalRelease()`
		`throw()`
		`{`
		`  unsigned long ref = 0;`
		
		`  if (p) {`
		`    ref = p->Release();`
		`    p = nullptr;`
		`  }`
		`  return ref;`
		`}`

winrt::com_ptr	`void release_ref() noexcept { if (m_ptr) { std::exchange(m_ptr, {})->Release(); } }`	`void release_ref() noexcept { if (m_ptr) { m_ptr->Release(); m_ptr = nullptr; } }`

Why does the old value get detached from the smart pointer before releasing it? Why not release it, and then set it to null?

One theory is that it's for exception safety, in case an exception occurs in the deleter or `Release` method. But that theory doesn't hold up because the method is marked `noexcept` (or its old-school sort-of equivalent `throw()`).

Another theory is that it enables tail call optimization. While that's true, it's not the primary reason for the order of operations being the way it is.

The reason why the member variable is nulled out before releasing its former value is to avoid reentrancy issues.

If the deleter or the `Release` method leads (through some chain of intermediate operations) to a call on the original smart pointer, we don't want that call to use a pointer that is in mid-destruction.

For example, the smart pointer might be part of a cache. During a pruning pass, the cache entry is determined to be stale and is released. The `Release` method might call back into the cache to unregister itself, and the unregister method will look in its cache to find the matching entry and release it. Congratulations, you just created a double-free bug.

Detaching the wrapped pointer before destructing it avoids this re-entrancy problem.

Bonus chatter: Another way of looking at this is that it is an inlined version of the copy-and-swap idiom. The empty object is swapped in, and then the old object is destructed.

Bonus reading: The proper order of managing reference counts when changing the target of a smart pointer.

Raymond Chen

Follow