On transferring or copying ABI pointers between smart pointers

devblogs.microsoft.com/oldnewthing/20230908-00

September 8, 2023

Raymond Chen

A customer traced a crash to a reference count underflow. But they were using smart pointers. I thought smart pointers avoided reference count problems.

Smart pointers make it easier to avoid reference count problems, but you still have to use them correctly.

There are some basic operations for ABI interop with smart pointers.

- Take ownership: Given a raw pointer, accept the pointer and assume responsibility for releasing it. This typically goes by the name “attach”.
- Share ownership: Given a raw pointer, accept the pointer and share ownership by incrementing the reference count. This doesn’t have a standard name, but I’ll call it “copy” for the purpose of this discussion.
- Non-destructive access: Given a smart pointer, produce the raw pointer without relinquishing ownership. This is typically called “get”.
- Abandon ownership: Given a smart pointer, produce the raw pointer and relinquish ownership. This is typically called “detach”.

It is important that when you move raw pointers between smart pointers, that you match up the two semantics: If you want to transfer ownership, then the donor should use detach semantics and the recipient should use attach semantics. If you want to share ownership, then the donor should use get semantics and the recipient should use copy semantics.

If you mess up, then you end up with reference count bugs: If the donor detaches but the recipient copies, then you have a reference count leak. If the donor offers with “get” but the recipient attaches, then you have an over-release.

Here are some tables showing various Windows smart pointer libraries and how they express the two pairs of operations. Note that this table is just an overview; consult the corresponding documentation for further information. For example, some of the methods require that the smart pointer be non-null.

First, the attach/detach (transfer) operations:

Library	**Detach (donor)**	**Attach (recipient)**

_com_ptr_t	`sp.Detach()`	`sp.Attach(p)`
		`_com_ptr<T>(p, false)`

ATL (CComPtr)	`sp.Detach()`	`sp.Attach(p)`

MFC (IPTR)	`sp.Detach()`
	`/* Note 1 */`
		`sp.Attach(p)`
		`sp.Attach(p, FALSE)`
		`IPTR(T)(p, FALSE)`

WRL	`sp.Detach()`	`sp.Attach(p)`
wil (com_ptr)	`sp.detach()`	`sp.attach(p)`

C++/WinRT (com_ptr)	`sp.detach()`
	`detach_abi(sp)`
		`sp.attach(p)`
		`attach_abi(sp, p)`
		`com_ptr<T>(p, take_ownership_from_abi)`

Note 1: IPTR's `Detach()` method does not return the raw pointer.

And then the get/copy (share) operations:

Library	**Get (donor)**	**Copy (recipient)**

_com_ptr_t	`static_cast<T*>(sp)`
	`sp.GetInterfacePtr()`
		`sp = p`
		`sp.Attach(p, true)`
		`_com_ptr<T>(p, true)`

ATL (CComPtr)	`static_cast<T*>(sp)`
	`*sp`
	`sp.p`
		`sp = p`
		`CComPtr<T>(p)`

MFC (IPTR)	`static_cast<T*>(sp)`
	`sp.GetInterfacePtr()`
		`sp = p`
		`sp.Attach(p, TRUE)`
		`IPTR(T)(p)`
		`IPTR(T)(p, TRUE)`

WRL	`sp.Get()`	`sp = p`
		`ComPtr<T>(p)`

wil (com_ptr)	`sp.get()`	`sp = p`
		`com_ptr<T>(p)`

C++/WinRT (com_ptr)	`sp.get()`
	`get_abi(sp)`
		`sp.copy_from(p)`

Of course, if you are remaining within one row of the table, then you can usually avoid having to operate through ABI pointers. For example, you can just use `sp1 = sp2` to copy one smart pointer to another smart pointer of the same type, or `sp1 = std::move(sp2)` to transfer ownership. The purpose of the above tables is to help you move between rows: The donor and recipient should both be attach/detach (transfer) semantics, or they should both be get/copy (share) semantics.

But wait, there's another option, which I will call the "recipient" pattern.

Library	**Create recipient**	**Transfer to recipient**	**Copy to recipient**
_com_ptr_t	`&sp`	`*r = sp.Detach()`	
ATL (CComPtr)	`&sp` `&sp.p`	`*r = sp.Detach()`	`sp.CopyTo(r)`
MFC (IPTR)	`&sp`		

WRL	`&sp`
	`sp.GetAddressOf()`
	`sp.ReleaseAndGetAddressOf()`
		`*r = sp.Detach()`	`sp.CopyTo(r)`

wil (com_ptr)	`&sp`
	`sp.addressof()`
	`sp.put()`
	`sp.put_void()`
		`*r = sp.detach()`	`sp.query_to(r)`
			`sp.copy_to(r)`

C++/WinRT (com_ptr)	`sp.put()`
	`sp.put_void()`
	`put_abi(sp)`
		`*r = sp.detach()`
		`*r = detach_abi(sp)`
			`sp.copy_to(r)`

The "recipient" pattern produces a `T**`, and it's up to the donor to decide whether to transfer or copy ownership to it. This pattern is used by most of COM: For example, `CreateStream-OnHGlobal` takes a recipient as its final parameter, and it puts a reference to the newly-created stream in that recipient. You as the caller don't know or care whether the function copied or transferred a reference to the stream into your recipient pointer; all that you care about is that when the function returns, your recipient pointer received a reference to the thing.

Bonus chatter: C++ `shared_ptr` and `unique_ptr` have similar patterns and pitfalls. For example, given the declarations `unique_ptr<T> u1, u2;`, you shouldn't write things like `u1.reset(u2.get())` or `std::shared_ptr<int>(u1.get());` since they result in double-ownership and therefore will eventually result in double-destruction.

Bonus reading: We're using a smart pointer, so we can't possibly be the source of the leak.