Additional notes on the various patterns for passing C-style arrays across the Windows Runtime boundary

devblogs.microsoft.com/oldnewthing/20201203-00

December 3, 2020

Raymond Chen

Some time ago, I wrote about the various patterns for passing C-style arrays across the Windows Runtime boundary. A customer had some questions about that table: Who owns the data? Is it passed by value or by copy? Who is responsible for freeing the data?

Let's look at that table again, but in little pieces:

	PassArray	**FillArray**
Allocated by	Caller	Caller
Size	Caller decides	Caller decides
Policy	Read-only	Write-only
IDL	`void M(T[] value);`	`void M(` `  ref T[] value);`

ABI	`HRESULT M(`
	`  UINT32 size,`
	`  _In_reads_(size)`
	`  T* value);`
		`HRESULT M(`
		`  UINT32 size,`
		`  _Out_writes_all_(`
		`  size) T* value);`

In the case of PassArray and FillArray, the memory is allocated by the caller, and a pointer to it is passed to the callee. This a non-owning pointer. The callee can use the pointer for the duration of the function, but not after the function returns. The caller retains ownership of the data, and therefore the caller retains responsibility for freeing the data in whatever manner it sees fit.

If you think about it, there's no choice in the matter for FillArray, since the caller still needs to be able to access the data after the function returns. In theory, you could use ownership transfer semantics for PassArray, but that would just be a pessimization: Why introduce extra copies into a scenario that doesn't need to? Besides, ownership transfer semantics for inbound parameters does not mix with smart pointers.

At the ABI level, the array is passed by reference, since it's a pointer and a length. Of course, if the calling language's semantics are that arrays are passed by value, then for PassArray, the language projection will probably make a copy of the array and pass a reference to the copy. But that's the projection's decision, not the ABI's.

The other two columns are for ReceiveArray:

	ReceiveArray	
	Parameter	**Return value**
Allocated by	Callee	Callee
Size	Callee decides	Callee decides
Policy	Write-only	Write-only
IDL	`void M( out T[] value);`	`T[] M();`
ABI	`HRESULT M( _Out_ UINT32* size, _Outptr_result_buffer_all_( *size) T** value);`	

For ReceiveArray, the result is allocated by the callee. If you think about it, there's no choice in the matter, because only the callee knows the size of the result.

The memory is then passed back to the caller, who assumes responsibility for the memory. If you think about it, there's no choice in the matter here either, because the callee has no opportunity to free the memory after it returns. The memory must be freed with `CoTaskMemFree`, which is the standard allocator for COM methods.

I didn't bother calling out these details because they are direct consequences of COM memory rules. Memory that is allocated by one component and freed by another component must use the COM task allocator: `CoTaskMemAlloc` / `CoTaskMemFree`. If the allocating and freeing all happens on the same side of the function boundary, then that side of the boundary controls the lifetime.

To make things a bit more complete, I retroactively went back and added two more rows to the table:

	PassArray	**FillArray**	**ReceiveArray**	
			Parameter	**Return value**

Allocated by	Caller	Caller	Callee	Callee
Size	Caller decides	Caller decides	Callee decides	Callee decides
Freed by	Caller	Caller	Caller	Caller
Allocator	Caller decides	Caller decides	COM allocator	COM allocator
Policy	Read-only	Write-only	Write-only	Write-only

Raymond Chen

Follow