# The various patterns for passing C-style arrays across the Windows Runtime ABI boundary

devblogs.microsoft.com/oldnewthing/20200205-00

February 5, 2020

Raymond Chen

The Windows Runtime supports C-style arrays. These are contiguous blocks of memory that consist of multiple consecutive instances of the same type. (This is not to be confused with a Windows Runtime *vector*, which is an interface that resembles an array but which does not require any particular storage format.)

Arrays are kind of weird, because they aren't "objects" (there is no identity), but they aren't scalars either (they are variable-sized). And there are multiple patterns for passing these arrays across the ABI boundary, depending on who is allocating the memory, whether the size is known to the caller, and whether the memory is being passed into or out of the function.

- *PassArray*: The caller passes a read-only array, and the implementation reads from it.
- *FillArray*: The caller passes a write-only array, and the implementation fills it with data.
- *ReceiveArray*: The implementation allocates a block of memory for the array and the caller receives a pointer to that block of memory, as well as the number of elements in the array.

Here's a table, since people tend to like tables.

| | PassArray | FillArray | ReceiveArray | |
|---|---|---|---|---|
| | | | **Parameter** | **Retu** |
| **Allocated by** | Caller | Caller | Callee | Calle |
| **Size** | Caller decides | Caller decides | Callee decides | Calle |
| **Freed by** | Caller | Caller | Caller | Calle |
| **Allocator** | Caller decides | Caller decides | COM allocator | COM |

| Policy | Read-only | Write-only | Write-only | Write |
|---|---|---|---|---|
| IDL | `void M(T[] value);` | `void M( ref T[] value);` | `void M( out T[] value);` | `T[]` |
| ABI | `HRESULT M( UINT32 size, _In_reads_(size) T* value);` | `HRESULT M( UINT32 size, _Out_writes_all_( size) T* value);` | `HRESULT M( _Out_ UINT32* size, _Outptr_result_buff( *size) T** value)` | |
| C++/WinRT | `void M( array_view<T const> value);` | `void M( array_view<T> value);` | `void M( com_array<T>& value);` | `com_ M();` |
| C++/CX | `void M( const Array<T>^ value);` | `void M( WriteOnlyArray<T>^ value);` | `void M( Array<T>& value);` | `Arra M();` |
| C# | `void M(T[] value);` | `void M(T[] value);` | `void M( out T[] value);` | `T[]` |
| VB | `Sub M(value As T[])` | `Sub M(value As T[])` | `Sub M(ByRef value As T[])` | `Func As` |
| JS | `function M(value : TypedArray)` | `function M(value : TypedArray)` | `function M() : TypedArray` | `Func : Type` |

I gave the JavaScript prototypes in TypeScript notation so I could annotate the data types. The case of an `out` parameter in JavaScript is a bit more complicated than it looks. I'll save that topic for another day.

Note that in the case of PassArray, the formal parameter at the ABI level is not declared `const T*`, even though the buffer is read-only.

**Update**: "Freed by" and "Allocator" rows added later.

Raymond Chen

**Follow**