The COM marshaller uses the COM task allocator to allocate and free memory

devblogs.microsoft.com/oldnewthing/20090923-00

September 23, 2009

Raymond Chen

It should be second nature to you that the code which allocates memory and the code which frees memory need to use the same allocator. Most of the time, you think of it as “If you allocate memory, you need to free it with the corresponding mechanism,” but this sentence works in the reverse direction as well: If you hand memory to a function that will free it, you have to *allocate* the memory with the corresponding mechanism.

Let’s look at this question that appeared on a discussion group:

> I have the following method defined in my IDL file:
>
> ```
> HRESULT GetSomething([in, string] LPCWSTR pszArg,
>                      [out] DWORD* pcchOut,
>                      [out, size_is(, *pcchOut)] LPWSTR* ppszOut);
> ```
>
> My server implementation of this method goes like this:
>
> ```
> STDMETHODIMP CSomething::GetSomething(
>     LPCWSTR pszArg, DWORD* pcchOut, LPWSTR* ppszOut)
> {
>     HRESULT hr = ...
>     DWORD cch = ...
>     *pcchOut = cch;
>     *ppszOut = new(nothrow) WCHAR[cch];
>     // ... fill in *ppszOut if successful ...
>     return hr;
> }
> ```
>
> When I call this method from a client, the COM server crashes after `CSomething::GetSomething` returns. What am I doing wrong?

The answer should be obvious to you, particularly given the hint in the introductory paragraph, but for some reason, the people on the discussion group got all worked up about how the annotations on the `ppszOut` parameter should have been written, whether `*pcchOut` is a count of bytes or `WCHAR`s, how the marshaller was registered, and nobody even noticed that the allocator didn't match the deallocator.

The rule for COM is that any memory that one module allocates and another module frees must use the COM task allocator. The intent of this rule is to set down one simple, straightforward rule; without it, everybody would have to create their own mechanism for allocating and freeing memory across module boundaries, resulting in the same mishmash that we have in plain Win32, with the global heap, the local heap, the process heap, the C runtime library, or even ad-hoc explicitly paired memory allocation functions like `NetApiBufferAllocate` and `NetApiBufferFree`.

Instead, with COM, it's very simple. If you allocate memory that another COM component will free, then you must use `CoTaskMemAlloc`* and if you free memory that another COM component allocated, then you must use `CoTaskMemFree`.*

In this case, the `CSomething::GetSomething` method is allocating memory that the calling component will eventually free. Therefore, the memory must be allocated with `CoTaskMemAlloc`.*

Nitpicker's corner

*Or a moral equivalent. Note that `SysAllocString` is not a moral equivalent to `CoTaskMemAlloc`.

Remark: MSDN can't seem to make up its mind whether to double the L at the end of "marshal" before adding a suffix, so when searching for information about marshalling, try it both ways.

Raymond Chen

Follow