Allocating and freeing memory across module boundaries

devblogs.microsoft.com/oldnewthing/20060915-04

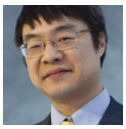
September 15, 2006

Raymond Chen

I'm sure it's been drilled into your head by now that you have to free memory with the same allocator that allocated it. `LocalAlloc` matches `LocalFree`, `GlobalAlloc` matches `GlobalFree`, `new[]` matches `delete[]`. But this rule goes deeper. If you have a function that allocates and returns some data, the caller must know how to free that memory. You have a variety of ways of accomplishing this. One is to state explicitly how the memory should be freed. For example, the `FormatMessage` documentation explicitly states that you should use the `LocalFree` function to free the buffer that is allocated if you pass the `FORMAT_MESSAGE_ALLOCATE_BUFFER` flag. All `BSTR`s must be freed with `SysFreeString`. And all memory returned across COM interface boundaries must be allocated and freed with the COM task allocator. Note, however, that if you decide that a block of memory should be freed with the C runtime, such as with `free`, or with the C++ runtime via `delete` or `delete[]`, you have a new problem: Which runtime? If you choose to link with the static runtime library, then your module has its own private copy of the C/C++ runtime. When your module calls `new` or `malloc`, the memory can only be freed by your module calling `delete` or `free`. If another module calls `delete` or `free`, that will use the C/C++ runtime of **that other module** which is not the same as yours. Indeed, even if you choose to link with the DLL version of the C/C++ runtime library, you still have to agree which version of the C/C++ runtime to use. If your DLL uses `MSVCRT20.DLL` to allocate memory, then anybody who wants to free that memory must also use `MSVCRT20.DLL`. If you're paying close attention, you might spot a looming problem. Requiring all your clients to use a particular version of the C/C++ runtime might seem reasonable if you control all of the clients and are willing to recompile all of them each time the compiler changes. But in real life, people often don't want to take that risk. "If it ain't broke, don't fix it." Switching to a new compiler risks exposing a subtle bug, say, forgetting to declare a variable as volatile or inadvertently relying on temporaries having a particular lifetime. In practice, you may wish to convert only part of your program to a new compiler while leaving old modules alone. (For example, you may want to take advantage of new language features such as templates, which are available only in the new compiler.) But if you do that, then you lose the ability to free memory that was allocated by the old DLL, since that DLL expects you to use `MSVCRT20.DLL`, whereas the new compiler uses `MSVCR71.DLL`. The solution to this

requires planning ahead. One option is to use a fixed external allocator such as `LocalAlloc` or `CoTaskMemAlloc`. These are allocators that are universally available and don't depend on which version of the compiler you're using. Another option is to wrap your preferred allocator inside exported functions that manage the allocation. This is the mechanism used by the `NetApi` family of functions. For example, the `NetGroupEnum` function allocates memory and returns it through the `bufptr` parameter. When the caller is finished with the memory, it frees it with the `NetApiBufferFree` function. In this manner, the memory allocation method is isolated from the caller. Internally, the `NetApi` functions might be using `LocalAlloc` or `HeapAllocate` or possibly even `new` and `free`. It doesn't matter; as long as `NetApiBufferFree` frees the memory with the same allocator that `NetGroupEnum` used to allocate the memory in the first place.

Although I personally prefer using a fixed external allocator, many people find it more convenient to use the wrapper technique. That way, they can use their favorite allocator throughout their module. Either way works. The point is that when memory leaves your DLL, the code you gave the memory to must know how to free it, even if it's using a different compiler from the one that was used to build your DLL.



Raymond Chen

Follow