Self-esteem gone overboard: The perils of a global namespace

devblogs.microsoft.com/oldnewthing/20081107-00

November 7, 2008

Raymond Chen

There are items with overly generic names. `HANDLE` , `CloseHandle` , `GetObject` , `DIFFERENCE` , `query.exe` . But their functionality doesn't live up to their name. `HANDLE` refers only to kernel handles, `CloseHandle` can only close kernel handles, `GetObject` only gets information about GDI objects, `DIFFERENCE` applies only to the numerical difference between group resources and standalone resources, and `query.exe` only queries information about ~~Terminal Services~~ Remote Desktop Services.

Why do functions that operate only inside a specific realm have names that suggest a broader scope?

Self-esteem gone bad.

You're on the kernel team. You have a handle manager. What should you call your handles? Well, since they're handles, just call them `HANDLE` . Because that's what they are, right? And naturally the function that closes `HANDLE` s should be called `CloseHandle` . Sure, there are other types of handles out there, but they don't exist in your world. Your world is the kernel, and in the kernel world, you can call them `HANDLE` s and everybody will know that you're talking about kernel handles because that's why you're in the kernel in the first place! Why would somebody pass a handle to a non-kernel object to a kernel function? That makes no sense!

Similarly, the GDI folks came up with their own object system, and naturally the way you get information about an object is to call `GetObject` . There's no confusion here, right? I mean, this is GDI, after all. What other types of objects are there?

The ~~Terminal Services~~ Remote Desktop Services folks thought the same thing when they created their `query.exe` program. Hey, this is a computer set up to run Remote Desktop Services; of course you want to query information about Remote Desktop Services.

Of course, when your symbol exists in a shared namespace, the context of your naming decision becomes lost, and your generic-sounding function name (which worked just great for generic operations *in the world in which it was created*) ends up carrying more meaning than you originally intended.

Commenter Sean W. tries to explains that Unix doesn't have this problem. "A Unix-flavored close() system call can close any file descriptor." This explanation ends up being its own counter-argument. When you say that it can close any file descriptor, you're admitting that it can't close *anything*. You can't use `close()` to close the objects opened by `opendir()` or `dbm_open()` or `XtOpenDisplay`.

"Well, yeah, but it can close any file descriptor regardless of where it came from." And `CloseHandle` works the same way: It can close any kernel handle regardless of where it came from.

Sean W. later clarified that "the scope of `close()` is the system kernel, so it's reasonable to expect that it applies to kernel data and no other data, whereas the scope of `CloseHandle` is all of Win32, including at least KERNEL/USER/GDI/ADVAPI." Um, actually, the scope of `CloseHandle` is also the kernel.

And in the category of "suggesting things that are already done" goes this comment from Daniel, who suggests that the documentation explain which `HANDLE`s can be closed by `CloseHandle`. Actually, if you look at each function that creates a handle, it also tells you the function to use to close it. Not quite the same thing, but since you have to open something in order to close it, you'll find the information even sooner.

Raymond Chen

Follow