What should you do if somebody passes a null pointer for a parameter that should never be null? What if it's a Windows Runtime class?

devblogs.microsoft.com/oldnewthing/20190613-00

June 13, 2019

Raymond Chen

If you have a function for which a parameter may not be null, what should you do if somebody passes null anyway?

There are multiple layers to this question, depending on the technology you are using, so let's start small and work our way up.

If the function runs in the same process as the caller, then you can just crash. No security boundary was crossed. The caller has a logic error where they thought something was non-null, but it ended up being null, and there's no real recovery from a logic error. Dereferencing the null pointer in the normal course of business will result in an access violation, and that will crash the caller's process (which happens to be the same as the process your function is in).

If possible, crash early in the function, so that the reason is more clear. Put a default value into the output parameter, for example. This convention is fairly common for COM methods, because output pointers are generally expected to contain *something* on exit, even if the function as a whole fails. (This rule is important in the case where the function call has been marshaled, because the result of the function call needs to be marshaled back to the caller, and if you put garbage in the output parameter, the marshaler will crash trying to copy the results back to the caller.)

If the function runs in a separate process from the caller, then you need to protect the integrity of your process. On Windows, the standard mechanisms for inter-process function calls are COM or RPC (the layer beneath COM). In those cases, the function returns an `HRESULT`s, and it is common to report `E_POINTER` to say, "You passed a null pointer when a null pointer isn't allowed."

But wait, there's more. If you are indeed using COM or RPC for your inter-process function calls, then the RPC marshaling layer will check for null pointers so you don't have to! In your interface definition (IDL) file, you annotate pointer parameters to say whether a null pointer

is allowed. If you write `[ref]`, then a null pointer is not allowed, but if you write `[unique]`, then a null pointer is permitted. If you say `[in]` or `[out]` without a modifier, then the modifier defaults to the `pointer_default` for the enclosing class. And if there is no `pointer_default` declaration, then the default default is `ref`.

Once you've annotated your pointers, the RPC infrastructure does the parameter validation for you. If somebody passes a null pointer for a parameter that is annotated as `[ref]`, then RPC fails the call immediately with the error `RPC_X_NULL_REF_POINTER`, and the call never reaches your implementation. Of course, if your function was called directly from within the process, it won't go through the RPC layer, and an invalid pointer can get through.

If you put the cases of in-process and out-of-process callers together, you see that the conclusion is "Go ahead and dereference those pointers." If the caller is in-process, then it's okay to crash because you are crashing the caller's process (which happens to be the same process that you are in). If the caller is out-of-process, then the RPC layer will prevent invalid null pointers from getting through.

There's an additional wrinkle to this general principle, however, for the case where you are implementing a Windows Runtime class. Windows Runtime objects are primarily consumed through *projection*, which is the mechanism by which the ugly low-level infrastructure is exposed to higher-level languages in a way that makes more sense for each language. For example, the low-level `HSTRING` is exposed to C# and Visual Basic as a `System.String`, to JavaScript as `String`, to C++/CX as a `String^`, and to C++/WinRT as a `winrt::hstring`.

In the case where you are implementing a Windows Runtime class, and somebody passes a null pointer for an input parameter, then instead of crashing, you should return a COM error code, traditionally `E_POINTER`. This error code will be transformed by the projection into a language-specific exception.

It's better to convert the invalid null pointer to a language-specific exception because that integrates better with language debugging tools. The debugger will see a language exception and give the developer a chance to inspect the exception to see what went wrong. If you had dereferenced the null pointer in native code, the C# debugger (for example) will report that an exception occurred in unmanaged code, and there is unlikely to be a meaningful stack trace because the exception was generated far away from anything the C# debugger can see.

This principle applies only to input parameters. You can freely dereference output parameters because the projections will always pass a valid output pointer. (The C# developer didn't pass an output pointer explicitly. The C# developer merely called the method, and your `[out, retval]` pointer was created by the projection.)

You might have observed that the consequences for passing an invalid null parameter vary depending on whether the method call is marshaled or not. If marshaled, then the result is `RPC_X_NULL_REF_POINTER`, but if not marshaled, then the result is `E_POINTER`. While this seems strange, it's also inconsequential, because any sort of exception from a Windows Runtime method is considered fatal. The process is crashing either way, and the developer studying the crash dumps will know that both `RPC_X_NULL_REF_POINTER` and `E_POINTER` mean "You passed a null pointer when you shouldn't have."

Related viewing: De-fragmenting C++: Making exceptions more affordable and usable, in particular, the part where Herb Sutter talks about the difference between errors and bugs.

Raymond Chen

Follow