Basic ground rules for programming – function parameters and how they are used

devblogs.microsoft.com/oldnewthing/20060320-13

March 20, 2006

Raymond Chen

There are some basic ground rules that apply to all system programming, so obvious that most documentation does not bother explaining them because these rules should have been internalized by practitioners of the art to the point where they need not be expressed. In the same way that when plotting driving directions you wouldn't even consider taking a shortcut through somebody's backyard or going the wrong way down a one-way street, and in the same way that an experienced chess player doesn't even consider illegal moves when deciding what to do next, an experienced programmer doesn't even consider violating the following basic rules without explicit permission in the documentation to the contrary:

- Everything not defined is undefined. This may be a tautology, but it is a useful one. Many of the rules below are just special cases of this rule.
- All parameters must be valid. The contract for a function applies only when the caller adheres to the conditions, and one of the conditions is that the parameters are actually what they claim to be. This is a special case of the "everything not defined is undefined" rule.
    - Pointers are not `NULL` unless explicitly permitted otherwise.
    - Pointers actually point to what they purport to point to. If a function accepts a pointer to a `CRITICAL_SECTION`, then you really have to pass pointer to a valid `CRITICAL_SECTION`.
    - Pointers are properly aligned. Pointer alignment is a fundamental architectural requirement, yet something many people overlook having been pampered by a processor architecture that is very forgiving of alignment errors.
    - The caller has the right to use the memory being pointed to. This means no pointers to memory that has been freed or memory that the caller does not have control over.
    - All buffers are valid to the size declared or implied. If you pass a pointer to a buffer and say that it is ten bytes in length, then the buffer really needs to be ten bytes in length.
    - Handles refer to valid objects that have not been destroyed. If a function wants a window handle, then you really have to pass a valid window handle.

- All parameters are stable.
  - You cannot change a parameter while the function call is in progress.
  - If you pass a pointer, the pointed-to memory will not be modified by another thread for the duration of the call.
  - You can't free the pointed-to memory either.
- The correct number of parameters is passed with the correct calling convention. This is another special case of the "everything not defined is undefined" rule.
  - Thank goodness modern compilers refuse to pass the wrong number of parameters, though you'd be surprised how many people manage to sneak the wrong number of parameters past the compiler anyway, usually by devious casting.
  - When invoking a method on an object, the `this` parameter is the object. Again, this is something modern compilers handle automatically, though people using COM from C (and yes they exist) have to pass the `this` parameter manually, and occasionally they mess up.
- Function parameter lifetime.
  - The called function can use the parameters during the execution of the function.
  - The called function cannot use the parameters once the function has returned. Of course, if the caller and the callee have agreed on a means of extending the lifetime, then those rules apply.
    - The lifetime of a parameter that is a pointer to a COM object can be extended by the use of the `IUnknown::AddRef` method.
    - Many functions are passed parameters with the express intent that they be used after the function returns. It is then the caller's responsibility to ensure that the lifetime of the parameter is at least as long as the function needs it. For example, if you register a callback function, then the callback function needs to be valid until you deregister the callback function.
- Input buffers.

  A function is permitted to read from the full extent of the buffer provided by the caller, even if not all of the buffer is required to determine the result.

- Output buffers.
  - An output buffer cannot overlap an input buffer or another output buffer.
  - A function is permitted to write to the full extent of the buffer provided by the caller, even if not all of the buffer is required to hold the result.
  - If a function needs only part of a buffer to hold the result of a function call, the contents of the unused portion of the buffer are undefined.
  - If a function fails and the documentation does not specify the buffer contents on failure, then the contents of the output buffer are undefined. This is a special case of the "everything not defined is undefined" rule.
  - Note that COM imposes its own rules on output buffers. COM requires that all output buffers be in a marshallable state even on failure. For objects that require nontrivial marshalling (interface pointers and BSTRs being the most common examples), this means that the output pointer must be `NULL` on failure.

(Remember, every statement here is a basic ground rule, not an absolute inescapable fact. Assume every sentence here is prefaced with "In the absence of indications to the contrary". If the caller and callee have agreed on an exception to the rule, then that exception applies. For example, a pointer is prototyped as `volatile` is explicitly marked as "This value can change from another thread," so the rule against modifying function parameters does not apply to such a pointer.) Coming up with this was hard, in the same way it's hard to come up with a list of illegal chess moves. The rules are so automatic that they aren't really rules so much as things that simply are and it would be crazy even to consider otherwise. As a result, I'm sure there are other "rules so obvious they need not be said" that are missing. (For example, "You cannot terminate a thread while it is inside somebody else's function.")

One handy rule of thumb for what you can do to a function call is to ask, "How would I like it if somebody did that to me?" (This is a special case of the "Imagine if this were possible" test.)

Raymond Chen

Follow