The C language specification describes an abstract computer, not a real one

devblogs.microsoft.com/oldnewthing/20130328-00

March 28, 2013

Raymond Chen

If a null pointer is zero, how do you access the memory whose address is zero? And if C allows you to take the address one past the end of an array, how do you make an array that ends at `0xFFFFFFFF`, since adding one to that value would wrap around?

First of all, who says that there is a byte zero? Or a byte `0xFFFFFFFF`?

The C language does not describe an actual computer. It describes a theoretical one. On this theoretical computer, it must be possible to do certain things, like generate the address of one item past the end of an array, and that address must compare greater than the address of any member of the array.

But how the C language implementation chooses to map these theoretical operations to actual operations is at the discretion of the C language implementation.

Now, most implementations will do the "obvious" thing and say, "Well, a pointer is represented as a numerical value which is equal to the low-level memory address." But they are not required to do so. For example, you might have an implementation that says, "You know what? I'm just going to mess with you, and every pointer is represented as a numerical value which is equal to the low-level memory address *minus 4194304*. In other words, if you try to dereference a pointer whose numeric value is 4096, you actually access the memory at 4194304 + 4096 = 4198400. On such a system, you could have an array that goes all the way to `0xFFFFFFFF`, because the numeric value of the pointer to that address is `0xFFBFFFFF`, and the pointer to one past the end of the array is therefore a perfectly happy `0xFFC00000`.

Before you scoff and say "That's a stupid example because nobody would actually do that," think again. Win32s did exactly this. (The 4194304-byte offset was done in hardware by manipulating the base address of the flat selectors.) This technique was important because byte 0 was the start of the MS-DOS interrupt table, and corrupting that memory was a sure way to mess up your system pretty bad. By shifting all the pointers, it meant that a Win32s

program which dereferenced a null pointer ended up accessing byte 4194304 rather than byte 0, and Win32s made sure that there was no memory mapped there, so that the program took an access violation rather than corrupting your system.

But let's set aside implementations which play games with pointer representations and limit ourselves to implementations which map pointers to memory addresses directly.

"A 32-bit processor allegedly can access up to 2^{32} memory locations. But if zero and `0xFFFFFFFF` can't be used, then shouldn't we say that a 32-bit processor can access only $2^{32} - 2$ memory locations? Is everybody getting ripped off by two bytes? (And if so, then who is pocketing all those lost bytes?)"

A 32-bit processor can address 2^{32} memory locations. There are no "off-limits" addresses from the processor's point of view. The guy that made addresses zero and `0xFFFFFFFF` off-limits was the C language specification, not the processor. That a language fails to expose the full capabilities of the underlying processor shouldn't be a surprise. For example, you probably would have difficulty accessing the byte at `0xFFFFFFFF` from JavaScript.

There is no rule in the C language specification that the language must permit you to access any byte of memory in the computer. Implementations typically leave certain portions of the address space intentionally unused so that they have wiggle room to do the things the C language specification requires them to do. For example, the implementation can arrange never to allocate an object at address zero, so that it can conform to the requirement that the address of an object never compares equal to the null pointer. It also can arrange never to allocate an object that goes all the way to `0xFFFFFFFF`, so that it can safely generate a pointer one past the end of the object which behaves as required with respect to comparison.

So you're not getting ripped off. Those bytes are still addressable in general. But you cannot get to them in C without leaving the C abstract machine.

A related assertion turns this argument around. "It is impossible to write a conforming C compiler for MS-DOS because the C language demands that the address of a valid object cannot be zero, but in MS-DOS, the interrupt table has address zero."

There is a step missing from this logical argument: It assumes that the interrupt table is a C object. But there is no requirement that the C language provide access to the interrupt table. (Indeed, there is no mention of the interrupt table anywhere in the C language specification.) All a conforming implementation needs to do is say, "The interrupt table is not part of the standard-conforming portion of this implementation."

"Aha, so you admit that a conforming implementation cannot provide access to the interrupt table."

Well, certainly a conforming implementation can provide language extensions which permit access to the interrupt table. It may even decide that dereferencing a null pointer grants you access to the interrupt table. This is permitted because dereferencing a null pointer invokes *undefined behavior*, and one legal interpretation of undefined behavior is "grants access to the interrupt table."

Raymond Chen

Follow