How to check if a pointer is in a range of memory

devblogs.microsoft.com/oldnewthing/20170927-00

September 27, 2017

Raymond Chen

Suppose you have a range of memory described by two variables, say,

```
byte* regionStart;
size_t regionSize;
```

And suppose you want to check whether a pointers lies within that region. You might be tempted to write

```
if (p >= regionStart && p < regionStart + regionSize)
```

but is this actually guaranteed according to the standard?

The relevant portion of the C standard (6.5.8 Relational Operators)[1] says

> If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression P points to an element of an array object and the expression Q points to the last element of the same array object, the pointer expression Q+1 compares greater than P. In all other cases, the behavior is undefined.

Now remember that the C language was defined to cover a large range of computer architectures, including many which would be considered museum relics today. It therefore takes a very conservative view of what is permitted, so that it remains possible to write C programs for those ancient systems. (Which weren't quite so ancient at the time.)

Bearing that in mind, it is still possible for an allocation to generate a pointer that satisfies the condition despite the pointer not pointing into the region. This will happen, for example, on an 80286 in protected mode, which is used by Windows 3.x in Standard mode and OS/2 1.x.

In this system, pointers are 32-bit values, split into two 16-bit parts, traditionally written as `XXXX:YYYY`. The first 16-bit part (`XXXX`) is the "selector", which chooses a bank of 64KB. The second 16-bit part (`YYYY`) is the "offset", which chooses a byte within that 64KB bank. (It's more complicated than this, but let's just leave it at that for the purpose of this discussion.)

Memory blocks larger than 64KB are broken up into 64KB chunks. To move from one chunk to the next, you add 8 to the selector. For example, the byte after `0101:FFFF` is `0109:0000`.

But why do you add 8 to move to the next selector? Why not just increment the selector? Because the bottom three bits of the selector are used for other things. In particular, the bottom bit of the selector is used to choose the selector table. Let's ignore bits 1 and 2 since they are not relevant to the discussion. Assume for convenience that they are always zero.[2]

There are two tables which describe how selectors correspond to physical memory, the Global Descriptor Table (for memory shared across all processes) and the Local Descriptor Table (for memory private to a single process). Therefore, the selectors available for process private memory are `0001`, `0009`, `0011`, `0019`, *etc*. Meanwhile, the selectors available for global memory are `0008`, `0010`, `0018`, `0020`, *etc*. (Selector `0000` is reserved.)

Okay, now we can set up our counter-example. Suppose `regionStart = 0101:0000` and `regionSize = 0x00020000`. This means that the guarded addresses are `0101:0000` through `0101:FFFF` and `0109:0000` through `0109:FFFF`. Furthermore, `regionStart + regionSize = 0111:0000`.

Meanwhile, suppose there is some global memory that happens to be allocated at `0108:0000`. This is a global memory allocation because the selector is an even number.

Observe that the global memory allocation is not part of the guarded region, but its pointer value does satisfy the numeric inequality `0101:0000` ≤ `0108:0000` < `0111:0000`.

Bonus chatter: Even on CPU architectures with a flat memory model, the test can fail. Modern compilers take advantage of undefined behavior and optimize accordingly. If they see a relational comparison between pointers, they are permitted to assume that the pointers point into the same aggregate or array (or one past the last element of that array), because any other type of relational comparison is undefined. Specifically, if `regionStart` points to the start of an array or aggregate, then the only pointers that can legally be relationally compared with `regionStart` are the ones of the form `regionStart`, `regionStart + 1`, `regionStart + 2`, …, `regionStart + regionSize`. For all of these pointers, the condition `p >= regionStart` is true and can therefore be optimized out, reducing the test to

```
if (p < regionStart + regionSize)
```

which will now be satisfied for pointers that are numerically less than `regionStart`.

(You might run into this scenario if, as in the original question that inspired this answer, you allocated the region with `regionStart = malloc(n)`, or if your region is a “quick access” pool of preallocated items and you want to decide whether you need to `free` the pointer.)

Moral of the story: This code is not safe, not even on flat architectures.

But all is not lost: The pointer-to-integer conversion is implementation-defined, which means that your implementation must document how it works. If your implementation defines the pointer-to-integer conversion as producing the numeric value of the linear address of the object referenced by the pointer, and you know that you are on a flat architecture, then what you can do is compare *integers* rather than *pointers*. Integer comparisons are not constrained in the same way that pointer comparisons are.

```
    if ((uintptr_t)p >= (uintptr_t)regionStart &&
        (uintptr_t)p < (uintptr_t)regionStart + (uintptr_t)regionSize)
```

[1] Note that comparison for equality and inequality are not considered relational comparisons.

[2] I know that in practice they aren’t. I’m assuming they are zero for convenience.

(This article was adapted from my answer on StackOverflow.)

Update: Clarification that the “start of region” optimization is available only when `regionStart` points to the start of an array or aggregate.

Raymond Chen

Follow