What is smallest offset for which I can safely omit overflow checking when I add it to a pointer?

Asked 15 years, 8 months ago Modified 15 years, 8 months ago

Viewed 405 times

0

Can I expect that any "data" pointer in user space programs lies in a safe distance from the addresses 0 and 0xffffffff..., so that I can safely add a small offset to the pointer without checking for an overflow? What is the largest positive n for which I can safely assume that p + n doesn't overflow when p is a char pointer into a constant character buffer or a dynamically allocated string (on a modern >= 32-bit operating system)?

To avoid confusion: I'm talking about *overflow checking, not about bounds checking*. For example: If you have a pointer p to the beginning of a string with m chars and you want to access the char at the positive offset i, then you either need to check that i < m or you can check indirectly p + i < p + m. However, in the latter case you also have to make sure that p + i doesn't overflow, i.e. you have to make sure that p + i >= p.

Update: Ok, p + i is not valid standard C if i > m, regardless of whether p + i is actually dereferenced or whether it overflows. However, the question I'm really interested in is whether there is a small n for which p + n won't overflow *in practice*. Answering this question obviously requires some knowledge about how modern operating systems organize the address space.

Update2: It would already be very interesting to hear about any one particular platform, even if it's not generalizable. Preferably not some obscure embedded one. x86 or Power-based 32-bit Win, Linux and Mac would be most interesting.

c++ c pointers overflow

Share

Improve this question

Follow

edited Apr 5, 2009 at 11:36

asked Apr 5, 2009 at 9:35

Stephan

117 ● 1 ● 7

1 Don't do this! The only safe n is 0. – Anton Tykhyy Apr 5, 2009 at 9:44

Even venturing into platform-specific (and from C++'s point of view, undefined) behavior, there is no single answer. Not every "modern >=32 bit OS" places data in the same address range. Your question doesn't make sense, and that indicates that you're simply going about this the wrong way.
– Stack Overflow is garbage Apr 5, 2009 at 11:30

No, but I'd assume that most/all don't put data directly next to the address limits (0 and 0xffffffff). Or do you have knowledge about any one particular platform where data is put next to the limits? – Stephan Apr 5, 2009 at 11:53

8 Answers

Sorted by: Highest score (default)

8

The only offset you can safely add to the pointer (and then dereference it) is one that would position the pointer in the block of memory you are working with. This block must have been allocated with new or malloc, or exist on the stack.

Either way, it is guaranteed that the memory exists (and that one past the end of the memory is a sensible address), otherwise you would have got an exception from new, a NULL pointer from malloc or undefined behaviour if you tried to mis-allocate on the stack. In no case do you have to check for overflow.

Share Improve this answer Follow

edited Apr 5, 2009 at 9:50

answered Apr 5, 2009 at 9:42

anon

Please see the updated question. I wasn't talking about bounds checking, but about overflow checking. And yes, sometimes you do have to check for overflow. – Stephan Apr 5, 2009 at 10:01

You appear to bea sking for a way of checking that any pointer (because p + i could have any value) is valid - there is no way of doing this. However, this is not an issue in properly written code, – anon Apr 5, 2009 at 10:07

No, I'm not. I'm trying to optimize the bounds checking code in a performance critical library function. – Stephan Apr 5, 2009 at 10:09

That doesn't explain why you don't just use i < m – anon Apr 5, 2009 at 10:20

1 @Stephan: Need that in the question... and would mean i < 2^10 (in case of /largeaddressaware). And would not be true if 32bit process on 64bit Windows. – Richard Apr 5, 2009 at 12:16

4

Strictly speaking the answer is 0 because `p` could already be pointing to one past the end of the array and that is what the standard says is valid.

In a specific implementation you might be able to get away with some amount but that is completely implementation defined. There has has been hardware, and maybe is still, that checks operations on pointers within the CPU instructions: if `p` points to an array of 2 ints, doing `p+3` will cause the CPU to fail the instruction. On the other hand, on most current hardware you can get away with a lot.

Share Improve this answer Follow

answered Apr 5, 2009 at 9:42

Richard
109k ● 21 ● 210 ● 272

3

Given the information you've provided, the answer is 0. 0 is the only answer that is valid **according to the C++ standard**.

If you are willing to venture into undefined behavior (hint: don't), you'll have to provide us some platform-specific information **and** wave goodbye to **any** guarantees about the validity of the state of your application. Your application may still run, but you're relying on the arbitrary, and possibly changing, decisions made by the OS and compiler writers.

If we know the exact details of your platform (which CPU, OS, which compiler, primarily), then it may be possible to give you an answer that will *usually* work, as long as nothing changes in either compiler or OS.

But it seems like you're going about this the wrong way. IF this is so performance-critical like you keep saying, *arrange it so that pointer overflows are not an issue*.

Strictly speaking, adding *anything* to a pointer that makes it point past the same block of memory it pointed at before, is undefined. It may work, or it may act really really funky on some architectures. It may overflow at unexpected times, it may cause hard crashes on some.

That is why the language simply says "it's not allowed", and why we can't say what'd happen in practice when you haven't told us the platform it's running on.

A C++ pointer *is not a memory address*. That is how compilers typically implement it, yes, but they obey different rules. Some things are legal with memory addresses, according to the CPU instruction set, which are not legal with pointers.

But seriously, my best advice: Take a step back and examine how you can avoid *needing* to check for overflows.

Share Improve this answer

Follow

answered Apr 5, 2009 at 11:23

Stack Overflow is garbage

247k ● 52 ● 353 ● 555

2

In C, pointer arithmetic and relative comparisons are defined only within "objects" (not to be confused with C++ objects). An "object" is a variable (of any type) or a memory area allocated with malloc/calloc/realloc. You can compute a pointer to "one past" the object, and this will always work in a standard conforming implementation.

Looking at things at a lower level, a pointer is typically implemented as an (unsigned) integer. The size of the integer is large enough to hold the address of any memory location. The only way for a pointer to overflow is if you exceed the address space, and that is impossible to do while conforming to the C standard.

However, if you are writing low-level code, and ignoring the restriction that pointer arithmetic is only valid within an object, the best way to do this is to take advantage of the knowledge of how pointers are represented. In most modern environments, this then becomes the same thing as checking for overflow with unsigned integer arithmetic.

(Exceptions to that will be things like segmented memory architectures, of the 8086 or Multics kinds, and probably other things I may have suppressed from my memory to preserve my sanity.)

Share Improve this answer Follow

answered Apr 5, 2009 at 10:21

user25148

1

Of the top of my head, it seems like this is very dependent on:

- the OS
- the underlying HW
- the compiler

As a rule, and only as far as I remember, the stack is allocated way at the top of the linear address space. Considering you have all sorts of runtime libraries running, your actual data most likely isn't running at the top of that space. Either way, if you exceed the area allocated by `malloc` you'll run into other trouble, and if you exceed your stack frame, you'll also run into trouble. The bottom line is that I don't think you have to worry about wraparound from 0xffffffffffffffff to 0x0, but you still have to ensure you don't go over the bounds of either your static, automatic or manually allocated memory.

Share Improve this answer

Follow

answered Apr 5, 2009 at 9:46

Nathan Fellman

127k ● 104 ● 264 ● 326

0

This is dependent on two things

- The architecture
- The compiler

So the only way to be certain is to consult the reference documentation for your compiler and arch.

Share Improve this answer

Follow

answered Apr 5, 2009 at 9:43

Schildmeijer

20.9k ● 12 ● 64 ● 79

0

Usually memory allocation should be done in blocks, for example, even if you want to utilize 1 byte, minimum allocation should always be in power of 2, for example, CString in MFC uses 128 bytes, or 64 bytes, by doing this way you can waste some space for sure but also reduce amount of calculation. If your allocation was done on the basis of blocks, then you can use size of block and current pointer value to use maximum number of offset to avoid overflows.

Share Improve this answer Follow

answered Apr 5, 2009 at 9:51

Akash Kava
39.9k ● 20 ● 124 ● 171

0

For address overflow, a useful trick as I know used for time camparsion should also works for your situation.

If you platform is 32bits, then each address is 32bits wide with (unsigned long type), you can try following macro:

> #define address_after(a,b) ((long)(b) - (long)(a) < 0))
>
> #define address_before(a,b) address_after(b,a)

Then you can compare address as address_after(p+i, p+m) safely only if the |m-i| < 0x7FFFFFFF (which is common situation). This macro handles the overflow problem very well.

Share Improve this answer Follow

edited Apr 5, 2009 at 17:06

answered Apr 5, 2009 at 16:59

Sam Liao
45.9k ● 15 ● 54 ● 65