

Re: Is C99 the final C? (some suggestions)

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2003-12/1765.html

From: Sidney Cadot (*sidney_at_jigsaw.nl*)

Date: 12/11/03

Date: Thu, 11 Dec 2003 22:13:57 +0100

Paul Hsieh wrote:

(I'm snipping superfluous bits, I propose we concentrate on the important things and leave some other things aside, to keep this manageable...)

> *Sidney Cadot wrote:*

>

>> *Paul Hsieh wrote:*

>> [snip]

>>

>> *I think there are relatively few excuses for any professional C programmer not to have access to the standard (and reading it once or twice wouldn't hurt either). It's only 18 bucks in electronic form, and you don't even have to go out to get it.*

>

> *Care to take a quick survey of comp.lang.c patrons who own their own copy of the standard?*

Sure, I think you'd be surprised.

What is your excuse for not having the standard? A free draft (N869) is also available.

>>> *My objection to your proposal can simply be restated as "all machines have limited resources", deal with it.*

>>

>> *Then the standard would have to specify what it means exactly by a "resource". Standards are no places for vague terminology.*

>

>

> *You mean vague terminologies like "stack"?*

No, I meant "resource".

There doesn't have to be any mention of "stack" in what I try to propose. Obviously, active function invocations consume memory – that's what I would like to see formalized.

comp.lang.c: Re: Is C99 the final C? (some suggestions)

>>>I had to look this one up -- that's one of the classic useless
>>>non-solutions (especially if you want to call malloc more than 4
>>>billion times, for example), which explains why I've never
>>>committed it to memory.
>>
>>We have long longs nowadays. 2^32 is not an important limit.
>
>
> And long long's are portable?!?! Are you even following your own
> train of thought?

You really seem a bit out of touch with current events... Since you don't seem to have access to the C99 standard, here's the relevant quote:

6.2.5 clause #4:

"There are five standard signed integer types, designated as signed char, short int, int, long int, and long long int. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types."

(somewhat further down, the unsigned long long int is defined).

Both must be able to represent at least 64 bit values ($-2^{63}..2^{63}-1$, $0..2^{64}-1$, respectively).

>>>You still need to be able to identify process instances, which *BY
>>>ITSELF* is not portable. Process instances identifiers would
>>>count as something *BEYOND* just malloc/free/etc.
>>
>>Why do you need to distinguish process instances?
>
> Reread the Lamport's bakery algorithm. One of the crucial steps is
> "for each process, check their number, and pick the max, then add 1".
> Another is "for each process that has a lower number, wait in line
> behind them." This means you have a way of knowing how many processes
> there are, and how to index them. This means you have to hook out or
> wrap whatever "spawn" mechanism you have in your language (or use some
> other platform specific mechanism to iterate through all running
> threads.)

Could you provide a reference to the description of the algorithm you're using? I'm not saying it's incorrect or anything, it's just that I'd like to respond in the same terminology.

>>[...] In your API, each call carries the heap it refers to as a
>>parameter.
>
>

Re: Is C99 the final C? (some suggestions)

comp.lang.c: Re: Is C99 the final C? (some suggestions)

- > *Yes, but if two threads simultaneously call heap functions on the same*
- > *heap (heaps are not restricted to being distinct across threads) then*
- > *there is a race condition.*

So we need to put a mutex around the critical section, no big deal. All hinges now on whether the Lamport algorithm needs process id's. Fortunately, there's no subjective element in that, we should be able to sort that out.

>>>>*I would say that writing a more powerful heap manager is possible in principle, even in a multithreaded environment, as long as you have a well-implemented malloc() (and friends) at your disposal.*

>>>>

>>>>**BZZZT!!!* Wrong.*

>>>>

>>>>

>>>>*It would be a drag to do so and it would be slow, but it is possible.*

>>>>

>>>>*Impossible. Slow, fast or otherwise. Its just impossible.*

>>>>

>>>>*If you can explain to me why you need to explicitly distinguish threads*

>>>>*in the memory manager, I'll concede the point.*

- > *Go look up the source code for *ANY* multithreading supporting*
- > *malloc/free. I absolutely guarantee that every single one of them*
- > *uses some kind of platform specific exclusion object to make sure that*
- > *two simultaneous calls to the heap do not collide.*

Sure. Mutexes would suffice and I contend the bakery algorithm can provide them. I feel rather silly getting dragged in this line of discussion (multithreading has little to do with the C execution model) but we'll see where this leads us.

- > *One way or another the heap is some kind of quasi-linked list of memory*
- > *entries. You should have learned from your OS classes (assuming you passed it*
- > *and/or comprehended the material)*

Could you refrain from the silly ad hominem? I am trying to take you seriously, even though you seem to deny the importance of standards, and even though you are advocating an (in my opinion) confused "operator introduction" feature. Maintaining a basic level of respect helps in any discussion.

>>>>----- *Proposed addition to the standard to add stack overflows:*

>>>>----- *Draft*

>>>>

>>>>*a. For an active function invocation, define an upper bound for the*

>>>>*amount of memory in use by this function as follows (in the*

>>>>*following, denote "Upper Limit on Memory Usage" by "ULMU"):*

>>>>

>>>>*And what if the local/parameters are not stored, or storable on the stack?*

Re: Is C99 the final C? (some suggestions)

comp.lang.c: Re: Is C99 the final C? (some suggestions)

>>
>>*How do you mean? They have to be stored. And I'm presuming they're*
>>*stored on the stack.*
>
>
> *Most microprocessors have these temporary storage elements called*
> *"registers". They are often not part of any stack structure. Many modern*
> *compilers will map local variables and temporaries to these "registers", thus*
> *not requiring stack storage for them.*

Not an issue, as I try to establish an upper bound. The fact that the implementation can do better by using registers is of no concern.

>>>*What if an implementation uses multiple stacks depending on the*
>>>**type* of the parameter?*
>>
>>*The hypothetical implementation [...]*
>
>
> *That the ANSI C standard currently does not have any problem with ...*
>
>
>>*[...] that does this will be out of luck. Not because the proposal*
>>*wouldn't work, mind you, but because the ULMU is a very loose bound.*
>>*But hey, it beats having no bound whatsoever.*
>
>
> *No it doesn't. Dealing with the stack is an area that clearly should be*
> *platform specific. For example, it could be that an embedded system has*
> *fast and slow ram. Certain data types might be best suited to being mapped*
> *into fast RAM, whether they are local or not, but you might have a lot more*
> *slow RAM, so you'd put all the rest of the stuff in there.*

To what does "no it doesn't" refer?

How does your example invalidate the "upper bound" idea?

>>>> – *For every parameter/local variable of basic type there is a*
>>>> *corresponding constant macro of type size_t:*
>>>
>>>*You missed all the implicit temporaries required. Homework*
>>>*exercise: for any fixed number n, create a function which requires*
>>>*at least n implicit temporaries (not explicitly declared) to*
>>>*implement.*
>>
>>*Ah yes! So it seems. An interesting problem. Have to think about that*
>>*for a bit.*
>
> *Why don't you think about the implication of the results of this*
> *exercise with respect to your proposal while you are at it.*

comp.lang.c: Re: Is C99 the final C? (some suggestions)

If no way can be found to address this, we can no longer give an upper bound, and the proposal is dead in the water. However, I'm not convinced it can't be handled. For one thing, your proposal to make macro's available outside the function scope that give the functions stack usage upper bound may come in handy.

>>>And what about the number of `alloca()` calls? `alloca()` is a common
>>>extension which just eats extra space off the stack dynamically --
>>>the big advantage being that you don't have to call `free` or
>>>anything like it, for it to be properly cleaned up. It will be
>>>cleaned up upon the function's return (the simplest implementation
>>>of a "freeall".)

>>

>>Ok, we can add `alloca`'ed space is each function invocation, plus an
>>overhead constant per active `alloca`.

>

>

> The `alloca()` calls can be in a loop. Knowing how much memory a set of
> `alloca`'s will perform in any function is equivalent to solving the
> halting problem.

So what? Why is this relevant? Most of the time you can still give an upper bound. In cases where you can't (few and far inbetween) you don't have the guarantee, which is a pity; the code can no longer be written in a way that's fully portable. It would still be much better than what happens now (standard-compliant programs segfaulting without so much as a hint in the standard).

>>Unless you allow that I'm trying to define an upper bound of total use,
>>which may be loose as far as I am concerned. The idiomatic use would be
>>to compare this to the `ULMU_TOTAL` to see if it fits. This may yield the
>>number of bytes on the smallest check for all I care. On those silly
>>platforms, you will grossly underestimate stack capacity now and then.
>>[...] Still beats the current situation.

>

>

> This versus just living with runtime stack checking? I'll take the
> runtime stack checking.

Then define a proposal to formalize "runtime stack checking" in standard-like language.

>> [...]

>>My opinion is determined largely by the psychological objections (to
>>which you don't subscribe.)

> Says who? You see, unlike some people, by idea isn't driven by an
> opinion. Its driven by a consideration for design. Since you **CAN'T**
> add just any old operator to satisfy everyone, then how do you satisfy
> the demand for more operators? Some variation on redefinable
> operators or operator overloading is the logical conclusion -- just

comp.lang.c: Re: Is C99 the final C? (some suggestions)

- > *fix its problems (since type-index functions don't really exist in C,*
- > *so you just add in a bunch of them so that you don't lose any of the*
- > *ones you have). For all I know, I might not ever use such a feature,*
- > *even if it *WERE* to be endorsed by the standard.*

I fail to see the point you want to make here.

- >>*My technical problem is that I can see no way that this could be*
- >>*implemented.*
- >
- >
- > *Well I don't know what to make of this -- you can't see a race*
- > *condition when its staring you in the face, and you can't imagine*
- > *problems with knowing the size of a function's stack usage.*

Be careful not to presume things that are not there. Other than that, make of it what you like.

- >> *[...]*
- >>*There is a possibility of error caused by bad design (which isn't*
- >>*trimmed to the type of mistakes we fallible humans make) in your*
- >>*operator introduction scheme.*
- >
- >
- > *Same is true with free form function names, of course.*

The risks of errors are minute, in comparison.

- >>>>*You haven't shown me that it can be implemented in a compiler,*
- >>>>*even in theory. That should count for something.*
- >>>>
- >>>>*I don't think there's a credible assertion that my idea is*
- >>>>*impossible. You want me to describe a whole compiler*
- >>>>*implementation just to support my idea?*
- >>>>
- >>>>*I listed some specific objections with regards to disambiguation of*
- >>>>*the grammar that the parser would have to be able to handle.*
- >
- > *Even without a full grammar being specified? That's a pretty good trick.*

Yes, it's called "abstract thinking". It's a cousin of "thought experiment".

- > *[smartcards]*
- > *Yes they do. That's the point of creating a pico-java assembly*
- > *language. Its so that you just have to upload a minimal java kernel to*
- > *it, and all of a sudden you have the whole world's Java libraries at your*
- > *disposal.*

How much would these combined libraries consume, in terms of memory on the smartcard? I mean, the bignum-multiply code has to reside somewhere.

comp.lang.c: Re: Is C99 the final C? (some suggestions)

>>[...]
>>Great. Now what has all this to do with defining a sensible signed <+
>>semantics for use in C (you know, a general purpose language, not tied
>>to your particular pet peeve problem).
>
>
> Oh yeah my particular pet peeve problem, which only every serious CPU
> company on the planet is only too willing to solve for me.

You haven't answered the question.

>>>It all just comes down to the carry flag. The other flags are not g
>>>necessary for bignum libraries. Maybe there's a way to transform
>>>the OF into something that can be used in a bignum library, but I
>>>am not aware of it.
>>
>>Because, we're talking about expanding the C language here (not about
>>your fairy-tale operator introduction hocus-pocus). You need decent
>>semantics.

> CPU companies have laid out the decent semantics. I and others have
> written bignum routines -- the carry flag is where its at as far as
> this is concerned.

That's fine. Now here's a big idea: hardware design and general-purpose
language design are two different things.

>>>[...]
>>>OF is never the important flag. Its not relevant at all.
>>
>>I guess that's why everything since the 4004 supports it.

> That's because OF is used for <, >, <=, >= comparisons. Outside of
> this, I have never seen OF being used for any purpose. I.e., I would
> surmise that the only relevant usage of OF is already encoded in the
> language semantics of C (and just about any other language with
> integers.) But I could be wrong -- can you name at least one other
> purpose which isn't really just an encoding of <, >, <=, >= ?

Adding, subtracting, multiplying, dividing. Basically anything
arithmetic you can do that can map the result of an operation on signed
ints to a value that doesn't correspond to the mathematical
interpretation of what the result should be.

>>You must really consider that there's more to life than multiplying big
>>numbers (and that's coming from one of the few people that sees the fun
>>in it). We're not going to extend C with an operator to fit this (or
>>ANY) one purpose, however important it may be.
>
>
> Carry flag comes up time and again. Just look it up on google:

Re: Is C99 the final C? (some suggestions)

comp.lang.c: Re: Is C99 the final C? (some suggestions)

>
> <http://www.google.com/search?hl=en&lr=&ie=ISO-8859-1&safe=off&c2coff=1&q=jc+ret+mov>

That seems to yield some results concerning intel assembly. Interesting, but how does that address my point?

> *Off the top of my head, there's chain shifting, bit counting, and predication/masking.*

You don't have to convince me that carry flags are important. At times I have found myself in want of a way to access them myself, from C. However, the point is that you would have to find a good semantics, that fits in with the rest of C, not some ad-hoc operator to make some bignum implementors happy.

>>[...]
>>>*In this case, I am just copying the x86's MUL -> EDX:EAX or AMD64's MUL -> RDX:RAX semantics.*
>>
>>*So again (as with the carry vs. overflow), you're going to perform an instruction that works on UNSIGNED integers even on signed integers.*
>
> *Yes.*

That doesn't rhyme well with C. There are many subtle issues with signed/unsigned representation and operations, and the standard is careful to find a compromise; not assuming too much on the architecture side, while at the same time maintaining a sound semantics for both signed and unsigned operations. Your semantics just doesn't fit in.

>>*Pardon my French, but that's just plain stupid. Perhaps you should be a little less confident about your own understanding of bit-twiddling...*

> *Perhaps you should consider that there's a reason every assembly language in existence associated signedness with particular *OPERATIONS* not the *OPERANDS*.*

Are you aware that the C standard allows for other representations than 2-complement? Assume 1-complement representation for a moment (the standard is specifically worded to allow this).

10000001 - 00000001 yields 10000010 if '-' is a signed operator
10000001 - 00000001 yields 10000000 if '-' is an unsigned operator

Now I don't know assembly for any 1-complement architecture, but from this it seems to me that there will have to be different signed and unsigned versions of 'SUB'.

> *Having signed and non-signed versions of every operator may be something that fit C when it was first designed, but that*

Re: Is C99 the final C? (some suggestions)

comp.lang.c: Re: Is C99 the final C? (some suggestions)

> *doesn't make it necessarily correct for all relevant computations.*

Well, if (and that's a big if) something resembling your wide-mul would ever make it in the standard, I'd be disappointed if it didn't for instance allow me to multiply two big signed integers. It would be messy to only support unsigned ones.

Best regards,

Sidney