

Re: The annotated annotated annotated C standard

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2008-01/msg01006.html>

- *From:* spinoza1111 <spinoza1111@xxxxxxxx>
 - *Date:* Sun, 20 Jan 2008 00:33:01 -0800 (PST)
-

On Jan 20, 3:16 pm, Richard Heathfield <r...@xxxxxxxxxxxxxxxx> wrote:

spinoza1111 said:

On Jan 20, 6:35 am, "Stephen Howe" <sjhoweATdialDOTpipexDOTcom> wrote:

<snip>

Most of the places where C (and C++ for that matter) standards calls something "undefined" is because the hardware (or OS) may do some unexpected. Like terminate your program. For example "dividing by 0" is undefined. free()'ing the same pointer twice is undefined. The standards "undefined" is equivalent to posting a notice on a pond frozen with ice, "THIN ICE". As long as programmers take care to never allow their programs to skate in that area, their programs will behave well. If you decide to skate over the "THIN ICE", there is no guarantee what will happen to your program.

This was the case in out of date languages. It is no longer the case. In fact, a runtime error produced by a bounds checker is preferable to "undefined behavior", as is INF or a similar symbol.

Whether this is preferable depends very much on the situation. What Stephen didn't mention is that there are times when a programmer needs to step

Re: The annotated annotated annotated C standard

outside the Standard in order to achieve a platform-specific goal. When we do this, it very often involves constructs that the Standard declines to define. For example, we might need to point to a particular hardware address that we know exists and is writeable on our particular platform.

Why not use assembler language and thereby avoid even the hint that the code might "port". Job One should be giving no manager no excuse to force your mates to "reuse" a machine dependent program.

Programmers are over-frequently alienated not only from the tools of production but from their own creations; as I have pointed out before, most code has no father and decays: code in C decays rapidly.

Therefore it's your professional responsibility to choose assembler in this case and not C. If you're a studly programmer you don't need syntactical sugar.

The C Standard can't possibly be expected to know every nuance of every architecture past, present, and future, so it can't tell us what's going to happen when we use such techniques. That is, the behaviour is undefined. Nevertheless, if we know what we're doing, we can get the machine to do what we want, by using our platform-specific knowledge and recognising that we are rendering our code non-portable to other systems.

Programming low level in a high level language is, I now believe, a species of intellectual fraud.

Doug Gwyn once said that "UNIX was not designed to stop you from doing stupid things, because that would also stop you from doing clever things."

I am underwhelmed by programming cleverness, having been clever myself on occasion, having programmed a compiler in machine language. I'd rather learn more about music and art.

Very much the same idea applies to C. Yes, some modern languages are sand-boxes in which you can't break anything no matter how hard you try, but this is at the expense of limiting your horizons to things which, at some level, the language designers envisaged when they decided what to allow and what to disallow.

"The anarchism of the lower middle class". – E. F. Hobsbawm

Re: The annotated annotated annotated C standard

ISO C doesn't impose such limits. It draws a line around the abstract machine and says, effectively, "in here, we can make some promises about program behaviour – but if you step across the line, you're out of the box and you're on your own".

In fact, mathematics itself progressed because mathematicians weren't intimidated by thugs on standards committees, whence complex numbers.

Because mathematicians frequently ignored rigour, mathematics found itself in a deep hole in (I believe) the mid-19th century, because while one mathematician was proudly publishing a proof of some theorem or other, some other mathematician was equally proudly publishing a disproof – and no flaw could be found in either paper. Goedel hadn't done his thing yet, and mathematics was still believed to be consistent and complete. So mathematicians were forced to the conclusion that some of the stuff they were relying on had not properly been proved, and they had to re-examine *everything* from the ground up, to put mathematics on a (relatively) sound axiomatic and theoretical footing. This took a lot of people a lot of time. Nowadays, rigour is de rigueur for formal mathematics. And so it is for competent programmers.

The problem with this metaphor was that the "rigor" of Lord Russell's Principia Mathematica turned out to be a complete dead end and a complete waste of ink, perhaps as will the C standard. It was followed in Holland and elsewhere on the Continent with a return to Kant and Brouwer's intuitionism, according to Knuth sometime ago, can be compared to structured programming.

Intuitionism in fact refuses the use of the law of the excluded middle. It is unlike C in that it takes tools away from the mathematician and is closer in spirit to Algol and Pascal.

Elegance, and a Kantian admission of unknowability, is even more important than "rigor", and as far as I can see, you nowhere think rigorously anyway.

<snip>

The analogy is imperfect. For example, nonsense about "sequence points" probably and as far as I can tell probably belongs with cold fusion and the aether.

The importance of sequence points is that we, as programmers, can *use*

Re: The annotated annotated annotated C standard

them to say to the compiler, "once you get to this point, you need to make sure that no matter what optimisations you're doing, you should by now have done everything that I've told you to do so far", and the compiler is required to make the program behave as if this were true (and by far the easiest way of doing that is to ensure that it /is/ true). Thus, sequence points are a tool for the careful programmer, not some kind of club with which to beat ignorant programmers.

But in mathematics, as opposed to programming, division by zero has a result in the sense that it's the limit of the function x/y as y approaches zero, I believe. Genuine math geeks as opposed to thugs are welcome to correct me on this matter, of course.

In mathematics, it is true that there are some cases where division by zero is meaningful. Wolfram has this to say about division by zero:

++++++ begin quote ++++++

Division by zero is the operation of taking the quotient of any number $[x]$ and 0, i.e., $[x/0]$. The uniqueness of division breaks down when dividing by zero, since the product $[0.y==0]$ is the same for any $[y]$, so $[y]$ cannot be recovered by inverting the process of multiplication. 0 is the only number with this property and, as a result, division by zero is undefined for real numbers and can produce a fatal condition called a "division by zero error" in computer programs.

To the persistent but misguided reader who insists on asking "What happens if I do divide by zero," Derbyshire (2004, p. 36) provides the slightly flippant but firm and concise response, "You can't. It's against the rules." Even in fields other than the real numbers, division by zero is never allowed (Derbyshire 2004, p. 266).

There are, however, contexts in which division by zero can be considered as defined. For example, division by zero $[z/0]$ for $[z \text{ in } C^{*!}=0]$ in the extended complex plane C^{*-} is defined to be a quantity known as complex infinity. This definition expresses the fact that, for $[z!=0]$, $[\lim_{(w \rightarrow 0)} z/w == \text{infty}]$ (i.e., complex infinity). However, even though the formal statement $[1/0 == \text{infty}]$ is permitted in C^{*-} , note that this does not mean that $[1 == 0.\text{infty}]$. Zero does not have a multiplicative inverse under any circumstances.

++++++ end quote ++++++

<snip>

Wolfram contradicts himself: you can't, but you can.

Basic with the structured programming additions happens to be superior to C. It evaluates the for limit once and doesn't allow stupid cleverness.

Re: The annotated annotated annotated C standard

If what you say about BASIC's for loop is correct (and I am not about to take your word for it), this makes BASIC's for loop less flexible than that of C. Full evaluation of the termination condition each time through the loop is a powerful construct if used properly.

It already exists, and it's called the while statement.

People need a minatory "rigor" only when infected all the way down by the lack of committment to truth which is a necessary precondition to lower middle class survival in lives that are fueled by denial. This is why Dijkstra says "in computing science, elegance [not rigor] is not a dispensable luxury, but a matter of life and death".

.