

Suggestions for the ISO/IEC C committee

Eric Blair

30 October 2007

I'm the most fanatical advocate of the C programming language I know. I wrote a darn book about how it should be applied to places where scripting languages are more commonly used.

So, as one who has written way too much on why everybody else's language sucks, here are some notes on how I think C could be improved. Beyond a few security tweaks, there isn't really any pressing for evolving C beyond the 1999 standard, but here's my wish list anyway, in case another round of reform should ever happen. As above, it's primarily based on a desire to learn from some scripting languages regarding syntactic conveniences that are almost implementable in C, but not quite. And as a wish list, I have no idea what debates have been had before about these issues or the counterarguments about why these will cause havoc (and the ISO/IEC doesn't post rationales for past rejections anywhere that I could find them). I just want them.

The reader will note that none of these suggestions will require rewriting existing code, and none of them require a serious shift from what a C compiler can do now (save for maybe the thing about counting arguments in variadic functions, which is a frickin' security risk and needs to be addressed anyway). The first half are simple wish list items; the second half is about the form of variadic functions, which shows such potential but in its current form is broken.

asprintf needs to be part of the standard The problem is that any string operation is three steps: measure the length the string is about to be, resize the string, then place the data you'd just measured into the string. You can home-brew convenience functions to do these three steps for most operations like copying or concatenation, but for `snprintf`, it is basically impossible to do. Everybody keeps reinventing `asprintf` (GCC and BSD both have it), and it is so much more convenient than `snprintf` that I consider that function to be basically obsolete. Because string length bugs are such a hacker favorite, the missing `asprintf` is also a security concern.

-> **versus .** I don't remember who pointed this out, but there's no difference between these two. If you use `parent.child` when you meant `parent->child`, or vice versa, you just get an error and have to go back and switch to the other one. There is no ambiguity: the syntax should use a period for both. This is already how function pointers are handled, because if you call a pointer-to-function when you meant to call the function itself, the system can do what you unambiguously mean without being all pedantic and demanding that you fix the syntax.

Division by int In many programming languages, $8/5=1$. This is part of the conflict between computer scientists, who really want division by two ints to always return an int, and we humans who expect $8/5=1.6$. This should be an optional compiler warning that humans can turn on as desired. The compiler can even do one of those polite things like ‘suggest parentheses around int divided by int’ so when you really need int/int you can have it. [So no, the ISO/IECC folks don’t have to get involved.]

Nested functions All those guys who brag endlessly about how wonderful LISP-type languages can be are really just saying that it’s great to have lightweight inline functions. And hey, it is. [Maybe they’re also bragging about how far you can get without using any state variables, but you can write in a state-variable-free style in C too. The typical modern compiler will even recognize tail recursion and compile accordingly.]

GCC already supports nested functions, and it seems like it’d be harder to not support them. After all, the placement of a function is really just a question of what variables are in scope. We already know how to deal with scope where one block is inside another block (like a for loop inside a for loop), so it’s a simple application of the block-in-block scoping rules to have functions-in-functions. This made so much sense to me that I didn’t even know nested functions aren’t part of the standard until somebody told me my code won’t compile on MSFT compilers.

As for proper in-line functions, they could maybe be declared in the format used for anonymous structs below, but I don’t know if there’d be much benefit. It’s already hard to read when people in those super-elegant languages put an anonymous function in the middle of a lengthy function call, and a single function header is blockier in C, what with all those types. Nested functions, such that you can declare the function on one line and use it on the next, will suffice.

A smarter preprocessor The preprocessor gives you one form for macros: `newfunc(a, b, c)`. We can use regular expressions to do better. For example, some people really prefer `2Darray[i, j]` over `2Darray[i][j]`—even the first edition of KR acknowledges this. It’d be pretty simple to specify that the preprocessor, whose sole job is text substitutions, be able to substitute one for the other.

Two other examples I’d immediately implement if somebody gave me the chance: `v[[i]]` to `gsl_vector_get(v, i)`, and `object.process(input)` to `process(object, input)`. That second form allows you to have a `self` or `this` variable passed to functions inside structs, but without having to deal with the other sixty-one (61) keywords of C++. [I’ve discussed this before¹.] All you need is a simple text substitution, but the preprocessor won’t even give us that much.

Better variadic functions: default values You could almost do this now with an inline struct. Here is code which would be valid if it were in a real program. Instead of passing individual arguments, it passes a single ad hoc struct that holds all the function arguments. Because you can send an anonymous inline struct whose values are set via designated initializers, and whose undeclared values are set to zero, you can simulate named arguments with default values.

¹<http://fluff.info/blog/arch/00000162.htm>

```

typedef struct{
    int top, bottom;
    char *left, *right
} ad_hoc_struct;

int call_me(ad_hoc_struct in){
    if (!in.top) in.top = 287;
    if (!in.bottom) in.bottom = 34;
    ...
}

call_me((ad_hoc_struct) {top=12, left="Hi there"});

```

Of course, this is a syntactic mess, but the fact that you can do this via standard C shows that we're not talking about completely gutting the language, just major cleanup on form.

To outline the work the system would need to do: if there's an = in the function declaration, the compiler would need to declare a struct with the same scope as the function, and would need to parse the call through the struct. Because it is currently not valid to have an = in a function declaration, this does not interfere with existing code.

I bet somebody could write this as a preprocessor script. That would be neat.

Better variadic functions: argument count My impression is that the variadic function setup is exactly sufficient to implement `printf`, and no more. I feel like I'm on a tightrope without a net every time I use a variadic function because there is no way to know the type or number of arguments. The system knows at compilation, but the standard obligates it to just throw that info out.

For the number of arguments, it'd be cheap and easy for the system to simply count them and send in a phantom variable with a name like `_nargs` (or such) that indicates the count. This would be a cheap way to implement default values as well: in the example above, if `_nargs < 2`, then let `bottom=34`, or such.

Oh, and it's entirely '70s that we need at least one fixed-type argument (and is once again exactly what you need for `printf` but annoying in many other contexts). That'd be obsolete if we got the number of args sent in.

Better variadic functions: type safety Also, variadic functions throw the type checking mechanism out the window. For the `printf` family, there's a sufficiently set means of dealing with the variable arguments that GCC can check up on you, but for anything else, you're screwed. If you have a function that takes in two `long doubles`, and you make a call like `doublevariadic(in, 1, 0)`, then you're screwed, because the bits that represent the integer 1 look nothing like the bits that represent the long double 1.0. In fact, on a bad day, this will just crash and/or produce a security risk, because reading two long doubles may go past the input data block.

To make this concrete, here's some sample code that crashes and burns. The fix is to replace 2 and 3 with 2. and 3., which shows how subtle are the errors that variadic functions in their current form invite. Note the incredibly convenient C99 use of a declaration inside a `for` loop header, indicating that the ISO/IEC standards committee is not insensitive to the human desire for grace and convenience.

```
#include <stdio.h>
#include <stdarg.h>

void vtest(int ct, ...){
    va_list va;
    va_start(va, ct);
    for (int i=0; i<ct; i++)
        printf("%g\n", va_arg(va, double));
}

int main(){
    vtest(4, 1., 2, 3, 4.);
}
```

This one may be a compiler issue too, because the C standard allows functions to take on compiler-specific `__attribute__`s. For example, GCC lets you check a function with a `printf`-style format string against subsequent arguments via the `format` attribute. Similarly, one could imagine a `vartypes(char*)` attribute that would tell the compiler to check that all variadic arguments be `char*s`, for example. That would work for the function above, but wouldn't necessarily be helpful if we're expecting, say, a potentially infinite list of `int-double*` pairs.

Anyway, variadic functions need reform, because they are the most unsafe part of the language. But they also have a lot of potential, because they could be used to implement the sort of pleasantries like default arguments and named arguments that every modern coolio scripting language has.