Better variadic functions in C

Ben Klemens

XXX

I really dislike how C's variadic functions are implemented. I think they create lots of problems and don't fulfil their potential. So this is my effort to improve on things.

A variadic function is one that takes a variable number inputs. The most famous example is `printf`, where both `printf("Hi.")` and `printf ("%f %f %i\n", first, second, third)` are valid, even though the first example has one input and the second has four.

Simply put, C's variadic functions provide exactly enough power to implement `printf`, and nothing more. You must have an initial fixed argument, and it's more-or-less expected that that first argument provides a catalog to the types of the subsequent elements, or at least a count. In the example above, the first two items are expected to be floating-point variables, and the third an integer.

There is no type safety: if you pass an `int` like 1 when you thought you were passing a `float` like 1.0, results are undefined. If you think there are three elements passed in but only two were passed in, you're likely to get a segfault. Because of issues like this, CERT, the software security group, considers variadic functions to be a security risk[1] (Severity: high. Likelihood: probable).

I understand that the designers of the system are reluctant to impose too much magic to make variadic functions work, like magically dropping into place an `nargs` variable giving an argument count. So today's post is an exercise in how far we can get in implementing decent variadic functions using only ISO C. To give away the ending, I manage some of the things we use variadic functions for in a safe and more convenient manner—optional arguments work very well—but it takes many little tricks, and I'm still short of true `printf` functionality.

Designated intializers First, a digression into a pair of nifty tricks that C99 gave us: compound literals and designated initializers. I find that many people aren't aware of these things, because they're learning C from textbooks written before 1999, and using compilers that may not use the 1999 standard by default.

Darn it people, it's been a decade. This is not new.

The idea is simple: if you have a `struct` type, you can use forms like these to use an anonymous struct wherever it's appropriate:

[1] `https://www.securecoding.cert.org/confluence/display/seccode/DCL11-C.+Understand+the+type+issues+associated+with+variadic+functions`

```
typedef struct {
    int first, second;
    double third;
    gsl_vector *v;
} stype;

stype newvar = {3, 5, 2.3, a_vector};
stype nextvar = {3, 5};
newvar = (stype)  {.third = 3.12, .second=5};
function_call( (stype) {.third = 8.3});
```

In each case, a full struct is set up, and the compiler is smart enough to know what goes where among those elements you specified, and sets the other elements to zero or NULL.

These sorts of features that we have for initializing a `struct` are exactly the sort of thing many more recent languages put into their function calls: default values are filled in, and named elements are allowed via designated initializers.

At the end of the example, I put a compound literal inside a function call, so we are technically calling a function using these pleasant variable-input features, but it's not yet looking much like `printf`.

Cleaner function calls We can clean up the struct-to-function trick to get a lot closer to variadic functions. Here's the agenda for making this work:

- For each function, set up a struct where the elements of the struct are the inputs to the function.
- Produce a shadow function whose sole input is that struct, which sets the default vaules and then calls the original function.
- Write a wrapper macro so that the instead of the user having to type the full compound literals form `f( (ftype) {arg1, arg2})`, they can just type the usual `f(arg1, arg2)`.

So, here it is. The first third is a set of general macros, the second third sets up a single function, and the last third actually makes use. This program should compile with any C99-compliant compiler. After the code, I'll have some detailed notes to walk you through it.

```
#define varad_head(type, name) \
         type variadic_##name(variadic_type_##name x)

#define varad_declare(type, name, ...) \
        typedef struct {            \
                    __VA_ARGS__       ;  \
                } variadic_type_##name;     \
    varad_head(type, name);
```

```
#define varad_var(name, value) name = x.name ? x.name : (value);
#define varad_link(name,...) \
        variadic_##name((variadic_type_##name) {__VA_ARGS__})


//////////////////// header + code file

varad_declare(double, sum, int first; double second; int third;)
#define sum(...) varad_link(sum,__VA_ARGS__)


varad_head(double, sum) {
    int varad_var(first, 0)
    double varad_var(second, 2.2)
    int varad_var(third, 8);

    return first + second + third;
}


//////////////////// actual calls
#include <stdio.h>

int main(){
    printf("%g\n", sum());
    printf("%g\n", sum(4, 2));
    printf("%g\n", sum(.third=2));
    printf("%g\n", sum(2, 3.4, 8));
}
```

• There are three macros in the first section, roughly corresponding to the three steps of the agenda. `varad_declare` declares a special type and a function to use that type. Notice that the third and later arguments to the macro go into the struct, not a function header, so variables are separated by semicolons. `varad_var` sets default values for each variable. `varad_link` is used to clean up the function call.

• The second third sets up a single function. The bulk declares that intermediate function that takes in a struct, sets default values, and calls the real function.

• There is one more macro in this section, which needs to be rewritten for every new function. It'd be great if there were a macro to just churn out this trivial macro for each new function, but you can't write macros that generate macros. Why not? I dunno. Seems like it wouldn't be a big deal for the preprocessor, but them's the rules. The too-simple preprocessor is my second big complaint about C.

• The main part of the intermediate function has a line for each element of the struct, declaring an intermediate variable and setting a default value. The compiler gave

missing elements a default value, but we often want the default to be something other than zero. We can also have more intelligent defaults based on variable information, like maybe `int varad_in(third, first * 3)`.

• The third part is a call to the function we've set up, and you can see that it works great: we can give it no arguments, all arguments, named arguments, or whatever else seems convenient, with no regard to the internal guts from prior sections.

Infinite input OK, so far, the result looks much more modern relative to C's standard fixed inputs. It allows optional arguments, and named arguments. It checks types, and complains during compilation if you've got mismatched types, meaning that a lot of the security holes of the standard variadic form are gone.

But we want more from our variadics than just optional arguments: we'd like to specify arbitrary-length lists. Can we declare a structure that could take an arbitrary number of inputs, such as a function to sum n inputs?

The short answer is no. [The long answer: the last element of a struct can be an array of indeterminate size, to be allocated at compile-time. When the anonymous struct is being generated for the function call, a compiler could count the elements at the end of the list and allocate the variable-size array appropriately.

However, this depends on whether the anonymous struct is dynamic or static. By static, I mean something produced at the initialization, like the constants or global variables; by dynamic, I mean variables that are initialized along the way during the run. For static variables, the variable-length last argument will be stuck in the form set at the first allocation; for dynamic variables, there are more options. So what are the anonymous structs used for the function calls here? It is my reading that the ISO C standard doesn't demand things one way or the other, so we can't rely on dynamic allocation of the type we'd get elsewhere via a line like `x = (structtype) {1, 2, {3, 5, 9, 10}}`, where the variable-lenght allocation would be valid.

Also, for an array of fixed length, you can usually get the size by `sizeof(list)/sizeof(list[1])`. So if the system allocated the right-sized list, you wouldn't even need a separate `nargs` element taking up space.]

We're instead stuck just making up a size for an element of the struct, like 1,000, and letting the compiler pad it with zeros. Given that we generally use variadic arrays for lists of items hand-typed into the code, and array inputs for lists of truly arbitrary length, we can probably get away with 1,000 inputs max, but it's certainly not ideal.

```
//Put the macros from the first third above in "variadic.h".
#include "variadic.h"

varad_declare(double, sum, int first;
                double second; int third[1000];)

#define sum(...) varad_link(sum,__VA_ARGS__)
varad_head(double, sum) {
    int varad_var(first, 0)
    double varad_var(second, 2.2)
    int * varad_var(third, NULL);

    double sum = first + second;
    for (int i=0; i< 1000; i++)
```

```
        sum += third[i];
    return sum;
}

int main(){
    printf("%g\n", sum());
    printf("%g\n", sum(4, 2));
    printf("%g\n", sum(.third={2}));
    printf("%g\n", sum(2, 3.4, {8, 8}));
}

#endif
```

So the `third` element can have variable length, as desired. If you're not sure of the type coming in, the last element can be an array of `void *`, where `void *` is your signal to the compiler that you're willing to take your chances on types and have a catalog or system on hand to do your own casts.

How're we doing? So there's the story: we can do a lot better with our variable-length function calls than we do, and it's not even something involving crazy re-writing of everything. Standard C already gives us the tools to go 90% of the way.

However, it's a frickin' pain to set up. C's preprocessor is limited, and we had to write several macros to make this happen. For every function, the namespace has to have another auxiliary function and a type floating around. You won't notice this normally, but it can create quirks in the debugger and other places that expect a little more normalcy out of the code base.

Apophenia uses this setup for a few dozen functions, but with a few more tricks. Notably, everything is wrapped in `#ifdef`s to let everything degrade to the standard function call if needed. Many things beyond the compiler eat C code, like documentation generators, interface generators, &c. Even though all the above is 100% standard C compliant, some systems like the setup more than others. [I also wrote a sed script to generate all this boilerplate from appropriate markers. The script also this gets around the problem that we can't use the preprocessor to generate macros.]

OK, summary paragraph: we need to fix C's variadic function calling scheme, which is built around `printf` to the detriment of many other possibilities, and even to the detriment of security. Being that we can already do most of what we want via ISO C99, we can fix them without introducing incompatibilities or changing the character of C. But given the amount of extras and tricks involved, and given that we still don't quite achieve proper variadic functions, there'd need to be some fixes in the language itself to update variadic functions to a modern form.