C/C++: Passing variable number of arguments around

Say I have a C function which takes a variable number of arguments: How can I call another function which expects a variable number of arguments from inside of it, passing all the arguments that got into the first function?

Example:

```
void format_string(char *fmt, ...);

void debug_print(int dbg_lvl, char *fmt, ...) {
    format_string(fmt, /* how do I pass all the arguments from '...'? */);
    fprintf(stdout, fmt);
 }
```

c varargs

edited Jul 9 at 19:37

Mooing Duck
21.3k 4 32 72

asked Oct 15 '08 at 16:58

Vicent Marti
2,548 2 16 26

4 Your example looks a bit weird to me, in that you pass fmt to both format_string() and to fprintf(). Should format_string() return a new string somehow? – Kristopher Johnson Oct 15 '08 at 17:14

71 "should be googled": I disagree. Google has a lot of noise (unclear, often confusing information). Having a good (voted up, accepted answer) on stackoverflow really helps! – Ansgar Feb 6 '09 at 9:53

48 Just to weigh in: I came to this question from google, and because it was stack overflow was highly confident that the answer would be useful. So ask away! – tenpn Feb 24 '09 at 12:18

3 @Ilya: "Just look in the linux implimentation of printf" - ha ha - good one, no really - funny. Ha ha still laughing... Ive got a job to do, I want an answer - I found it here and applied it in less that 1 minute. You really think Ive got time to firetruck about in linux source code? I envy your lifestyle. – Ricibob Jul 3 '12 at 20:22

10 @Ilya: if nobody ever wrote down stuff outside of Google, there would be no information to search for on Google. – Erik Allik Aug 29 '12 at 20:27

show **2** more comments

7 Answers

To pass the ellipses on, you have to convert them to a va_list and use that va_list in your second function. Specifically;

```
void format_string(char *fmt,va_list argptr, char *formatted_string);


void debug_print(int dbg_lvl, char *fmt, ...)
{
 char formatted_string[MAX_FMT_SIZE];

 va_list argptr;
 va_start(argptr,fmt);
 format_string(fmt, argptr, formatted_string);
```

```
 va_end(argptr);
 fprintf(stdout, "%s",formatted_string);
}
```

edited Jul 9 at 19:28
Servy
60.3k 6 53 109

answered Oct 15 '08 at 17:11
Shane MacLaughlin
12.1k 3 37 87

3 I'm writing this comment as a sort of public safety announcement. One of my students got confused by the fprintf call at the end, and thought that you don't need to send on argptr as an argument. To make things worse it actually worked, as undefined behavior sometimes does. So, please note that, as Vicent Marti wrote in a comment to his question, the final call to fprintf doesn't make sense! – Thomas Padron-McCarthy Feb 24 '12 at 16:03

2 The code is taken from the question, and is really just an illustration of how to convert ellipses rather than anything functional. If you look at it `format_string` will hardly be useful either, as it would have to make in-situ modifications to fmt, which certainly shouldn't be done either. Options would include getting rid of format_string altogether and use vfprintf, but that makes assumptions about what format_string actually does, or have format_string return a different string. I'll edit the answer to show the latter. – Shane MacLaughlin Feb 24 '12 at 16:35

There's no way of calling (eg) printf without knowing how many arguments you're passing to it, unless you want to get into naughty and non-portable tricks.

The generally used solution is to always provide an alternate form of vararg functions, so `printf` has `vprintf` which takes a `va_list` in place of the `...`. The `...` versions are just wrappers around the `va_list` versions.

answered Oct 15 '08 at 17:11
Mike F

Variadic Functions can be **dangerous**. Here's a safer trick:

```
void func(type* values) {
     while(*values) {
         x = *values++;
         /* do whatever with x */
     }
 }

func((type[]){val1,val2,val3,val4,0});
```

answered Nov 27 '11 at 4:51
Rose Perrone
9,034 2 47 53

+1, That's Neat! – Alok Save Nov 27 '11 at 9:18

4 Even better is this trick: `#define callVardicMethodSafely(values...) ({ values *v = { values }; _actualFunction(values, sizeof(v) / sizeof(*v)); })` – Richard J. Ross III Mar 3 '12 at 22:54

2 @RichardJ.RossIII I wish you would expand on your comment, it's hardly readable like this, I can't make out the idea behind the code and it actually looks very interesting and useful. – penelope Mar 30 '12 at 10:09

@penelope basically, it creates an array from the values passed in to the macro. It may only work on GCC, but I don't know for sure. – Richard J. Ross III Mar 30 '12 at 12:29

2 @ArtOfWarfare i am not sure i agree that its a bad hack, Rose has a great solution but it involves typing func((type[]){val1, val2, 0}); which feels clunky, wheras if you had #define func_short_cut(...) func((type[]){**VA_ARGS**}); then you could simply call func_short_cut(1, 2, 3, 4, 0); which gives you the same syntax as a normal variadic function with the added benefit of Rose's neat trick...whats the issue here? – chrispepper1989 Jul 24 at 14:05

show 2 more comments

show 2 more comments

In magnificent C++0x you could use variadic templates:

```
template <typename ... Ts>
void format_string(char *fmt, Ts ... ts) {}

template <typename ... Ts>
void debug_print(int dbg_lvl, char *fmt, Ts ... ts)
{
  format_string(fmt, ts...);
}
```

answered Mar 24 '11 at 22:14

user2023370
2,289 8 26

1 +1. I like this. – Xoorath Oct 6 '11 at 22:51

Don't forget that variadic templates are still not available in Visual Studio... this may well be of no concern to you of course! – Tom Swirly Sep 7 at 1:01

If you are using Visual Studio, variadic templates can be added to Visual Studio 2012 using the November 2012 CTP. If you're using Visual Studio 2013, you will have variadic templates. – user2023370 Sep 7 at 14:12

You can use inline assembly for the function call. (in this code I assume the arguments are characters).

```
void format_string(char *fmt, ...);
void debug_print(int dbg_level, int numOfArgs, char *fmt, ...)
    {
        va_list argumentsToPass;
        va_start(argumentsToPass, fmt);
        char *list = new char[numOfArgs];
        for(int n = 0; n < numOfArgs; n++)
            list[n] = va_arg(argumentsToPass, char);
        va_end(argumentsToPass);
        for(int n = numOfArgs - 1; n >= 0; n--)
        {
            char next;
            next = list[n];
            __asm push next;
        }
        __asm push fmt;
        __asm call format_string;
        fprintf(stdout, fmt);
    }
```

answered Jul 26 '11 at 19:40

Yoda
37 1

2 Not portable, depends on the compiler, and prevent compiler optimization. Very bad solution. – Geoffroy Oct 20 '11 at 18:55

1 New without delete too. – user7116 Jul 9 at 19:38

Ross' solution cleaned-up a bit. Only works if all args are pointers. Also language implementation must support eliding of previous comma if **VA_ARGS** is empty (both Visual Studio C++ and GCC do).

```
// pass number of arguments version
 #define callVardicMethodSafely(...) {value_t *args[] = {NULL, __VA_ARGS__}; _actu
```

```
// NULL terminated array version
 #define callVardicMethodSafely(...) {value_t *args[] = {NULL, __VA_ARGS__, NULL};
```

edited Oct 8 '12 at 11:49

answered Oct 5 '12 at 23:49

BSalita

700 4 19

I'm unsure if this works for all compilers, but it has worked so far for me.

```
void inner_func(int &i)
{
  va_list vars;
  va_start(vars, i);
  int j = va_arg(vars);
  va_end(vars); // Generally useless, but should be included.
}

void func(int i, ...)
{
  inner_func(i);
}
```

You can add the ... to inner_func() if you want, but you don't need it. It works because va_start uses the address of the given variable as the start point. In this case, we are giving it a reference to a variable in func(). So it uses that address and reads the variables after that on the stack. The inner_func() function is reading from the stack address of func(). So it only works if both functions use the same stack segment.

The va_start and va_arg macros will generally work if you give them any var as a starting point. So if you want you can pass pointers to other functions and use those too. You can make your own macros easily enough. All the macros do is typecast memory addresses. However making them work for all the compilers and calling conventions is annoying. So it's generally easier to use the ones that come with the compiler.

answered May 8 at 21:30

Jim

1

Not the answer you're looking for? Browse other questions tagged c varargs or ask your own question.