<https://blog.csdn.net/li_ning_/article/details/51571165>

说的非常详细，但是有部分口误，希望只吸取精华：

原文地址：http://www.cnblogs.com/cpoint/p/3368993.html

C函数调用的栈结构

 可变参数函数的实现与函数调用的栈结构密切相关，正常情况下C的函数参数入栈规则为\_\_stdcall, 它是从右到左的，即函数中的最右边的参数最先入栈。例如，对于函数：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. **void** fun(**int** a, **int** b, **int** c)
2. {
3. **int** d;
4. ...
5. }

其栈结构为

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. 0x1ffc-->d
3. 0x2000-->a
5. 0x2004-->b
7. 0x2008-->c

对于在32位系统的多数编译器，每个栈单元的大小都是sizeof(int), 而函数的每个参数都至少要占一个栈单元大小，如函数 void fun1(char a, int b, double c, short d) 对一个32的系统其栈的结构就是

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. 0x1ffc-->a  (4字节)（为了字对齐）
3. 0x2000-->b  (4字节)
5. 0x2004-->c  (8字节)
7. 0x200c-->d  (4字节)

**因此，函数的所有参数是存储在线性连续的栈空间中的，基于这种存储结构，这样就可以从可变参数函数中必须有的第一个普通参数来寻址后续的所有可变参数的类型及其值。**

先看看固定参数列表函数：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. **void** fixed\_args\_func(**int** a, **double** b, **char** \*c)
2. {
3. printf("a = 0x%p\n", &a);
4. printf("b = 0x%p\n", &b);
5. printf("c = 0x%p\n", &c);
6. }

对于固定参数列表的函数，每个参数的名称、类型都是直接可见的，他们的地址也都是可以直接得到的，比如：通过&a我们可以得到a的地址，并通过函数原型声明了解到a是int类型的。

   但是对于变长参数的函数，我们就没有这么顺利了。还好，按照C标准的说明，支持变长参数的函数在原型声明中，必须有至少一个最左固定参数(这一点与传统C有区别，传统C允许不带任何固定参数的纯变长参数函数)，这样我们可以得到其中固定参数的地址，但是依然无法从声明中得到其他变长参数的地址，比如：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. **void** var\_args\_func(**const** **char** \* fmt, ...)
2. {
3. ... ...
4. }

这里我们只能得到fmt这固定参数的地址，仅从函数原型我们是无法确定"..."中有几个参数、参数都是什么类型的。回想一下函数传参的过程，无论"..."中有多少个参数、每个参数是什么类型的，它们都和固定参数的传参过程是一样的，简单来讲都是栈操作，而栈这个东西对我们是开放的。这样一来，一旦我们知道某函数帧的栈上的一个固定参数的位置，我们完全有可能推导出其他变长参数的位置。

我们先用上面的那个fixed\_args\_func函数确定一下入栈顺序。

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. **int** main()
2. {
3. fixed\_args\_func(17, 5.40, "hello world");
4. **return** 0;
5. }
6. a = 0x0022FF50
7. b = 0x0022FF54
8. c = 0x0022FF5C

从这个结果来看，显然参数是从右到左，逐一压入栈中的(栈的延伸方向是从高地址到低地址，栈底的占领着最高内存地址，先入栈的参数，其地理位置也就最高了)。

我们基本可以得出这样一个结论：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. c.addr = b.addr + x\_sizeof(b);  /\*注意:  x\_sizeof !=sizeof \*/
2. b.addr = a.addr + x\_sizeof(a);

有了以上的"等式"，我们似乎可以推导出 void var\_args\_func(const char \* fmt, ... ) 函数中，可变参数的位置了。起码第一个可变参数的位置应该是：first\_vararg.addr = fmt.addr + x\_sizeof(fmt);  根据这一结论我们试着实现一个支持可变参数的函数：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. #include <stdarg.h>
2. #include <stdio.h>
4. **void** var\_args\_func(**const** **char** \* fmt, ...)
5. {
6. **char**    \*ap;
8. ap = ((**char**\*)&fmt) + **sizeof**(fmt);
9. printf("%d\n", \*(**int**\*)ap);
11. ap =  ap + **sizeof**(**int**);
12. printf("%d\n", \*(**int**\*)ap);
14. ap =  ap + **sizeof**(**int**);
15. printf("%s\n", \*((**char**\*\*)ap));
16. }
18. **int** main()
19. {
20. var\_args\_func("%d %d %s\n", 4, 5, "hello world");
21. **return** 0;
22. }

期待输出结果:

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. 4
2. 5
3. hello world

  先来解释一下这个程序。我们用ap获取第一个变参的地址，我们知道第一个变参是4，一个int 型，所以我们用(int\*)ap以告诉编译器，以ap为首地址的那块内存我们要将之视为一个整型来使用，\*(int\*)ap获得该参数的值；接下来的变参是5，又一个int型，其地址是ap + sizeof(第一个变参)，也就是ap + sizeof(int)，同样我们使用\*(int\*)ap获得该参数的值；最后的一个参数是一个字符串，也就是char\*，与前两个int型参数不同的是，经过ap + sizeof(int)后，ap指向栈上一个char\*类型的内存块(我们暂且称之tmp\_ptr, char \*tmp\_ptr)的首地址，即ap -> &tmp\_ptr，而我们要输出的不是printf("%s\n", ap)，而是printf("%s\n", tmp\_ptr); printf("%s\n", ap)是意图将ap所指的内存块作为字符串输出了，但是ap -> &tmp\_ptr，tmp\_ptr所占据的4个字节显然不是字符串，而是一个地址。如何让&tmp\_ptr是char \*\*类型的，我们将ap进行强制转换(char\*\*)ap <=> &tmp\_ptr，这样我们访问tmp\_ptr只需要在(char\*\*)ap前面加上一个\*即可，即printf("%s\n",  \*(char\*\*)ap);

   一切似乎很完美，编译也很顺利通过，但运行上面的代码后，不但得不到预期的结果，反而整个编译器会强行关闭（大家可以尝试着运行一下），原来是ap指针在后来并没有按照预期的要求指向第二个变参数，即并没有指向5所在的首地址，而是指向了未知内存区域，所以编译器会强行关闭。其实错误开始于：ap =  ap + sizeof(int);由于内存对齐，编译器在栈上压入参数时，不是一个紧挨着另一个的，编译器会根据变参的类型将其放到满足类型对齐的地址上的，这样栈上参数之间实际上可能会是有空隙的。（[C语言内存对齐详解（1）](http://www.cnblogs.com/cpoint/p/3369273.html) [C语言内存对齐详解（2）](http://www.cnblogs.com/cpoint/p/3369456.html) [C语言内存对齐详解（3）](http://www.cnblogs.com/cpoint/p/3369486.html)）所以此时的ap计算应该改为：ap =  (char \*)ap +sizeof(int) + \_\_va\_rounded\_size(int);

改正后的代码如下：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. #include<stdio.h>
3. #define \_\_va\_rounded\_size(TYPE)  \
4. (((**sizeof** (TYPE) + **sizeof** (**int**) - 1) / **sizeof** (**int**)) \* **sizeof** (**int**))
6. **void** var\_args\_func(**const** **char** \* fmt, ...)
7. {
8. **char** \*ap;
10. ap = ((**char**\*)&fmt) + **sizeof**(fmt);
11. printf("%d\n", \*(**int**\*)ap);
13. ap = (**char** \*)ap + **sizeof**(**int**) + \_\_va\_rounded\_size(**int**);
14. printf("%d\n", \*(**int**\*)ap);
16. ap = ap + **sizeof**(**int**) + \_\_va\_rounded\_size(**int**);
17. printf("%s\n", \*((**char**\*\*)ap));
18. }
20. **int** main()
21. {
22. var\_args\_func("%d %d %s\n", 4, 5, "hello world");
23. **return** 0;
24. }

var\_args\_func只是为了演示，并未根据fmt消息中的格式字符串来判断变参的个数和类型，而是直接在实现中写死了。

为了满足代码的可移植性，C标准库在stdarg.h中提供了诸多便利以供实现变长长度参数时使用。这里也列出一个简单的例子，看看利用标准库是如何支持变长参数的：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. #include <stdarg.h>
2. #include <stdio.h>
4. **void** std\_vararg\_func(**const** **char** \*fmt,...)
5. {
6. **va\_list** ap;
7. va\_start(ap, fmt);
9. printf("%d\n", va\_arg(ap, **int**));
10. printf("%f\n", va\_arg(ap, **double**));
11. printf("%s\n", va\_arg(ap, **char**\*));
13. va\_end(ap);
14. }
16. **int** main() {
17. std\_vararg\_func("%d %f %s\n", 4, 5.4, "hello world");        <span style="font-family: SimSun;">**return** 0;}</span>

对比一下 std\_vararg\_func和var\_args\_func的实现，va\_list似乎就是char\*， va\_start似乎就是 ((char\*)&fmt) + sizeof(fmt)，va\_arg似乎就是得到下一个参数的首地址。没错，多数平台下stdarg.h中va\_list, va\_start和var\_arg的实现就是类似这样的。一般stdarg.h会包含很多宏，看起来比较复杂。

下面我们来探讨如何写一个简单的可变参数的C 函数.

使用可变参数应该有以下步骤:   
1)首先在函数里定义一个va\_list型的变量,这里是arg\_ptr,这个变量是指向参数的指针.   
2)然后用va\_start宏初始化变量arg\_ptr,这个宏的第二个参数是第一个可变参数的前一个参数,是一个固定的参数.   
3)然后用va\_arg返回可变的参数,并赋值给整数j. va\_arg的第二个参数是你要返回的参数的类型,这里是int型.   
4)最后用va\_end宏结束可变参数的获取.然后你就可以在函数里使用第二个参数了.如果函数有多个可变参数的,依次调用va\_arg获取各个参数.

在《C程序设计语言》中，Ritchie提供了一个简易版printf函数：

**[cpp]** [view plain](https://blog.csdn.net/li_ning_/article/details/51571165) [copy](https://blog.csdn.net/li_ning_/article/details/51571165)

1. #include<stdarg.h>
3. **void** minprintf(**char** \*fmt, ...)
4. {
5. **va\_list** ap;
6. **char** \*p, \*sval;
7. **int** ival;
8. **double** dval;
10. va\_start(ap, fmt);
11. **for** (p = fmt; \*p; p++) {
12. **if**(\*p != '%') {
13. putchar(\*p);
14. **continue**;
15. }
16. **switch**(\*++p) {
17. **case** 'd':
18. ival = va\_arg(ap, **int**);
19. printf("%d", ival);
20. **break**;
21. **case** 'f':
22. dval = va\_arg(ap, **double**);
23. printf("%f", dval);
24. **break**;
25. **case** 's':
26. **for** (sval = va\_arg(ap, **char** \*); \*sval; sval++)
27. putchar(\*sval);
28. **break**;
29. **default**:
30. putchar(\*p);
31. **break**;
32. }
33. }
34. va\_end(ap);
35. }