也谈C语言变长参数 - [[**语言探索**](http://bigwhite.blogbus.com/c1381794/)]

Tag：[语言探索](http://bigwhite.blogbus.com/tag/%E8%AF%AD%E8%A8%80%E6%8E%A2%E7%B4%A2/)

[版权声明](http://creativecommons.org/licenses/by/3.0/deed.zh)：转载时请以超链接形式标明文章原始出处和作者信息及[本声明](http://bangzhuzhongxin.blogbus.com/logs/11205960.html)  
<http://bigwhite.blogbus.com/logs/20468193.html>

很多技术人员都有在"技术细节"上"钻牛角尖"的"癖好"，对此很多人褒贬不一；无论怎样，我也是属于这类人。C语言的变长参数在平时做开发时很少会在自己设计的接口中用到，但我们最常用的接口printf就是使用的变长参数接口，在感受到printf强大的魅力的同时，是否想挖据一下到底printf是如何实现的呢？这里我们一起来挖掘一下C语言变长参数的奥秘。  
  
先考虑这样一个问题：如果我们不使用C标准库(libc)中提供的Facilities，我们自己是否可以实现拥有变长参数的函数呢？我们不妨试试。  
  
一步一步进入正题，我们先看看固定参数列表函数，  
void fixed\_args\_func(int a, double b, char \*c) {  
        printf("a = 0x%p\n", &a);  
        printf("b = 0x%p\n", &b);  
        printf("c = 0x%p\n", &c);  
}  
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; 通过&b我们可以得到b的地址，并通过函数原型声明了解到b是double类型的; 通过&c我们可以得到c的地址，并通过函数原型声明了解到c是char\*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：  
void var\_args\_func(const char \* fmt, ... ) {  
    ... ...   
}  
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，自然也就无法确定其位置了。那么如何可以做到呢？在大脑中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](http://bigwhite.blogbus.com/logs/1592114.html" \t "_blank)，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，顺着这个思路，我们继续往下走，通过一个例子来诠释一下：(这里要说明的是：函数参数进栈以及参数空间地址分配都是"实现相关"的，不同平台、不同编译器都可能不同，所以下面的例子仅在IA-32，Windows XP， MinGW gcc v3.4.2下成立)  
  
我们先用上面的那个fixed\_args\_func函数确定一下这个平台下的入栈顺序。  
  
int main() {  
    fixed\_args\_func(17, 5.40, "hello world");  
    return 0;  
}  
a = 0x0022FF50  
b = 0x0022FF54  
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：  
 c.addr = b.addr + x\_sizeof(b);  /\*注意:  x\_sizeof != sizeof，后话再说 \*/  
 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：  
  
void var\_args\_func(const char \* fmt, ... ) {  
    char    \*ap;  
  
    ap = ((char\*)&fmt) + sizeof(fmt);  
    printf("%d\n", \*(int\*)ap);    
          
    ap =  ap + sizeof(int);  
    printf("%d\n", \*(int\*)ap);  
  
    ap =  ap + sizeof(int);  
    printf("%s\n", \*((char\*\*)ap));  
}  
  
int main(){  
    var\_args\_func("%d %d %s\n", 4, 5, "hello world");  
}  
  
输出结果:  
4  
5  
hello world  
  
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，如果你把这个程序拿到solaris 9下，运行后，一定得不到正确的结果，为什么呢，后续再说。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);  
  
前面说过，如果将var\_args\_func放到solaris上，一定是得不到正确结果的？为什么呢？由于[内存对齐](http://bigwhite.blogbus.com/logs/1347304.html" \t "_blank)。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。上述例子中，我是根据反编译后的汇编码得到的参数间隔，还好都是4，然后在代码中写死了。  
  
为了满足代码的可移植性，[C标准库](http://bigwhite.blogbus.com/logs/2793927.html)在stdarg.h中提供了诸多Facilities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：  
#include <stdarg.h>  
  
void std\_vararg\_func(const char \*fmt, ... ) {  
        va\_list ap;  
        va\_start(ap, fmt);  
  
        printf("%d\n", va\_arg(ap, int));  
        printf("%f\n", va\_arg(ap, double));  
        printf("%s\n", va\_arg(ap, char\*));  
  
        va\_end(ap);  
}  
  
int main() {  
        std\_vararg\_func("%d %f %s\n", 4, 5.4, "hello world");  
}  
输出:  
4  
5.400000  
hello world  
  
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。在有的系统中stdarg.h的实现依赖some special functions built into the the compilation system to handle variable argument lists and stack allocations，多数其他系统的实现与下面很相似：(Visual C++ 6.0的实现较为清晰，因为windows上的应用程序只需要在windows平台间做移植即可，没有必要考虑太多的平台情况)。  
  
Microsoft Visual Studio\VC98\Include\stdarg.h中，  
typedef char \*  va\_list;  
  
#define \_INTSIZEOF(n)   ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )  
#define va\_start(ap,v)  ( ap = (va\_list)&v + \_INTSIZEOF(v) )  
#define va\_arg(ap,t)    ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )  
#define va\_end(ap)      ( ap = (va\_list)0 )  
  
这里有两个地方需要深入挖掘一下：  
1、#define \_INTSIZEOF(n)   ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )  
我们这里简化一下这个宏：  
#define \_INTSIZEOF(n)  ((sizeof(n) + x) & ~(x))  
x = sizeof(int) - 1 = 3 = 0000 0000 0000 0011(b)  
~x = 1111 1111 1111 1100(b)  
  
当一个数 & (-x)时，得到的值始终是sizeof(int)的倍数，也就是说\_INTSIZEOF(n)的功能是将n[圆整](http://bigwhite.blogbus.com/logs/5503006.html)到sizeof(int)的倍数上去。sizeof(n) >= 1, sizeof(n)+sizeof(int)-1经过圆整后，一定会是>=4的整数；在其他系统平台上，圆整的目标值有的是4，有的则是8，视具体系统而定。  
  
2、#define va\_arg(ap,t)    ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )  
其实有了var\_args\_func的实现，这里也就不难理解了。不过这里有一个trick，很多人一开始肯定对先加上\_INTSIZEOF(t)，又减去\_INTSIZEOF(t)很不理解，其实这里是一点就透的：整个表达式((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) 返回的值其实和最初的ap所指向的地址是一致的，关键就是在整个表达式被evaluated后，ap却指向了下一个参数的地址了，就这么简单。  
  
在[P.J.Plauger](http://bigwhite.blogbus.com/logs/2853205.html)的"The standard C library"一书的第10章节中也有对stdarg实现的分析，那个版本虽然比较老，但我想应该是现有版本的一个雏形。

[使用va\_list自定义print的小例子.](http://blog.csdn.net/yao_guet/article/details/6059995)

///////////////////////////////////////////////////////////////////////////////

void print(char \*szBuf, char \*fmtStr, ...)

{

char \*fmt;

char \*cb;

char \*szTmp;

char c;

int nTmp;

unsigned int uiTmp;

cb = szBuf;

fmt = fmtStr;

va\_list ap;

va\_start(ap, fmtStr);

while(\*fmt)

{

if (\*fmt != '%')

{

\*cb = \*fmt;

cb++;

fmt++;

continue;

}

fmt++;

switch(\*fmt)

{

case 'd':

nTmp = va\_arg(ap, int);

\_itoa(nTmp, cb, 10);

while(\*++cb) ;

break;

case 'x':

uiTmp = va\_arg(ap, unsigned int);

\_itoa(uiTmp, cb, 16);

while(\*++cb) ;

break;

case 's':

szTmp = va\_arg(ap, char \*);

while(\*szTmp)

{

\*cb++ = \*szTmp;

szTmp++;

}

break;

case 'c':

c = va\_arg(ap, char);

\*cb++ = c;

break;

default:

\*cb++ = '%';

}

fmt++;

}

\*cb = 0;

va\_end(ap);

}