



# mm\_snprintf 说明

Added by 陈军, last edited by 陈军 on Jan 18, 2017

- [mm\\_snprintf 概述](#)
  - [mm\\_snprintf 和 C99 snprintf, VC++ \\_snprintf 的比较](#)
  - [其他特征](#)
- [当前 mm\\_snprintf 源码位置](#)
- [历史记录](#)
  - [\[2008-05-26\] 给 mm\\_snprintf 增加 Unicode 支持 \(v3.1\)](#)
  - [\[2014-07-13\] 添加 memory-dump 支持 \(v4.2\)](#)
  - [\[2014-10-17\] 支持 va\\_list 参数嵌套 \(v4.2\)](#)
    - [%w 用途描述](#)
    - [使用 %w, V4.4 起的写法 \(新写法, 推荐\)](#)
    - [使用 %w, V4.2 的写法 \(老式写法, 不推荐, 但仍旧支持\)](#)
    - [%w 实现者注意](#)
- [Q&A](#)
  - [Q: 要不要保留 mm\\_snprintf 这个真实函数名?](#)
  - [Q: 我们拿到的 va\\_list 能否被多次使用?](#)

## mm\_snprintf 概述

mm\_snprintf 是作为 C99 snprintf 的替代品出现的, 同时也作为 Visual C++ (6~9) \_snprintf 的替代品。

本人 (陈军) 很早发现 C99 和 VC 并没有提供相同语义的 snprintf, 而 snprintf 又是一个有责任心的 C 程序员经常要使用的函数, 如何在跨平台代码中顺畅地使用 snprintf 变成一个不大不小的难题。因此, 2004 左右, 我就在网上试图找一些 snprintf 的简洁实现, 就找到了这个 <http://www.ijs.si/software/snprintf/> (当时其版本是 2.2), 由于其作者名叫 Mark Martinec, 我将此族函数的命名前都加上了 mm\_ 前缀。

Mark Martinec 将 C99 snprintf 要求的细节实现得相当精确, 因此我很乐意接受了。不过, 还是有些功能缺失, 比如, 不支持格式化浮点数、Visual C++ 上不能格式化64位整数、不能作 Unicode 编译等。后来, 2006 的国庆节, 我找个时间将这些缺失功能补上了, 版本号被我增为 3.1; 2014 改进 PT850 USB 代码时, 意识到再给 mm\_snprintf 做一些改进, 可以很方便地用来格式化调试信息, 因此又给他加了 hexdump 和 嵌套 va\_list 功能, 成为 v4.2。

注: mm\_snprintf 内部调用了系统的 sprintf 来进行真正的数值格式化, mm\_snprintf 自身的代码来检查缓冲区长度防止溢出。因此, mm\_snprintf 并非100%意义的跨平台, 具体目标平台系统库的 sprintf 会影响其输出结果 (特别表现在浮点数格式化上)。

## mm\_snprintf 和 C99 snprintf, VC++ \_snprintf 的比较

考察各个方面后发现, 不一致的地方多如牛毛。

项目	Visual C++ 6, 8	C99(Linux)	mm_snprintf																															
结果字符串超出缓冲区, 缓冲区末尾如何处理?	末尾原本该填什么字符就填什么, 不保证以 NUL 结尾	保证以 NUL 结尾 (除非 bufsize 为0)	同 C99																															
返回值含义	<div>若缓冲区够大, 返回写入缓冲区的字符数, 该数值不包括末尾的 NUL。若缓冲区不够大, 返回 -1。注: "够大"的意思是, 够装下结果字符串本身就够了, 末尾的 NUL 是否装得下无所谓。</div> <div><pre>ret = _snprintf(buf, 4, "%s", "abcd"); // ret 得到 4, 末尾没有补 NUL</pre></div>	返回结果字符串原本应占用的字符数, 不包括结尾补的 NUL。这样的好处是, 用户就明白开多大的缓冲区 (返回值+1) 重试即可得到完整的结果字符串。	同 C99																															
64 位整数类型修饰符	%l64d, %l64u, %l64X, %l64o	%lld, %llu, %llX, %llo	同 C99																															
宽窄版本函数名差别	<table><tr><th>窄版本</th><th>宽版本</th><th>泛型</th><th>引入版本</th></tr><tr><td>_snprintf</td><td>_snwprintf</td><td>_sntprintf</td><td>VC6</td></tr><tr><td>_vsnprintf</td><td>_vsnwprintf</td><td>_vsntprintf</td><td>VC6</td></tr><tr><td>sprintf_s</td><td>swprintf_s</td><td>_stprintf_s</td><td>VS2005</td></tr></table> <div>另: swprintf 的原型在 VC6 和 VS2005 是不同的。</div>	窄版本	宽版本	泛型	引入版本	_snprintf	_snwprintf	_sntprintf	VC6	_vsnprintf	_vsnwprintf	_vsntprintf	VC6	sprintf_s	swprintf_s	_stprintf_s	VS2005	<div>glibc 2.2.x (2007)</div> <table><tr><th>窄版本</th><th>宽版本</th></tr><tr><td>snprintf</td><td>swprintf</td></tr><tr><td>vsnprintf</td><td>vswprintf</td></tr></table> <div>glibc 不提供泛型函数名。</div>	窄版本	宽版本	snprintf	swprintf	vsnprintf	vswprintf	<table><tr><th>窄版本</th><th>宽版本</th><th>泛型</th></tr><tr><td>mm_snprintfA</td><td>mm_snprintfW</td><td>mm_snprintf</td></tr><tr><td>mm_vsnprintfA</td><td>mm_vsnprintfW</td><td>mm_vsnprintf</td></tr></table>	窄版本	宽版本	泛型	mm_snprintfA	mm_snprintfW	mm_snprintf	mm_vsnprintfA	mm_vsnprintfW	mm_vsnprintf
窄版本	宽版本	泛型	引入版本																															
_snprintf	_snwprintf	_sntprintf	VC6																															
_vsnprintf	_vsnwprintf	_vsntprintf	VC6																															
sprintf_s	swprintf_s	_stprintf_s	VS2005																															
窄版本	宽版本																																	
snprintf	swprintf																																	
vsnprintf	vswprintf																																	
窄版本	宽版本	泛型																																
mm_snprintfA	mm_snprintfW	mm_snprintf																																
mm_vsnprintfA	mm_vsnprintfW	mm_vsnprintf																																
%s 和 %S 的含义差别	<ul style="list-style-type: none"><li>对于 _snprintf, %s 指示窄字符串, %S 指示宽字符串</li><li>对于 _snwprintf, %s 指示宽字符串, %S 指示窄字符串</li></ul>	不论是 snprintf 还是 swprintf, 统一用 %s 表示窄字符串、%ls 表示宽字符串	<div>只支持 %s, 窄版本函数指示窄字符串, 宽版本函数指示宽字符串。即, 无法一次函数调用混搭处理宽串与窄串。</div> <div>不支持 %S 或 %ls 。</div>																															

注:

- C99 snprintf 返回值的含意是: 假定缓冲区足够大的情况下, 格式化后的字符串的长度, 不包括结尾的NUL字符。因此, 提供 0 字节大小的缓冲区意味着让 mm\_snprintf 预先计算所需缓冲区的大小。
- 从 VC8 起, 微软的 \_snprintf 和 \_snwprintf 也支持 %ls 明确指示宽字符 (模仿 C99), 还支持 %hs, 明确指示窄字符。
- 从 VC8 起, 微软也支持 %lld、%llx 等来格式化 64-bit 整数了。
- 本人认为 %S、%ls、%hs 那些东西太混乱了, 用户绝大多数时候需要的是自然的 %s。你调宽字符版本, %s 就对应宽字符, 你调窄字符版本, %s 就对应窄字符。想混用的话, 用户应该自行先转成统一的宽窄再调用 snprintf 这族函数。
- 从 MsSDK Feb 2003 起 (VC6 可用), 微软引入了名字如 StringCchPrintfEx 的一组函数, 其最重要的作用 (我认为), 就是弥补 \_snprintf 这组老函数缓冲区不够时末尾没补 NUL 的问题。观察 StringCchPrintfEx 这组函数的源码 (在 strsafe.h 中) 可知, 它们只是很薄的一层包装函数, 格式化字符串的工作还是交给 \_vsnprintf 来进行, 因此, 这些“新函数”仍旧无法实现 C99 语义, 最重要的, 预先探知所需缓冲区大小就无法实现。

其他特征

可以使用 `mm_snprintf` 来实现安全的 `strcat`。如：

```
char *strcat_s(char *str, int strbufsize, const char* append)
{
    mm_snprintf(str, strbufsize, "%s%s", str, append)
    return str;
}
```

`str[]` 目标缓冲区按理不应和可变参数部分的"输入缓冲区"重叠，但这里有一个特殊情况是允许的，即，若格式化字符串以 `"%s"` 打头，此 `"%s"` 允许对应 `str` 本身，这即是起到连接字符串的效果。

印象中，某个早期的 `glibc` 版本（`gcc 3.2` 时代）的 `snprintf` 居然不支持这种操作。

当前 `mm_snprintf` 源码位置

[https://nlssvn/svnrepos/CommonLib/mm\\_snprintf/trunk/](https://nlssvn/svnrepos/CommonLib/mm_snprintf/trunk/)

历史记录

[2008-05-26] 给 `mm_snprintf` 增加 `Unicode` 支持 (v3.1)

- 2008底，`mm_snprintf` 已支持 `WinCE`，在 `WinCE` 上可以同时用窄字符版和宽字符版，即两者编在同一个库中。

随着 `WinCE` 的进入，现不得不考虑让 `mm_snprintf` 支持 `Unicode` 了。

目标：

- 让一个 `mm_snprintf` 库同时支持 `char*` 字符串和 `wchar_t*` 字符串。不采取嵌套实现（`mm_snprintfA` 用 `mm_snprintfW` 来实现或反之），此种实现的缺点是外层函数的效率差，因为要临时 `malloc` 内存来放内层函数的输出字符串。
- 起用 `mm_snprintfA` 和 `mm_snprintfW` 这两个名字，分别表示 `char` 型和 `wchar_t` 型函数，即原函数名尾巴加 `A` 或 `W`。

实施细则：

- `mm_snprintf.cpp` 只准在 `SVN` 库中出现一份，即不将其拷贝成 `mm_snprintfW.cpp` `check-in`，那会导致以后修改代码时的重复劳动。现策略是编译时当场拷贝一份 `mm_snprintfW.cpp`，编之。
- 格式化字符串 `"%s"`，对于 `mm_snprintfA` 就是对应 `char*` 字符串，对应 `mm_snprintfW` 就是对应 `wchar_t*` 字符串，不像 `MSVCRT` 那样 `"%S"` 反操作的名堂。

[2014-07-13] 添加 `memory-dump` 支持 (v4.2)

很多时候，我们希望将一个内存块内容以可打印的方式 `dump` 出来（常被叫作 `hexdump`），现在，`mm_snprintf` 直接提供了这个功能，免去了每次要作 `hexdump` 都要当场写一个小函数的麻烦。

此处将 `hexdump` 的功能叫作 `memory dump` (**memdump**) 是由于我给此功能分配的修饰符是 `%m` 和 `%M`，因此叫它 `memdump` 较为易记。原本想用 `%h %H`，但 `%h` 已作为 `short` 类型的修饰符，显然应该避开。

假定有如下代码框架：

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

void mprintA(const char *fmt, ...)
{
    char buf[2000];
    int bufsize = sizeof(buf);
    va_list args;
    va_start(args, fmt);
    int ret = mm_vsnprintfA(buf, bufsize, fmt, args);
    printf("%s\n", buf);
    va_end(args);
}

int main()
{
    const char *str="ABN";
    int i;
    unsigned char mem[256];
    for(i=0; i<sizeof(mem); i++) mem[i]=i;

    // Use mprintA() here.
}
```

修饰符及其用法举例如下：

修饰符（斜体字为可变的实际值）	解释
-----------------	----

%9m	<p>指定 <b>dump</b> 出 9 个字节的内容。</p> <table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%9m", mem);</pre></td><td><pre>000102030405060708</pre></td></tr><tr><td><p>若 <b>dump</b> 字节数是动态的, 可用 * 语法。</p><pre>mprintA("%*m", 9, mem);</pre></td><td><pre>000102030405060708</pre></td></tr><tr><td><p>指定内存字节数为 0 的方法:</p><pre>mprintA("%*m", 0, mem);</pre><pre>mprintA("%0m", mem);</pre></td><td>两种写法都输出空字符串。</td></tr></table>	代码	输出	<pre>mprintA("%9m", mem);</pre>	<pre>000102030405060708</pre>	<p>若 <b>dump</b> 字节数是动态的, 可用 * 语法。</p> <pre>mprintA("%*m", 9, mem);</pre>	<pre>000102030405060708</pre>	<p>指定内存字节数为 0 的方法:</p> <pre>mprintA("%*m", 0, mem);</pre> <pre>mprintA("%0m", mem);</pre>	两种写法都输出空字符串。
代码	输出								
<pre>mprintA("%9m", mem);</pre>	<pre>000102030405060708</pre>								
<p>若 <b>dump</b> 字节数是动态的, 可用 * 语法。</p> <pre>mprintA("%*m", 9, mem);</pre>	<pre>000102030405060708</pre>								
<p>指定内存字节数为 0 的方法:</p> <pre>mprintA("%*m", 0, mem);</pre> <pre>mprintA("%0m", mem);</pre>	两种写法都输出空字符串。								
%m	<p>将对应参数值看作一个 C 字符串, <b>dump</b> 字节数定为 <b>strlen(p)</b> 告知的字符串长度; 若是宽字符版本, <b>dump</b> 字节数为 <b>wcslen(p)*sizeof(wchar_t)</b> 。</p> <table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%m", str);</pre></td><td><pre>41424e</pre></td></tr></table> <p>特别提示, 此写法跟</p> <pre>mprintA("%0m", str);</pre> <p>的含义完全不同!</p>	代码	输出	<pre>mprintA("%m", str);</pre>	<pre>41424e</pre>				
代码	输出								
<pre>mprintA("%m", str);</pre>	<pre>41424e</pre>								
%M	所有特性同 %m, 只不过 hex 表达中的 a-f 变为 A-F 。								
%k	<p>%k 指定相邻两个 hex pair 之间使用的修饰符, 不使用 %k 的话, 两个 hex pair 之间没有任何的修饰字符。</p> <table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%k%5m,%M", " ", mem, str);</pre></td><td><pre>00 01 02 03 04,41 42 4E</pre></td></tr><tr><td><pre>mprintA("%k%5m", "--", mem);</pre></td><td><pre>00--01--02--03--04</pre></td></tr></table> <p>%k 的影响持续到本次 <b>mm_snprintf</b> 调用结束。</p>	代码	输出	<pre>mprintA("%k%5m,%M", " ", mem, str);</pre>	<pre>00 01 02 03 04,41 42 4E</pre>	<pre>mprintA("%k%5m", "--", mem);</pre>	<pre>00--01--02--03--04</pre>		
代码	输出								
<pre>mprintA("%k%5m,%M", " ", mem, str);</pre>	<pre>00 01 02 03 04,41 42 4E</pre>								
<pre>mprintA("%k%5m", "--", mem);</pre>	<pre>00--01--02--03--04</pre>								
%K	<p>%K 指定包围在 hex pair 两旁的修饰字符。用户仅为 %K 指定一个修饰字符串, 该字符串被分成两半, 第一半作为左侧修饰, 第二半作为右侧修饰。</p> <p>%k 和 %K 可结合使用, 同时起各自的作用, 两者的出现顺序任意。</p> <table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%K%5m", "&lt;&gt;", mem);</pre></td><td><pre>&lt;00&gt;&lt;01&gt;&lt;02&gt;&lt;03&gt;&lt;04&gt;</pre></td></tr><tr><td><pre>mprintA("%k%K%5m", "-", "&lt;&gt;", mem);</pre></td><td><pre>&lt;00&gt;-&lt;01&gt;-&lt;02&gt;-&lt;03&gt;-&lt;04&gt;</pre></td></tr></table> <p>%K 的影响持续到本次 <b>mm_snprintf</b> 调用结束。</p>	代码	输出	<pre>mprintA("%K%5m", "&lt;&gt;", mem);</pre>	<pre>&lt;00&gt;&lt;01&gt;&lt;02&gt;&lt;03&gt;&lt;04&gt;</pre>	<pre>mprintA("%k%K%5m", "-", "&lt;&gt;", mem);</pre>	<pre>&lt;00&gt;-&lt;01&gt;-&lt;02&gt;-&lt;03&gt;-&lt;04&gt;</pre>		
代码	输出								
<pre>mprintA("%K%5m", "&lt;&gt;", mem);</pre>	<pre>&lt;00&gt;&lt;01&gt;&lt;02&gt;&lt;03&gt;&lt;04&gt;</pre>								
<pre>mprintA("%k%K%5m", "-", "&lt;&gt;", mem);</pre>	<pre>&lt;00&gt;-&lt;01&gt;-&lt;02&gt;-&lt;03&gt;-&lt;04&gt;</pre>								
%r	<p>%r 指定开启 <b>memdump</b> 的多行模式。多行模式使得长度过大的内存块得以分行输出。</p> <p>%r 的对应值为 0 或未指定 %r 时, 不启用多行模式 (因为 column 数为 0 是没有意义的)。</p> <table><tr><th>代码</th><th>输出</th></tr><tr><td><p>指定每行输出 8 个字节 (columns=8)</p><pre>mprintA("%k%r%17m", " ", 8, mem);</pre></td><td><pre>00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre></td></tr></table>	代码	输出	<p>指定每行输出 8 个字节 (columns=8)</p> <pre>mprintA("%k%r%17m", " ", 8, mem);</pre>	<pre>00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre>				
代码	输出								
<p>指定每行输出 8 个字节 (columns=8)</p> <pre>mprintA("%k%r%17m", " ", 8, mem);</pre>	<pre>00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre>								

%R	%R 类似于 %r， 但加上了横纵标尺。						
	<table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%k%R%17m", " ", 8, mem);</pre></td><td><pre>-----00-01-02-03-04-05-06-07 0000: 00 01 02 03 04 05 06 07 0008: 08 09 0a 0b 0c 0d 0e 0f 0010: 10</pre></td></tr></table>	代码	输出	<pre>mprintA("%k%R%17m", " ", 8, mem);</pre>	<pre>-----00-01-02-03-04-05-06-07 0000: 00 01 02 03 04 05 06 07 0008: 08 09 0a 0b 0c 0d 0e 0f 0010: 10</pre>		
代码	输出						
<pre>mprintA("%k%R%17m", " ", 8, mem);</pre>	<pre>-----00-01-02-03-04-05-06-07 0000: 00 01 02 03 04 05 06 07 0008: 08 09 0a 0b 0c 0d 0e 0f 0010: 10</pre>						
%4r	指示全盘缩进(indent) 为 4 个空格。						
%4R	<table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%k%4r%17m", " ", 8, mem);</pre></td><td><pre>.  00 01 02 03 04 05 06 07    08 09 0a 0b 0c 0d 0e 0f    10</pre></td></tr><tr><td><pre>mprintA("%k%4R%17m", " ", 8, mem);</pre></td><td><pre>.  -----00-01-02-03-04-05-06-07    0000: 00 01 02 03 04 05 06 07    0008: 08 09 0a 0b 0c 0d 0e 0f    0010: 10</pre></td></tr></table>	代码	输出	<pre>mprintA("%k%4r%17m", " ", 8, mem);</pre>	<pre>.  00 01 02 03 04 05 06 07    08 09 0a 0b 0c 0d 0e 0f    10</pre>	<pre>mprintA("%k%4R%17m", " ", 8, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    0000: 00 01 02 03 04 05 06 07    0008: 08 09 0a 0b 0c 0d 0e 0f    0010: 10</pre>
代码	输出						
<pre>mprintA("%k%4r%17m", " ", 8, mem);</pre>	<pre>.  00 01 02 03 04 05 06 07    08 09 0a 0b 0c 0d 0e 0f    10</pre>						
<pre>mprintA("%k%4R%17m", " ", 8, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    0000: 00 01 02 03 04 05 06 07    0008: 08 09 0a 0b 0c 0d 0e 0f    0010: 10</pre>						
%0.3r	指定首行输出位置跳跃数(column skip) 为 3， 可与全盘缩进结合。						
%4.3r	<table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%k%0.3r%17m", " ", 8, mem);</pre></td><td><pre>          00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre></td></tr><tr><td><pre>indent = 4, column skip = 3, columns per line = 8 mprintA("%k%*.R%17m", " ", 4, 3, 8, mem);</pre></td><td><pre>.  -----00-01-02-03-04-05-06-07    FFFD:          00 01 02 03 04    0005: 05 06 07 08 09 0a 0b 0c    000D: 0d 0e 0f 10</pre></td></tr></table>	代码	输出	<pre>mprintA("%k%0.3r%17m", " ", 8, mem);</pre>	<pre>          00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre>	<pre>indent = 4, column skip = 3, columns per line = 8 mprintA("%k%*.R%17m", " ", 4, 3, 8, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    FFFD:          00 01 02 03 04    0005: 05 06 07 08 09 0a 0b 0c    000D: 0d 0e 0f 10</pre>
代码	输出						
<pre>mprintA("%k%0.3r%17m", " ", 8, mem);</pre>	<pre>          00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10</pre>						
<pre>indent = 4, column skip = 3, columns per line = 8 mprintA("%k%*.R%17m", " ", 4, 3, 8, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    FFFD:          00 01 02 03 04    0005: 05 06 07 08 09 0a 0b 0c    000D: 0d 0e 0f 10</pre>						
%v	<p>指定其后的 %m 所 dump 的数据的假想地址，该假想地址在 %R 引发的地址指示栏中体现。</p> <p>若不提供 %v， 则默认假想地址为 0。</p> <p>警告: %v 对应的数值必须是个指针类型，而非 int 或 unsigned int 类型。对于 64 位编译来说，这两者的宽度是不同的。</p>						
	<table><tr><th>代码</th><th>输出</th></tr><tr><td><pre>mprintA("%k%R%v%17m", " ", 8, (void*)0x1400, mem);</pre></td><td><pre>-----00-01-02-03-04-05-06-07 1400: 00 01 02 03 04 05 06 07 1408: 08 09 0a 0b 0c 0d 0e 0f 1410: 10</pre></td></tr><tr><td><pre>mprintA("%k%*.R%v%17m", " ", 4, 3, 8, (void*)0x427c00, mem);</pre></td><td><pre>.  -----00-01-02-03-04-05-06-07    427BFD:          00 01 02 03 04    427C05: 05 06 07 08 09 0a 0b 0c    427C0D: 0d 0e 0f 10</pre></td></tr></table>	代码	输出	<pre>mprintA("%k%R%v%17m", " ", 8, (void*)0x1400, mem);</pre>	<pre>-----00-01-02-03-04-05-06-07 1400: 00 01 02 03 04 05 06 07 1408: 08 09 0a 0b 0c 0d 0e 0f 1410: 10</pre>	<pre>mprintA("%k%*.R%v%17m", " ", 4, 3, 8, (void*)0x427c00, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    427BFD:          00 01 02 03 04    427C05: 05 06 07 08 09 0a 0b 0c    427C0D: 0d 0e 0f 10</pre>
代码	输出						
<pre>mprintA("%k%R%v%17m", " ", 8, (void*)0x1400, mem);</pre>	<pre>-----00-01-02-03-04-05-06-07 1400: 00 01 02 03 04 05 06 07 1408: 08 09 0a 0b 0c 0d 0e 0f 1410: 10</pre>						
<pre>mprintA("%k%*.R%v%17m", " ", 4, 3, 8, (void*)0x427c00, mem);</pre>	<pre>.  -----00-01-02-03-04-05-06-07    427BFD:          00 01 02 03 04    427C05: 05 06 07 08 09 0a 0b 0c    427C0D: 0d 0e 0f 10</pre>						

注:

- %k %K %r %R 指定的修饰与格式，其效果一直持续到 mm\_sprintf 函数返回。第二次的 mm\_sprintf 将恢复默认修饰行为。
- 若在一个 mm\_sprintf 调用中故意指定多次 %k， 第二次出现的 %k 对应的新参数将应用于其后出现的 memdump 动作。%K, %r, %R 也是类似行为。
- %v 指定的假想地址仅会生效一次。换言之，若一次 mm\_sprintf 调用中 %v 后出现过两次 %m， 第二次 %m 的假象地址复位为 0。因此，当需要指定假象地址时，格式化字符串中建议 %v 紧贴在 %m 之前。
- %r %R 助记法: 将 R 想象成 ruler。

[2014-10-17] 支持 va\_list 参数嵌套 (v4.2)

修饰符	解释
-----	----

%w	<p>指示 <code>va_list</code> 嵌套参数的出现。特别注意，<code>%w</code> 将消耗参数列表中的两个参数，而非一个。消耗的这两个参数称为“嵌套参数对”。</p> <ul style="list-style-type: none"><li>消耗的<code>第一个参数</code>指示一个格式化字符串。</li><li>消耗的<code>第二个参数</code>指示那个格式化字符串对应的参数列表对象，以 <code>va_list*</code> 类型表达。</li></ul> <p>一个参数嵌套对经过 <code>mm_snprintf</code> 的处理后将生成一个结果子串。如果将参数嵌套对的第一个参数取名为 <code>wfmt</code>，第二个参数取名为 <code>wargs</code>，那么这个子串等同于</p> <div><pre>mm_vsnprintf(buf, bufsize, wfmt, *wargs); // 注意, 此句的函数是 mm_vsnprintf ,不是 mm_snprintf</pre></div> <p>生成的子串。该子串将被输出为外层 <code>mm_snprintf</code> / <code>mm_vsnprintf</code> 输出字符串的一部分，出现在 <code>%w</code> 对应的位置。</p> <p>典型地，以下两种写法是等价的：</p> <div><pre>mm_vsnprintf(buf, bufsize, fmt, args); mm_snprintf(buf, bufsize, "%w", fmt, &amp;args); // 注意: args 前要加 &amp;</pre></div> <p><code>%w</code> 的嵌套效果是可以任意层数叠加的。</p> <p>助记：根据 <code>w</code> 的谐音，将其想象成 <code>double use parameters</code> 。</p>
----	--

%w 用途描述

想象你有一个格式化字符串并输出结果到文件的函数，名曰 `vaWriteFile`，实现为：

```
void vlWriteFile(const char * filename, const char *fmt, va_list args)
{
    int slen;
    char tbuf[4000];
    slen = mm_vsnprintf(tbuf, sizeof(tbuf), fmt, args);

    if(slen>=4000) slen = 4000-1;
    // Call OS file system APIs, some verbose code below
    HANDLE hfile = OpenFile(filename, /*...*/);
    if(hfile!=BAD_HANDLE)
    {
        int result = WriteFile(file_handle, tbuf, slen);
        if(result!=SUCCESS)
        {
            // generate some error message
        }
        CloseFile(hfile);
    }
    else
    {
        // generate some error message
    }
}

void vaWriteFile(const char * filename, const char *fmt, ...)
{
    int slen;
    char tbuf[4000];
    va_list args;
    va_start(args, fmt);
    vlWriteFile(filename, fmt, args);
    va_end(args);
}
```

稍后，你希望写一个 `vaLogToFile` 函数，用于在文件中生成一条日志信息，而且有一些附加要求：

- 希望 `vaLogToFile` 能自动前加时间戳、后加 `"\n"`。想想该函数如何编写？
- 内部希望调用 `vaWriteFile` 来进行，而非自行调用系统的文件操作函数，因为调用系统的文件操作函数需要很多行代码。

常规方法很难达到最优化的效果，你不得不在几种因素直接做出权衡。

常规写法的问题	代码
<p>常规写法一，消耗额外的内存来存放临时格式化结果。</p> <p>需要一个额外的 <code>sbuf[]</code> 数组，而且该数组的大小难以抉择。</p>	<div><pre>void vaLogToFile(const char *filename, const char *fmt, ...) {     char timebuf[40];     GetNowTime(timebuf, sizeof(timebuf));      char sbuf[4000]; // dilemma: how many to allocate?     va_list args;     va_start(args, fmt);     mm_vsnprintf(sbuf, sizeof(sbuf), fmt, args);     va_end(args);      vaWriteFile(filename, "%s%s\n", timebuf, sbuf); }</pre></div>

常规写法的问题	代码
<p>常规写法二，多次调用 <code>vaWriteFile</code>，得额外处理日志信息原子性问题。</p> <p>此方法免除了额外的 <code>sbuf[]</code> 数组，但在多线程环境中使用有不利影响，你得在 <code>vaLogToFile</code> 中加互斥量来保证每一条日志信息的三个元素（时间戳，用户内容，末尾的换行符）是连在一起的，即，不被同时调用 <code>vaLogToFile</code> 的其他线程打断。</p>	<pre>void vaLogToFile(const char *filename, const char *fmt, ...) {     MutexLock(g_some_mutex_object); // to make vaLogToFile atomic      char timebuf[40];     GetNowTime(timebuf, sizeof(timebuf));     vaWriteFile(filename, "%s", timebuf);      va_list args;     va_start(args, fmt);     vlWriteFile(filename, fmt, args);     va_end(args);     vaWriteFile(filename, "\n");      MutexUnlock(g_some_mutex_object); // to make vaLogToFile atomic }</pre>

现在，有了 `%w` 修饰符，我们就有了最好的写法，无需任何权衡。

小结一下，`%w` 使得你可以将“可变参数的函数”进行再次封装——只要你事先知道，那个“可变参数函数”内部用的是 `mm_snprintf`。

使用 `%w`，V4.4 起的写法（新写法，推荐）

分 C++ 和 C 两种写法，C++ 的较为方便且不易犯错（不易抄代码时抄错，比如漏了 `wpair` 前的 `&` 之类）。

C++	C
<pre>void vaLogToFile(const char *filename, const char *fmt, ...) {     char timebuf[40];     GetNowTime(timebuf, sizeof(timebuf));      va_list args;     va_start(args, fmt);     vaWriteFile(filename, "%s%w\n", timebuf, MM_WPAIR_PARAM(fmt, args));     va_end(args); }</pre>	<pre>void vaLogToFile(const char *filename, const char *fmt, ...) {     char timebuf[40];     va_list args;     struct mm_wpair_st wpair = { mm_wpair_magic, fmt, &amp;args };     GetNowTime(timebuf, sizeof(timebuf));      va_start(args, fmt);     vaWriteFile(filename, "%s%w\n", timebuf, &amp;wpair);     va_end(args); }</pre>

C++ 写法中，`MM_WPAIR_PARAM` 是一个宏，其展开后的内容是临时生成一个 `mm_wpair_st` 结构体对象、并取其地址传给 `vaWriteFile`。

使用 `%w`，V4.2 的写法（老式写法，不推荐，但仍旧支持）

老式写法要求一个 `%w` 对应两个数据参数，这个比较反直觉，而且容易犯错，比如忘记写 `args` 前头的 `&`（编译不会出错，运行时才崩溃）。此写法不建议使用。

<pre>void vaLogToFile(const char *filename, const char *fmt, ...) {     char timebuf[40];     GetNowTime(timebuf, sizeof(timebuf));      va_list args;     va_start(args, fmt);     vaWriteFile(filename, "%s%w\n", timebuf, fmt, &amp;args);     va_end(args); }</pre>
---

`%w` 能够兼容两种写法，是因为我在实施了一个技巧。针对 `%w` 取第一个数据参数时（取到的参数值必定是一个有效指针，不论用户采取新写法或老写法），判断其指向的 `unsigned int` 是否为预定的 `magic number`，若是，则判定为新写法。`Magic number` 现取为 `0xEF160913`，会跟这个内容撞车的字符串(`fmt` 字串)现实中几乎不可能出现。

`%w` 实现者注意

由于 `%w` 对 `va_list` 的使用方法比较另类，未见其他软件项目有类似用法，因此，每移植到一个新平台，应特别检查 `%w` 的实际效果。

目前用下面的 `test_w_specifier()` 来验证：

<pre>int print_with_prefix_suffix(TCHAR *buf, int bufsize, const TCHAR *fmt, ...) {     va_list args;     va_start(args, fmt);     int alen = mm_snprintf(buf, bufsize,         t("%c%w%s"), t('['), fmt, &amp;args, t("]")); // %w consumes two arguments     va_end(args);     return alen; }  int test_w_specifier() {     TCHAR buf[100];     int bufsize = sizeof(buf)/sizeof(buf[0]);     int alen = print_with_prefix_suffix(buf, bufsize, t("Hello '%%c' spec"), t('w')); // 15     if(alen==17 &amp;&amp; t_strcmp(buf, t("[Hello '%w' spec]"))==0)         mprint(t("test_w_specifier() ok\n"));     else     {         mprint(t("test_w_specifier() ERROR\n"));         assert(0); // 若不通过，此处 assert 失败     }     return 0; }</pre>
--

Note: V4.4 的 %w 新写法稍后更新。

## Q&A

### Q: 要不要保留 mm\_snprintf 这个真实函数名?

A: 没有保留，但允许使用。在 mm\_snprintf.h 中，mm\_snprintf 是个宏。如下：

```
#ifdef _UNICODE
# define mm_snprintf mm_snprintfW
# define mm_vsnprintf mm_vsnprintfW
#else
# define mm_snprintf mm_snprintfA
# define mm_vsnprintf mm_vsnprintfA
#endif
```

除了 mm\_snprintf，mm\_vsnprintf 也如此处理。

~~保留的好处是，已经在使用 mm\_snprintf 的二进制库不需要重新编译。~~（现在不采取此方案）

### Q: 我们拿到的 va\_list 能否被多次使用?

这个我还未 100% 肯定。试过在 Visual C++ 32bit/64bit 编译器中是可以重复使用的，其他的编译器待验证。

va\_list 重复使用的意思，如下所示：

```
void testmm(const char *szfmt, ...)
{
    char tbuf1[100] = {};
    char tbuf2[100] = {};
    char tbuf3[100] = {};

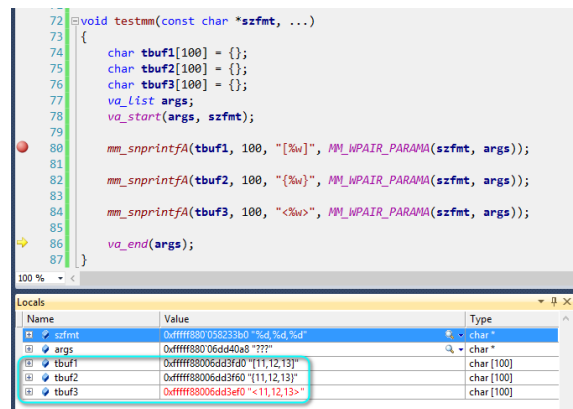
    va_list args;
    va_start(args, szfmt);

    mm_snprintfA(tbuf1, 100, "[%w]", MM_WPAIR_PARAM(szfmt, args));
    mm_snprintfA(tbuf2, 100, "{%w}", MM_WPAIR_PARAM(szfmt, args));
    mm_snprintfA(tbuf3, 100, "<%w>", MM_WPAIR_PARAM(szfmt, args));

    va_end(args);
}

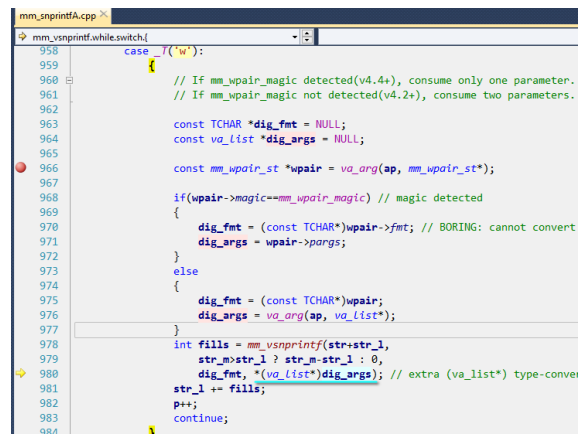
// Caller: testmm("%d,%d,%d", 11, 12, 13);
```

用 Visual C++ 编译运行，可看到 tbuf1, tbuf2, tbuf3 中都得到了正确结果：



Name	Value	Type
szfmt	0xffff80058233b0 "%d,%d,%d"	char *
args	0xffff80006dd40a0 "???"	char *
tbuf1	0xffff80006dd3fa0 "[11,12,13]"	char [100]
tbuf2	0xffff80006dd3fe0 "{11,12,13}"	char [100]
tbuf3	0xffff80006dd3ef0 "<11,12,13>"	char [100]

起初以为，“va\_list 这个对象里头有个指针要记录“当前取到参数栈上的哪个参数了”，用过一遍后，这个指针就跳走了，无法重复使用”，但实际上不会跳走，mm\_snprintf 内部嵌套调用 mm\_vsnprintf 时，va\_list 对象会被拷贝一份——下图蓝色划线处，va\_list 对象被传给 mm\_vsnprintf 时，是值传递、而非地址传递。



```
mm_vsnprintf while switch {
    case _T('w'):
        // If mm_wpair_magic detected(v4.4+), consume only one parameter.
        // If mm_wpair_magic not detected(v4.2+), consume two parameters.
        const TCHAR *dig_fmt = NULL;
        const va_list *dig_args = NULL;

        const mm_wpair_st *wpair = va_arg(ap, mm_wpair_st);

        if(wpair->magic==mm_wpair_magic) // magic detected
        {
            dig_fmt = (const TCHAR*)wpair->fmt; // BORING: cannot convert
            dig_args = wpair->pargs;
        }
        else
        {
            dig_fmt = (const TCHAR*)wpair;
            dig_args = va_arg(ap, va_list*);
        }

        int fills = mm_vsnprintf(str+str_1, str_m-str_1:0, dig_fmt, (va_list*)dig_args); // extra (va_list*) type-conver
        str_1 += fills;
        p++;
        continue;
}
```

另注： 使用 `mm_snprintf`，你无需使用 `va_copy`，这东西似乎不是 `ANSI C` 标准的东西，`Visual C++` 上似乎也没有提供 `va_copy`。

1 Child Page

 `mm_snprintf %w` 参数编译器验证列表