mm_snprintf 说明

Added by 陈军, last edited by 陈军 on Oct 07, 2017

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。其原始版本 2.2 功能其实挺少的，严重限制了使用领域。从 2006 年国庆节起，我陆陆续续给它加了很多功能。我添加的功能清点如下：

- v3.1, 支持格式化浮点数。这个需要系统的 sprintf 支持，因为我自己没有浮点数格式化算法。
- v3.1, 能格式化 64 位整数。
- v3.1, 提供 Unicode 和 MBCS 两套函数体，可以在同一个程序中混合使用。
- v4.2, 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。
- v5.0, 2017 加入千分号格式化功能。

注：mm_snprintf 内部格式化浮点数时调用了系统库的 sprintf，具体目标平台系统库给出的结果可能有所差异。 从这个意义上将，mm_snprintf 并非100%意义的跨平台，目前也仅有这个问题，已经相当好了。

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。

<table>
<tr><th>项目</th><th>Visual C++ 6 , 8</th><th>C99(Linux)</th><th>mm_snprintf</th></tr>
<tr><td>结果字符串超出缓冲区，缓冲区末尾如何处理?</td><td>末尾原本该填什么字符就填什么，不保证以 NUL 结尾</td><td>保证以 NUL 结尾（除非 bufsize 为0）</td><td>同 C99</td></tr>
<tr><td>64 位整数类型修饰符</td><td>%I64d, %I64u, %I64X, %I64o</td><td>%lld, %llu, %llX, %llo</td><td>同 C99</td></tr>
<tr><td>宽窄版本函数名差别</td><td>
<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
swprintf</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>
另: swprintf 的原型在 VC6 和 VC2005 是不同的。VC2005 的 swprintf 遵循 C99 。
</td><td>glibc 2.2.x (2007)
<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>
glibc 不提供泛型函数名。
</td><td>
<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>
</td></tr>
</table>

项目	Visual C++ 6 , 8	C99(Linux)	mm_snprintf

返回值含义	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。
	`ret = _snprintf(buf, 4, "%s", "abcd");`
	`// ret 得到 4，末尾没有补 NUL`
		snprintf 返回结果字符串理想状况下（假定缓冲区够大）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。
		但特别注意：C99 那帮人在设计 swprintf 的返回值时变态了，竟然在缓冲区不够的情况下返回 -1，但跟微软 _snwprintf 的 -1 又有所差别。若刚刚好末尾的 NUL 没地存放，swprintf 将被当其缓冲区不够而返回 -1 。
		微软 VC2005+ 也提供了 swprintf，语义跟 C99 一模一样。
			不论窄版本还是宽版本，遵循一致性原则，总是返回理想字符数（类同 C99 snprintf）。

%s 和 %S 的含义差别	• 对于 _snprintf，%s 指示窄字符串，%S 指示宽字符串 • 对于 _snwprintf，%s 指示宽字符串，%S 指示窄字符串	不论是 snprintf 还是 swprintf，统一用 %s 表示窄字符串、%ls 表示宽字符串	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。 不支持 %S 或 %ls 。

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/svnreps/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：

<table>
<tr><th>修饰符（斜体字为可变的实际值）</th><th>解释</th></tr>
<tr><td>%9m</td><td>指定 dump 出 9 个字节的内容。
<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>若 dump 字节数是动态的，可用 * 语法。<pre>mprintA("%*m", 9, mem);</pre></td><td><pre>000102030405060708</pre></td></tr>
<tr><td>指定内存字节数为 0 的方法：<pre>mprintA("%*m", 0, mem);
mprintA("%0m", mem);</pre></td><td>两种写法都输出空字符串。</td></tr>
</table></td></tr>
<tr><td>%m</td><td>将对应参数值看作一个 C 字符串，dump 字节数定为 <code>strlen(p)</code> 告知的字符串长度；若是宽字符版本，dump 字节数为 <code>wcslen(p)*sizeof(wchar_t)</code> 。
<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td><pre>mprintA("%m", str);</pre>特别提示，此写法跟
<code>mprintA("%0m", str);</code>
的含义完全不同！</td><td><pre>41424e</pre></td></tr>
</table></td></tr>
<tr><td>%M</td><td>所有特性同 %m ，只不过 hex 表达中的 a-f 变为 A-F 。</td></tr>
<tr><td>%k</td><td>%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。
<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>
%k 的影响持续到本次 mm_snprintf 调用结束。</td></tr>
</table>

<table>
<tr><th>修饰符（斜体字为可变的实际值）</th><th>解释</th></tr>
<tr><td>%K</td><td>%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。

%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。

<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td><pre>mprintA("%K%5m", "<>", mem);</pre></td><td><pre><00><01><02><03><04></pre></td></tr>
<tr><td><pre>mprintA("%k%K%5m", "-", "<>", mem);</pre></td><td><pre><00>-<01>-<02>-<03>-<04></pre></td></tr>
</table>
%K 的影响持续到本次 mm_snprintf 调用结束。</td></tr>
<tr><td>%r</td><td>%r 指定开启 memdump 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。

%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）。

<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td>指定每行输出 8 个字节 (columns=8)
<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></td></tr>
<tr><td>%R</td><td>%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></td></tr>
<tr><td>%4r

%4R</td><td>指示全盘缩进(indent) 为 4 个空格。

<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></td></tr>
<tr><td>%0.3r

%4.3r</td><td>指定首行输出位置跳跃数(column skip) 为 3 ，可与全盘缩进结合。

<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>indent = 4, column skip = 3, columns per line = 8
<pre>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></td></tr>
</table>

修饰符（斜体字为可变的实际值）	解释

%v

(updated for v7)

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。

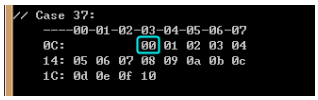
若不提供 %v ，则默认假想地址为 0 。

警示：%v 对应的数值必须是个 64 位整型数(称 Uint64)，而非 int 或 unsigned int 类型。对于 32 位程序来讲，为什么也要 64-bit 呢？思路放宽广些，你希望 hexdump 的内容可不仅仅是内存中的内容，也可能是文件的内容、或是一个长长的数据流内容，这些东西都可能超过 32 位的。

代码	输出

```
mprint(oks, t("%k%*.*R%v%17m"),
    t(" "), 4, 3, row_width, // indent=4, colskip=3
    (Uint64_mmv)0xF, // imaginary address is 0xF
    mem);
```

含义：蓝线框起的那个字节，其假想地址是 0xF 。



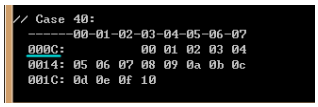
%.4v

%.4v 指定假想地址显示时，至少要补齐 4 位。

代码	输出

```
mprint(oks, t("%k%*.*R%t%.*v%17m"),
   t(" "), // %k
   2, 3, row_width, // %*.*R
   t("."),  // %t
   4, (Uint64_mmv)0xF, // %.*v
   mem // %17m
   );
```

若只用 %v , 则 mm_snprintf 认为 000C 最前头的两个 0 是多余的，用了 %4v 就会强制补零到 4 位。



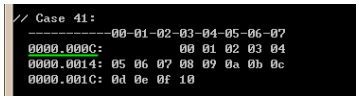
%4.8v

解释：

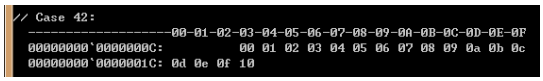
- %4.8v 中的 .8 ，仍旧解释为假想地址值“至少补齐 8 位”。
- %4.8v 中的 4 ，含义是：将 8 位显示进行分段，每段 4 位。段间的分隔符，默认为一个空格，可用 %t 来设置自定义分隔符。

代码	输出

```
mprint(oks, t("%k%*.*R%t%*.*v%17m"),
    t(" "),
    2, 3, row_width,
    t("."),  // %t
    4, 8, (Uint64_mmv)0xF, // %*.*v
    mem
    );
```



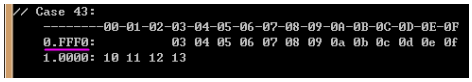

```
const int row_width_16 = 16;
mprint(oks, t("%k%*.*R%t%*.*v%17m"),
    t(" "), // %k
    2, 3, row_width_16, // %*.*R
    t("`"),  // %t
    8, 16, (Uint64_mmv)0xF, // %*.*v
    mem // %17m
    );
```



%4v

```
mprint(oks, t("%k%*.*R%t%4v%17m"), // %4v
    t(" "), // %k
    4, 3, row_width_16,
    t("."),  // %t
    (Uint64_mmv)0xFFF3, // %4v
    mem+3);
```

可以只要 %*.*v 前头的那个 * ，意即，只指定每段分隔长度，总长度由 mm_snprintf 自动确定。



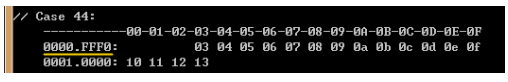
%4.0v

```
mprint(oks, t("%k%*.*R%t%4.0v%17m"), // note: %4.0v
    t(" "),
    4, 3, row_width_16,
    t("."),
    (Uint64_mmv)0xFFF3, // %4.0v
    mem+3);
```

接上一例，觉得 0.FFF0 这种样子不好看，希望在指定了分隔长度的情况下，显示的位数一定是分隔长度的整数倍。比如，

假想地址在 0xFFFF 以下，就显示四位；假想地址在 0xFFFFFFFF 以下，就显示 8 位，再超过就显示 12 位，以此类推。

需要做的，就是将 %*.*v 的第二个星号明确指定为 0 。



注：

- %k %K %r %R 指定的修饰与格式，其效果一直持续到 mm_snprintf 函数返回。第二次的 mm_snprintf 将恢复默认修饰行为。
- 若在一个 mm_snprintf 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_snprintf 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。

指定 **hexdump** 中途用的换行符

Hexdump 的字符串输出常常不只一行。多行的情况下，换行符用什么是个问题。

mm_snprintf 默认用一个 '\n' 作为换行符，但 Windows 上的很多程序对于 \n 不友好，会无视 \n ，比如 notepad.exe 、传给 edit 控件的字符串，传给 MessageBox 的字符串。

为了解决此问题，mm_snprintf 提供 **mm_set_crlf_style** 函数来让你指定它内部用的换行符风格，这是全局的，指定一次永久生效（mm_snprintf 内部的一个全局变量）。有三种选择：

- mm_crlf_lf=0 ，仅 \n ，适用于 Linux。
- mm_crlf_crlf=1 ， \r\n 两个字符，适用于 Windows 。
- mm_crlf_lf=2 ，仅 \r ，适用于 MAC 。

[2014-10-17] 支持 va_list 参数嵌套 (v4.2)

修饰符	解释

%w	指示 va_list 嵌套参数的出现。**特别注意，%w 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”。
	• 消耗的第一个参数指示一个格式化字符串。
	• 消耗的第二个参数指示那个格式化字符串对应的参数列表对象，以 `va_list*` 类型表达。
	一个参数嵌套对经过 mm_snprintf 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 wfmt, 第二个参数取名为 wargs ，那么这个子串等同于
	`mm_vsnprintf(buf, bufsize, wfmt, *wargs); // 注意，此句的函数是 mm_vsnprintf ,不是 mm_snprintf`
	生成的子串。该子串将被输出为外层 mm_snprintf / mm_vsnprintf 输出字符串的一部分，出现在 %w 对应的位置。
	典型地，以下两种写法是等价的：
	`mm_vsnprintf(buf, bufsize, fmt, args);`
	`mm_snprintf(buf, bufsize, "%w", fmt, &args); // 注意: args 前要加 &`
	%w 的嵌套效果是可以任意层数叠加的。
	助记：根据 w 的谐音，将其想象成 double use parameters 。

%w 用途描述

想象你已经有了一个格式化字符串并输出结果到文件的底层函数，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.
    // May require a mutex here, but omitted for simplicity.
    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);
}
```

可以看出，vaWriteFile 接受 sprintf 风格的可变参数，这比我们傻乎乎地直接调用 操作系统 API（Win32 WriteFile 之类）显然要方便很多。

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"。
- vaLogToFile 内部最终会调用 vaWriteFile 原子性地来输出一个日志行。一次 vaLogToFile 调用严格对应一个日志行，对应一个时间戳。

——总之意思是，借助 vaWriteFile 来封装出 vaLogToFile 。想想 vaLogToFile 如何编写？

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

常规写法的问题	代码

常规写法一，消耗额外的内存来存放临时格式化结果。
 	
需要一个额外的 sbuf[] 数组，而且该数组的大小难以抉择。
	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);
	}

常规写法二，多次调用 vaWriteFile，得额外处理日志信息原子性问题。
 	
此方法免除了额外的 sbuf[] 数组，但在多线程环境中使用有不利影响，你得在 vaLogToFile 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 vaLogToFile 的其他线程打断。
	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
	}

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++

```
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);
}
```

C

```
void vaLogToFile(const char *filename, const char *fmt, ...)
{
    char timebuf[40];
    va_list args;
    struct mm_wpair_st wpair = { mm_wpair_magic, fmt, &args };
    GetNowTime(timebuf, sizeof(timebuf));

    va_start(args, fmt);
    vaWriteFile(filename, "%s%w\n", timebuf, &wpair);
    va_end(args);
}
```

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。

```
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, &args);
    va_end(args);
}
```

%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：

```
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('['), MM_WPAIR_PARAM(fmt, 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 && 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;
}
```

[2017-02-15] 千分号格式化 (v5.0)

从 v5.0 起，所有整型（ %d, %u, %x, %lld, %llu, %llx, %o, %llo, %p ）的输出代码用自己写的了，不再借助系统的 `sprintf("%d", some_number)` 。用系统的 sprintf 本来也没啥问题，我还未遇到哪个编译器没有提供 sprintf 的，但 2017 的一个情况逼得我得自己实现这块功能：Windows 内核中的 swprintf ，不能在 DISPATCH_LEVEL 上使用，用了的话，会有极大可能触发 IRQL_NOT_LESS_OR_EQUAL(0A) 蓝屏，而且这是微软明说的，非常憋屈。 还好，格式化整型数的难度并不大，趁此机会将其实现了。这样一来，我用 mm_snprintf 来格式化我的日志信息行时，就不用在 mm_snprintfA 跟 mm_snprintfW 之间纠结了，通通都用 mm_snprintfW ，算是一大解脱。

趁这个机会，把千分号格式化功能给加上了，可用于所有整型数。

整体考虑：

- 千分号的具体字符不能定死。因为美国人跟欧洲人用的千分号字符不一样，美国人跟我们用逗号，欧洲人用小数点。 mm_snprintf 默认用空格符做千分号。
- 千分段的长度不能定死。当然，默认为 3 ，但我允许用户自定义千分段长度，以达到灵活的使用效果。比如，WinDBG 中显示一个 64-bit 指针时就以用了一种特殊的分隔符，比如 `0xffffe000`0382e240` ，用 mm_snprintf 可以方便得到这种输出。

<table>
<tr><th>修饰符</th><th>作用</th></tr>
<tr><td>%D %llD
%U %llU</td><td>带千分号来输出一个 signed 或 unsigned 整数。
<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td><pre>mprint(oks, t("[%D][%D]"), 1234, 567);</pre></td><td><pre>[1 234][567]</pre></td></tr>
</table></td></tr>
<tr><td>%T</td><td>修改千分号字符，仅针对 %D %llD %U %llU 生效。修改效果保持到一次 mm_snprintf 调用结束。千分符用一个字符串来表达，不是一个字符。用字符串的好处很明显，在 UTF8 环境中，一个字符很可能会超过一个字节。
<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td><pre>mprint(oks, t("%T[%D][%D]"), t(","), 1234, 56789);</pre></td><td><pre>[1,234][56,789]</pre></td></tr>
<tr><td><pre>__int64 i64 = 4123;
unsigned __int64 u64 = -2;
mprint(oks, t("%T[%llD][%llU]"), t(","), i64, u64);</pre></td><td><pre>[4,123][18,446,744,073,709,551,614]</pre></td></tr>
<tr><td>中途切换 %T 的值
<pre>mprint(oks, t("%T[%D]%T[%D]"), t(","), 1234, t(" "), 56789);</pre></td><td><pre>[1,234][56 789]</pre></td></tr>
</table></td></tr>
<tr><td>%t</td><td>为 %D %llD %U %llU 之外的所有整型格式指定千分号字符，效果持续到一次 mm_snprintf 调用结束。
<table>
<tr><th>代码</th><th>输出</th></tr>
<tr><td><pre>mprint(oks, t("%t[%d][%d]"), t(" "), 1234, 56789);</pre></td><td><pre>[1 234][56 789]</pre></td></tr>
<tr><td>先启用 %t, 再用空串来禁用 %t
<pre>mprint(oks, t("%t[%d]%t[%d]"), t(","), 1234, t(""), 56789);</pre></td><td><pre>[1,234][56789]</pre></td></tr>
</table></td></tr>
</table>

修饰符	作用
%_	为 %D %llD %U %llU 之外的所有整型格式指定千分段长度。

代码	输出
64 bit 机器上： `mprint(oks, t("%_%t[%p][%P]"), 8, t("`"), 0xFFFFF98008EE0D80, 0x00001FFFBFAF13D8);`	`[fffff980`08ee0d80][00001FFF`BFAF13D8]` 注: %p 使得 hex char 展现为小写，%P 展现为大写。
32 bit 机器上： `mprint(oks, t("%_%t[%p][%P]"), 4, t("."), 0xF8EE0D80, 0xBFAF13D8);`	`[f8ee.0d80][BFAF.13D8]`

千分号格式指定补零数目（此功能在 **v6.2** 时加入）

写法	输出
`mprint(oks, t("%t[%0.9u]"), t(","), 12345);` 解释： • %后紧跟的 0 表示要对 12345 前头要补 0 。 • .9 : 告知补 0 的个数, 补到总共的 digit 达到 9 位即可. 若原先的数字已有 9 位或更多, 就不用补了。 提示： 不可以写 `"[%09u]"` ， 那将达不到效果。	`[000,012,345]`
`mprint(oks, t("%t[%013.9d]"), t(","), -12345);` 解释： • %后紧跟的 0 表示要对 -12345 的前头要补 0 ， 当然，补的位置是负号和 12345 之间。 • .9 : 所谓的 precision，告知补 0 的个数, 补到总共的 digit 达到 9 位即可，数目 9 不包括前导的负号。 • 13 : 所谓的 width，告知整体的输出宽度是 13 字符，补零后还不够 13 的宽度的话，在前头填充空格。	`[ -000,012,345]` 提示：总共输出 15 字符，负号前有个空格。
`mprint(oks, t("%t[%-013.9d]"), t(","), -12345);` 和上一例类似，唯一不同是 fmtspec 中加了一个负号，表示左对齐，意即，空格补在右侧。	`[-000,012,345 ]`

`__int64 x9 = 9123456789;`
`mprint(oks, t("%t[%0*.*lld]"), t(","), 19, 14, x9);`
当然，可以用星号来动态指定 width 和 precision。
	`[ 00,009,123,456,789]`

`mprint(oks, t("%_%t[%#0.*llx]"), 4, t(" "), 18, 0xBA123456789);` 演示了补零功能可以和其他 fmtspec 维度一起使用，本例中， • # 表示额外自动加 0x 前缀。 • llx 中的 x 表示进行十六进制表达。	`[0x00 0000 0ba1 2345 6789]`

-

[2017-10-02] mm_printf_ct , 无须缓冲区, 直接指定输出目标 (v7.0)

终于了了最后一块心病，现在可以直接指定输出目标，比如，直接将格式化后的字符串写入文件、送往串口等。

没有 mm_printf_ct 之前，为了使用非内存块的输出目标，我们得用一块内存来中转，但这块内存得开多大是个问题，因此，一般得调用 mm_snprintf 两遍，第一遍查得需要的缓冲区大小、动态开辟这么大一个临时缓冲区、第二遍才将最终内容输出至该临时缓冲区，最后调用写文件、写串口函数将内容送至最终目标。

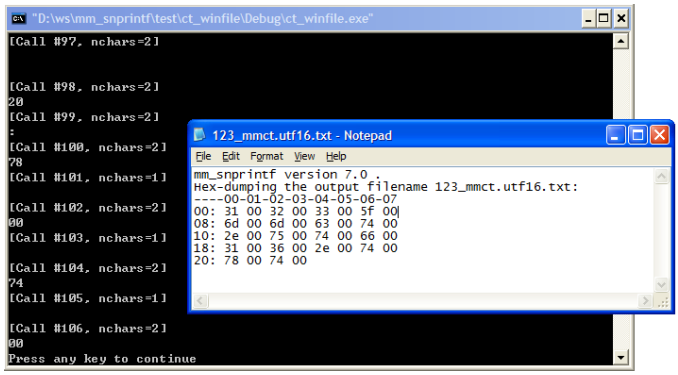
回顾前头的 vaLogToFile 和 vaWriteFile 例子，%w 消除了 vaLogToFile 需要额外开辟缓冲区的问题，但 vaWriteFile 的临时缓冲区还未消除，用 malloc 给 vaWriteFile 动态开辟 / 销毁缓冲区对运行效率很不利。

现在好了，从 v7 起，新增了 mm_snprintf_ct 函数，用户通过提供一个回调函数来一小片一小片地获知格式化后的字符串片断，当场就可以将它们送往最终目标，vaWriteFile 的缓冲区难题被解决。 _ct 后缀的含义是： custom target 。

mm_snprintf v7 自带如下例子程序，叫 ct_winfile ，它演示了调用 mm_printf_ct 来将输出字符串直接写入文件，无须临时缓冲区。

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <tchar.h>
#include <assert.h>
#include <stdio.h>
#include <locale.h>
#include <mm_snprintf.h>

struct mmct_WriteFile_st
{
    DWORD winerr;
    HANDLE hFile;
    UINT nCalls;
    UINT nChars;
};

void mmct_WriteFile(void *ctx_user, const TCHAR *pcontent, int nchars,
```

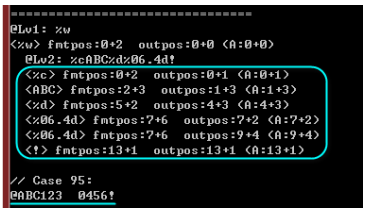
```
                    const struct mmctexi_st *pcti)
{
    // Caution: pcontent is not NUL-terminated.
    (void)pcti; // This example does not need it.
    if(nchars==0)
         return; // This can happen for %k %t %v etc

    mmct_WriteFile_st &ctx = *(mmct_WriteFile_st*)ctx_user;
    ctx.nCalls++;
    ctx.nChars += nchars;
    if(!ctx.winerr)
    {
        DWORD nwr = 0;
        BOOL b = WriteFile(ctx.hFile, pcontent, nchars*sizeof(TCHAR), &nwr, NULL);
        if(b)
        {
            mm_printf(_T("[Call #%d, nchars=%d]\n%.*s\n"),
                ctx.nCalls, nchars, nchars, pcontent);
        }
        else
        {
            ctx.winerr = GetLastError();
            mm_printf(_T("WriteFile() error: Winerr=%u."), ctx.winerr);
        }
    }
}

int _tmain()
{
    setlocale(LC_ALL, "");
    mm_set_crlf_style(mm_crlf_crlf);
#ifdef UNICODE
    TCHAR *filename = _T("123_mmct.utf16.txt");
#else
    TCHAR *filename = _T("123_mmct.ansi.txt");
#endif
    unsigned short mmver = mmsnprintf_getversion();
    int ver1 = mmver>>8, ver2 = mmver&0xff;
    mmct_WriteFile_st ctx = {0};
    ctx.hFile = CreateFile(filename,
        GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL, // no security attribute
        CREATE_ALWAYS, // create a new file
        0, // FILE_FLAG_OVERLAPPED,
        NULL);
#ifdef UNICODE
    // Write BOM mark so notepad can recognize it.
    DWORD nwr = 0;
    WriteFile(ctx.hFile, "\xFF\xFE", 2, &nwr, NULL);
#endif
    mm_printf_ct(mmct_WriteFile, &ctx,
        _T("mm_snprintf version %d.%d .\r\n"), ver1, ver2);
    mm_printf_ct(mmct_WriteFile, &ctx,
        _T("Hex-dumping the output filename %s:\r\n")
        _T("%k%R%m")
        ,
        filename,
        _T(" "), 8, filename
        );
    CloseHandle(ctx.hFile);
    return 0;
}
```



解释：

- 关键是用户自行提供一个回调函数，本例中的 mmct_WriteFile ； mm_snprintf 内部每得到一小片输出，就会回调一次用户函数，用户函数将其写入自定义的目标。
- mm_printf_ct 内部保证送出的字符流是顺序出现的，即，不会先送出后头的内容再送前头的。
- mmct 回调函数的最后一个参数 pcti 用于指示 mm_printf_ct 内部的工作进度。举例说，格式化字符串是 `"%s counts %d"` 的话，mmct 会被回调三次，第一次告知当前进展到 `%s` 了、第二次告知进展到 `counts` 了，第三次告知进展到 `%d` 了。这有什么用呢？比如，你不小心给 mm_printf_ct 传了错误的参数列表导致内部发生崩溃（访问了无效指针之类），该参数列表很长，不易定位具体的错误参数，通过 pcti 打印出进度可帮助你快速定位故障点。该参数仅提供辅助信息之用，通常情况不必理会。
- 本例故意用了 %m 进行 memdump，以此展示 mm_snprintf 回调用户函数的频次会很高，也即，每次回调给出的数据量很小，比如就一两个字符。这意味着，如果你在回调函数体中直接调用系统函数 WriteFile 的话，频繁琐碎的 WriteFile 对运行效率很不利。为了解决此问题，你应该自己准备一个固定大小的文件输出缓冲区（比如 32KB），在回调函数中先填充缓冲区，填满后再整体调用一次 WriteFile 。UFCOM vcomtest 中的 CBufferredLogfile 实现了此功能。

pcti 参数所能提供的信息，在自测程序 mmVerify → test_v7_ct() 中展示。比如，其中一个案例的屏幕输出是：



其中传达的信息是：

```
mm_printf_ct("%cABC%d%06.4d!", t('@'), 123, 456);
```

会导致 mmct 用户函数被回调六次，圆角矩形框中那六条。

为什么屏幕输出中为什么还有 @Lv1 %w 、 @Lv2 这些字样呢？因为 mmct 的回调信息（mmctexi_st 结构体）会告诉我们当前的 mmlevel （嵌套级别）。嵌套级别什么意思呢？mm_snprintf 是这样设计的，

- mm_snprintf 最初的调用者被认为是 0 级（mmlevel=0）。mm_snprintf 刚刚被调用那会儿，从 mm_snprintf 内部看，当前的 mmlevel=1；若用户指定了 mmct 回调函数，那么从回调函数中就会看到 mmctexi_st.mmlevel==1。
- %w 的出现会导致深一级的 mm_snprintf 嵌套调用，导致 mmlevel=2 。
- test_v7_ct 这块测试代码，对于所有的用例都拿 %w 给包装了一层，因此，分析到 "%cABC%d%06.4d!" 这层时，mmlevel=2 （对应屏幕显示的 Lv2）。Lv2 比 Lv1 缩进两个字符显示。

新的便利函数：用 mm_printf 替代系统的 printf

有了 custom target 功能后，我很自然地就能写一个叫 mm_printf 的函数来代替系统的 printf ，显然比 printf 方便多了，用户程序可以尽情地使用 mm_snprintf 支持的所有的 fmspec 。

mm_printf 的实现直接打包在 mm_snprintf v7 二进制库中，省得用户自己去封装了。封装代码如下，很简洁：

```
void mmct_os_printfA(void *ctx_user, const char *pcontent, int nchars,
                     const mmctexi_stA *pctexi)
{
    (void)ctx_user; (void)pctexi;
    printf("%.*s", nchars, pcontent);
}

int mm_printfA(const char *fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    mmv7_stA mmi = {0};
    mmi.proc_output = mmct_os_printfA;
    mmi.ctx_output = NULL;
    int ret = mm_vsnprintf_v7A(mmi, fmt, args);
    va_end(args);
    return ret;
}

void mmct_os_printfW(void *ctx_user, const wchar_t *pcontent, int nchars,
                     const mmctexi_stW *pctexi)
{
    (void)ctx_user; (void)pctexi;
    wchar_t szfmt[] = L"%.*s";
    szfmt[3] = mmps_wsfmt_char(); // will be "%.*S" on linux(glibc)
    wprintf(szfmt, nchars, pcontent);
}

int mm_printfW(const wchar_t *fmt, ...)
{
    va_list args;
    va_start(args, fmt);

    mmv7_stW mmi = {0};
    mmi.proc_output = mmct_os_printfW;
    mmi.ctx_output = NULL;
    int ret = mm_vsnprintf_v7W(mmi, fmt, args);

    va_end(args);
    return ret;
}
```

-

全局 fmtspec 进度调试函数

有一个问题，用户调用 mm_printf_ct 时，可借助 mmct 回调函数来诊断 fmtspec 处理进度。但用户调用 mm_snprintf() 时也想诊断 fmtspec 进度该如何是好？ mm_snprintf() 可没有参数来让用户提供一个回调函数啊。

解决方法是用一个全局函数：

```
void mm_set_DebugProgressCallback(FUNC_mm_DebugProgress *dbgproc, void *ctx_user);
```

简称此种函数为 dbgproc 好了。dbgproc 的函数格式如下：

```
void mm_DebugProgress(void *ctx_user, const char *psz_dbginfo);
```

要取消 dbgproc 回调，再调一次 mm_set_DebugProgressCallback ，两个参数都传入 NULL 即可。

用户的 dbgproc 函数要做的事就是将 psz_dbginfo 这个字符串保存起来就行，比如写入一个日志文件，事后人去检查日志文件的内容来探查先前记录的进度信息。比如，mmVerify 输出的日志文件 _mmprogressA.log 的片断如下（跟前头 pcti 那个输出有点像）：

```
[2126]  <<fmtstring>>(@lv1): time_t will overflow at UTC [%F].
[2127]  <time_t will overflow at UTC [> fmtpos:0+29 width:false preci:false stock:0(@lv1) outpos:0+29 (~0+29)
[2128]    <<fmtstring>>(@lv2): %04d-%02d-%02d %02d:%02d:%02d
[2129]    <%04d> fmtpos:0+4 width:true,4 preci:false stock:29(@lv2) outpos:0+4 (~29+4)
[2130]    <-> fmtpos:4+1 width:false preci:false stock:29(@lv2) outpos:4+1 (~33+1)
[2131]    <%02d> fmtpos:5+4 width:true,2 preci:false stock:29(@lv2) outpos:5+1 (~34+1)
[2132]    <%02d> fmtpos:5+4 width:true,2 preci:false stock:29(@lv2) outpos:6+1 (~35+1)
[2133]    <-> fmtpos:9+1 width:false preci:false stock:29(@lv2) outpos:7+1 (~36+1)
[2134]    <%02d> fmtpos:10+4 width:true,2 preci:false stock:29(@lv2) outpos:8+2 (~37+2)
[2135]    < > fmtpos:14+1 width:false preci:false stock:29(@lv2) outpos:10+1 (~39+1)
[2136]    <%02d> fmtpos:15+4 width:true,2 preci:false stock:29(@lv2) outpos:11+1 (~40+1)
[2137]    <%02d> fmtpos:15+4 width:true,2 preci:false stock:29(@lv2) outpos:12+1 (~41+1)
[2138]    <:> fmtpos:19+1 width:false preci:false stock:29(@lv2) outpos:13+1 (~42+1)
[2139]    <%02d> fmtpos:20+4 width:true,2 preci:false stock:29(@lv2) outpos:14+2 (~43+2)
[2140]    <:> fmtpos:24+1 width:false preci:false stock:29(@lv2) outpos:16+1 (~45+1)
[2141]    <%02d> fmtpos:25+4 width:true,2 preci:false stock:29(@lv2) outpos:17+1 (~46+1)
[2142]    <%02d> fmtpos:25+4 width:true,2 preci:false stock:29(@lv2) outpos:18+1 (~47+1)
[2143]  <].> fmtpos:31+2 width:false preci:false stock:0(@lv1) outpos:48+2 (~48+2)
```

-

[2017-10-02] %F: 用内嵌函数达成自定义 fmtspec 的效果 (v6.0~7.0)

想想看这样一种情况，某些时候 mm_snprintf 提供给我们的现成格式化符(format specifier)无法满足需求，我们该怎么办？比如，拿到一个 Unix epoch 时间值（ANSI C 函数 `time(NULL);` 可返回这样的一个值），要将其格式化为“年月日时分秒表达”，该怎么做？这肯定没有现成的格式化符，常规方法是，先另行准备一个缓冲区，将年月日时分秒格式化到那个缓冲区中，再用 **"%s"** 将其拷贝给 mm_snprintf 。写起来相当啰嗦，对代码密度十分不利。

现在，**"%F"** 能够帮我们达成自定义格式化符的效果。v6.0 时引入该功能，后来自己使用过程发现最初的语义有欠缺，在 6.2 和 7.0 时两次修改了其语义（打破了兼容性，抱歉）。这个 %F 的字符含义就是 **F**unction，暗示你自己得提供一个函数用于实质的格式化动作。 下面直接讲述 v7.0 的语义。

举例如下：

```c
#include <stdio.h>
#include <time.h>

int mmF_ansitime2ymdhms(void *ctx_F, mmv7_st &mmi)
{
    assert( mmi.bufsize<=0 || (mmi.buf_output && mmi.buf_output[0]==_T('\0')) );

    time_t now = *(time_t*)ctx_F;
    struct tm* ptm = gmtime(&now);
    int len = mm_snprintf_v7(mmi, t("%04d-%02d-%02d %02d:%02d:%02d"),
        ptm->tm_year+1900, ptm->tm_mon+1, ptm->tm_mday,
        ptm->tm_hour, ptm->tm_min, ptm->tm_sec
        );
    return len;
}

int main()
{
    time_t uepoch_end32 = 0x7FFFffff;
    mm_printf("time_t will overflow at UTC [%F]!\n",
        MM_FPAIR_PARAM(mmF_ansitime2ymdhms, &uepoch_end32)
        );
    return 0;
}
```

输出：

```
time_t will overflow at UTC [2038-01-19 03:14:07]!
```

可以看出：%F 需要配合一个自定义函数使用，该函数的原型规定如下：

```c
int MyFormatter(void *ctx_formatter, mmv7_st &mmi);
```

注解：

- ctx_formatter 是 mm_snprintf 调用者自己给出的，将原原本本回传给 MyFormatter 以作上下文。如果 MyFormatter 需要很复杂的输入参数，你可以将复杂参数打包成一个结构体，将结构体地址传入。
- Myformatter 会得到一组由 mm_snprintf 内部传出来的信息，这组信息告知了 mm_snprintf 内部的当前上下文。

```c
struct mmv7_st
{
    FUNC_mmct_output *proc_output; // If NULL, callee should fill .buf_output[]
    void *ctx_output;

    TCHAR *buf_output;
    mmbufsize_t bufsize;

    int mmlevel; // debugging purpose
    int nchars_stock; // only meaningful to %F callee
    bool suppress_dbginfo; // internal use, user should set 0
};
```

这里头的结构体成员有点多，稍后解释。就 mmF_ansitime2ymdhms 这个例子本身来说，mmv7_st 这个结构体被当作了一个透明的对象（透明，指无须关心其内部细节）直接传给了一个新函数 **mm_snprintf_v7** 。 mm_snprintf_v7 其实是个核心函数， mm_snprintf, mm_vsnprintf, mm_printf_ct, mm_print 其实都是 **mm_snprintf_v7** 的外围封装。mm_ 一族函数的用户平时一般无须直接调用 mm_snprintf_v7，挑一个好用的外围封装函数来用即可，但用到 %F 时，很可能要跟 mm_snprintf_v7 打交道了。本例调

用 mm_snprintf_v7 产生的效果很容易猜测，即是得到 "%04d-%02d-%02d %02d:%02d:%02d" 这样一个年月日时分秒格式的字符串。没错，就是这样，你是本例中 mm_snprintf 的用户， 而且，你为 %F 准备的回调函数，其内部的字符串输出动作又要再借助 mm_snprintf 来达成，这时的最佳实践就是调用 **mm_snprintf_v7** 来进行你的嵌套格式化动作，mmv7_st 这个结构体自然地会告知内层的 **mm_snprintf_v7** 有多少缓冲区可用。

- MyFormatter 应该返回一个字符计数，告知自己消耗了几个字符（假定缓冲区够的话），不包括末尾的 NUL 。该回调的返回值被定义为“理想填充字符数”而非“实际填充字符数”，其用意很明确，这是为了最外层的 mm_snprintf 能够准确报告理想填充字符数的总和，这样一来，最外层的调用者就能够重试开辟一个足够大的缓冲区来再次调用 mm_snprintf 。
- MyFormatter 拿到的 bufsize 有可能 ≤ 0，在这种情况下，MyFormatter 绝对不可以往 buf[] 中填充任何字符。为什么 MyFormatter 拿到的 bufsize 有可能 ≤0 呢？因为 mm_snprintf 一贯的行为特征是：在实际缓冲区消耗完毕的情况下还会在假想缓冲区中进行虚拟格式化动作、以便报告完整格式化需要的字符数，%F 也没有逃脱这个例外。当 bufsize<=0 时，buf 一定指向的是假想缓冲区中的某个位置。按照本例，内层调用 **mm_snprintf_v7** 的话，当然不必担心缓冲区溢出问题，因为 mm_snprintf_v7 内部自身会有防御机制。

另，MyFormatter 中可用多次调用 mm_snprintf_v7 来“追加”内容。比如，上头的 mmF_ansitime2ymdhms 可改用如下的 mmF_ansitime2ymdhms_twice ，效果相同。

```
int mmF_ansitime2ymdhms_twice(void *ctx_F, mmv7_st &mmi)
{
    // Call mm_snprintf_v7 *twice*
    //
    time_t now = *(time_t*)ctx_F;
    struct tm* ptm = gmtime(&now);
    int len1 = mm_snprintf_v7(mmi, t("%04d-%02d-%02d"),
        ptm->tm_year+1900, ptm->tm_mon+1, ptm->tm_mday);
    //
    // mmi.buf_output, mmi.bufsize and mmi.nchars_stock all get changed
    // after a mm_snprintf_v7 call.
    //
    int len2 = mm_snprintf_v7(mmi, t(" %02d:%02d:%02d"),
        ptm->tm_hour, ptm->tm_min, ptm->tm_sec);
    //
    return len1+len2;
}
```

-

拆解 mmv7_st，引出 MyFormatter 的另一种实现方法

倘若你的 MyFormatter 不愿意用嵌套 mm_snprintf 的手段来输出字符串（有点像），那么你得自行处理两样事情：

1. 若 mmv7_st.proc_output 有值，就回调该函数，传递给它真正的输出字符串。 mmv7_st.proc_output 有值，意味着最外层用户调用了 mm_printf_ct 来进行 custom target 输出，此时，最外层的用户很可能并未提供一个缓冲区，因此，MyFormatter 得自行准备缓冲区。
2. 若 mmv7_st.bufsize>0 ，将真正的输出字符串给填到 mmv7_st.buf_output 里头。

显然，mm_snprintf_v7 内部会自动帮我们处理以上两样事情，用 mm_snprintf_v7 能够省了我们不少事。

现在举例，若我们非要避开 mm_snprintf_v7 ，那 %F 的回调函数得如下写：

```
int mmF_ansitime2ymdhms_method2(void *ctx_F, mmv7_st &mmi)
{
    time_t now = *(time_t*)ctx_F;
    struct tm* ptm = gmtime(&now);

    // We have to cope with two things:
    // 1. Call mmi.proc_output with mmi.ctx_output
    // 2. Fill mmi_buf_output if mmi.bufsize>0

    const int tbsize = 80;
    TCHAR timebuf[tbsize]; // please ensure this temp buffer is large enough
    int ideal_len = mm_snprintf(timebuf, tbsize,
        t("%04d-%02d-%02d %02d:%02d:%02d"),
        ptm->tm_year+1900, ptm->tm_mon+1, ptm->tm_mday,
        ptm->tm_hour, ptm->tm_min, ptm->tm_sec);
    assert(ideal_len<tbsize);

    if(mmi.proc_output)
    {
        // should call custom-output with real content
        mmi.proc_output(mmi.ctx_output, timebuf, ideal_len, NULL);
    }


    if(mmi.bufsize>0)
    {
        memcpy(mmi.buf_output, timebuf, sizeof(TCHAR)*_MIN_(ideal_len, mmi.bufsize));
    }

    return ideal_len; // 无须自己更新 mmi 里头的 .bufsize 等成员。
}
```

-

[2017-10-07] %n 连续输出某个字符串 n 次 (v7.0)

这个格式化符很简单，同时也相当便民。

很多时候，我们希望连续输出/打印 n 个空格，这个 n 是运行时动态指定的，用 snprintf 的话，又得自己额外准备一个临时缓冲区、让代码变得啰嗦.。这下好了，用 mm_printf 的话，如下写即可。

```
int n = 4;
mm_printf("%*n", n, " ");
```

详细行为看测试用例：

```
void test_v7_fmtspec_n()
{
    oks = t("[###]");
    mprint(oks, t("[%*n]"), 3, t("#"));

    oks = t("[-+-+-+-+]");
    mprint(oks, t("[%*n]"), 4, t("-+"));

    oks = t("");
    mprint(oks, t("%*n"), 222, t(""));

    oks = t("");
    mprint(oks, t("%*n"), 222, NULL); // use NULL

    oks = t("");
    mprint(oks, t("%*n"), 0, t("xxx"));
}
```

-

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）

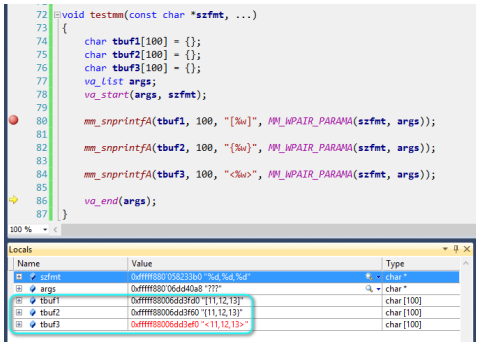
Q: 我们拿到的 va_list 能否被多次使用？

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。

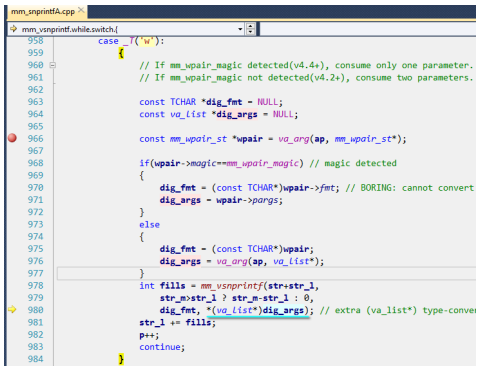
va_list 重复使用的意思，如下所示：

```
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_PARAMA(szfmt, args));
    mm_snprintfA(tbuf2, 100, "{%w}", MM_WPAIR_PARAMA(szfmt, args));
    mm_snprintfA(tbuf3, 100, "<%w>", MM_WPAIR_PARAMA(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：



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。



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