**C++字符串完全指引之一 —— Win32 字符编码**

原著：Michael Dunn

翻译：Chengjie Sun

原文出处：CodeProject：The Complete Guide to C++ Strings, Part I

**引言**

　　毫无疑问，我们都看到过像 TCHAR, std::string, BSTR 等各种各样的字符串类型，还有那些以 \_tcs 开头的奇怪的宏。你也许正在盯着显示器发愁。本指引将总结引进各种字符类型的目的，展示一些简单的用法，并告诉您在必要时，如何实现各种字符串类型之间的转换。

　　在第一部分，我们将介绍3种字符编码类型。了解各种编码模式的工作方式是很重要的事情。即使你已经知道一个字符串是一个字符数组，你也应该阅读本部分。一旦你了解了这些，你将对各种字符串类型之间的关系有一个清楚地了解。

　　在第二部分，我们将单独讲述string类，怎样使用它及实现他们相互之间的转换。

**字符基础 -- ASCII, DBCS, Unicode**

　　所有的 string 类都是以C-style字符串为基础的。C-style 字符串是字符数组。所以我们先介绍字符类型。这里有3种编码模式对应3种字符类型。第一种编码类型是**单子节字符集**（single-byte character set or **SBCS**）。在这种编码模式下，**所有的字符都只用一个字节表示**。**ASCII是SBCS**。**一个字节表示的0用来标志SBCS字符串的结束**。

　　第二种编码模式是**多字节字符集**（multi-byte character set or **MBCS**）。**一个MBCS编码包含一些一个字节长的字符，而另一些字符大于一个字节的长度**。用在Windows里的MBCS包含两种字符类型，单字节字符（single-byte characters）和双字节字符（double-byte characters）。由于Windows里使用的多字节字符绝大部分是两个字节长，所以MBCS常被用DBCS代替。

　　在DBCS编码模式中，一些特定的值被保留用来表明他们是双字节字符的一部分。例如，在Shift-JIS编码中（一个常用的日文编码模式），0x81-0x9f之间和 0xe0-oxfc之间的值表示"这是一个双字节字符，下一个子节是这个字符的一部分。"这样的值被称作"leading bytes",他们都大于0x7f。跟随在一个leading byte子节后面的字节被称作"trail byte"。在DBCS中，trail byte可以是任意非0值。像SBCS一样，**DBCS字符串的结束标志也是一个单字节表示的0**。

　　第三种编码模式是**Unicode**。**Unicode是一种所有的字符都使用两个字节编码的编码模式**。Unicode字符有时也被称作**宽字符**，因为它比单子节字符宽（使用了更多的存储空间）。注意，**Unicode不能被看作MBCS。MBCS的独特之处在于它的字符使用不同长度的字节编码。Unicode字符串使用两个字节表示的0作为它的结束标志**。

　　单字节字符包含拉丁文字母表，accented characters及ASCII标准和DOS操作系统定义的图形字符。双字节字符被用来表示东亚及中东的语言。Unicode被用在COM及Windows NT操作系统内部。

　　你一定已经很熟悉单字节字符。当你使用char时，你处理的是单字节字符。双字节字符也用char类型来进行操作（这是我们将会看到的关于双子节字符的很多奇怪的地方之一）。Unicode字符用**wchar\_t**来表示。Unicode字符和字符串常量用**前缀L**来表示。例如：

wchar\_t wch = L''1''; // 2 bytes, 0x0031

wchar\_t\* wsz = L"Hello"; // 12 bytes, 6 wide characters

**字符在内存中是怎样存储的**

**单字节字符串**：每个字符占一个字节，按顺序依次存储，最后以单字节表示的0结束。例如。"Bob"的存贮形式如下：

42 6F 62 00

B o b EOS（End Of String）

Unicode的存储形式，L"Bob"

42 00 6F 00 62 00 00 00

B o b EOS

使用两个字节表示的0来做结束标志。

　　一眼看上去，DBCS 字符串很像 SBCS 字符串，但是我们一会儿将看到 DBCS 字符串的微妙之处，它使得使用字符串操作函数和用字符指针遍历一个字符串时会产生预料之外的结果。字符串" " ("nihongo")在内存中的存储形式如下（LB和TB分别用来表示 leading byte 和 trail byte）

93 FA 96 7B 8C EA 00

LB TB LB TB LB TB EOS

EOS

值得注意的是，"ni"的值不能被解释成WORD型值0xfa93，而应该看作两个值93和fa以这种顺序被作为"ni"的编码。

**使用字符串处理函数**

　　我们都已经见过C语言中的字符串函数，strcpy(), sprintf(), atoll()等。这些字符串只应该用来处理单字节字符字符串。标准库也提供了仅适用于Unicode类型字符串的函数，比如wcscpy(), swprintf(), wtol()等。

　　微软还在它的CRT(C runtime library)中增加了操作DBCS字符串的版本。Str\*\*\*()函数都有对应名字的DBCS版本\_mbs\*\*\*()。如果你料到可能会遇到DBCS字符串（如果你的软件会被安装在使用DBCS编码的国家，如中国，日本等，你就可能会），你应该使用\_mbs\*\*\*()函数，因为他们也可以处理SBCS字符串。（一个DBCS字符串也可能含有单字节字符，这就是为什么\_mbs\*\*\*()函数也能处理SBCS字符串的原因）

　　让我们来看一个典型的字符串来阐明为什么需要不同版本的字符串处理函数。我们还是使用前面的Unicode字符串 L"Bob"：

42 00 6F 00 62 00 00 00

B o b EOS

　　因为x86 CPU是little-endian，值0x0042在内存中的存储形式是42 00。你能看出如果这个字符串被传给strlen()函数会出现什么问题吗？它将先看到第一个字节42，然后是00，而00是字符串结束的标志，于是strlen()将会返回1。如果把"Bob"传给wcslen()，将会得出更坏的结果。wcslen()将会先看到0x6f42，然后是0x0062，然后一直读到你的缓冲区的末尾，直到发现00 00结束标志或者引起了GPF。

　　到目前为止，我们已经讨论了str\*\*\*()和wcs\*\*\*()的用法及它们之间的区别。Str\*\*\*()和\_mbs\*\*()之间的有区别呢？明白他们之间的区别，对于采用正确的方法来遍历DBCS字符串是很重要的。下面，我们将先介绍字符串的遍历，然后回到str\*\*\*()与\_mbs\*\*\*()之间的区别这个问题上来。

**正确的遍历和索引字符串**

　　因为我们中大多数人都是用着SBCS字符串成长的，所以我们在遍历字符串时，常常使用指针的++和--操作。我们也使用数组下标的表示形式来操作字符串中的字符。**这两种方式是用于SBCS和Unicode字符串，因为它们中的字符有着相同的宽度**，编译器能正确的返回我们需要的字符。

　　然而，当碰到DBCS字符串时，我们必须抛弃这些习惯。这里有使用指针遍历DBCS字符串时的两条规则。违背了这两条规则，你的程序就会存在DBCS有关的bugs。

**1．在前向遍历时，不要使用++操作，除非你每次都检查lead byte；**

**2．永远不要使用--操作进行后向遍历。**

　　我们先来阐述规则2，因为找到一个违背它的真实的实例代码是很容易的。假设你有一个程序在你自己的目录里保存了一个配置文件，你把安装目录保存在注册表中。在运行时，你从注册表中读取安装目录，然后合成配置文件名，接着读取该文件。假设，你的安装目录是C:\Program Files\MyCoolApp，那么你合成的文件名应该是C:\Program Files\MyCoolApp\config.bin。当你进行测试时，你发现程序运行正常。

　　现在，想象你合成文件名的代码可能是这样的：

bool GetConfigFileName ( char\* pszName, size\_t nBuffSize )

{

char szConfigFilename[MAX\_PATH];

// Read install dir from registry... we'll assume it succeeds.

// Add on a backslash if it wasn't present in the registry value.

// First, get a pointer to the terminating zero.

char\* pLastChar = strchr ( szConfigFilename, '''' );

// Now move it back one character.

pLastChar--;

if ( \*pLastChar != ''\\'' )

strcat ( szConfigFilename, "\\" );

// Add on the name of the config file.

strcat ( szConfigFilename, "config.bin" );

// If the caller''s buffer is big enough, return the filename.

if ( strlen ( szConfigFilename ) >= nBuffSize )

return false;

else

{

strcpy ( pszName, szConfigFilename );

return true;

}

}

　　这是一段很健壮的代码，然而在遇到 DBCS 字符时它将会出错。让我们来看看为什么。假设一个日本用户使用了你的程序，把它安装在 C:\。下面是这个名字在内存中的存储形式：

43 3A 5C 83 88 83 45 83 52 83 5C 00

　 　 　 LB TB LB TB LB TB LB TB

C : \ EOS

　　当使用 GetConfigFileName() 检查尾部的''\\''时，它寻找安装目录名中最后的非0字节，看它是等于''\\''的，所以没有重新增加一个''\\''。结果是代码返回了错误的文件名。

　　哪里出错了呢？看看上面两个被用蓝色高量显示的字节。斜杠''\\''的值是0x5c。'' ''的值是83 5c。上面的代码错误的读取了一个 trail byte，把它当作了一个字符。

　　正确的后向遍历方法是使用能够识别DBCS字符的函数，使指针移动正确的字节数。下面是正确的代码。（指针移动的地方用红色标明）

bool FixedGetConfigFileName ( char\* pszName, size\_t nBuffSize )

{

char szConfigFilename[MAX\_PATH];

// Read install dir from registry... we''ll assume it succeeds.

// Add on a backslash if it wasn''t present in the registry value.

// First, get a pointer to the terminating zero.

char\* pLastChar = \_mbschr ( szConfigFilename, '''' );

// Now move it back one double-byte character.

pLastChar = CharPrev ( szConfigFilename, pLastChar );

if ( \*pLastChar != ''\\'' )

\_mbscat ( szConfigFilename, "\\" );

// Add on the name of the config file.

\_mbscat ( szConfigFilename, "config.bin" );

// If the caller''s buffer is big enough, return the filename.

if ( \_mbslen ( szInstallDir ) >= nBuffSize )

return false;

else

{

\_mbscpy ( pszName, szConfigFilename );

return true;

}

}

　　上面的函数使用CharPrev() API使pLastChar向后移动一个字符，这个字符可能是两个字节长。在这个版本里，if条件正常工作，因为lead byte永远不会等于0x5c。

　　让我们来想象一个违背规则1的场合。例如，你可能要检测一个用户输入的文件名是否多次出现了'':''。如果，你使用++操作来遍历字符串，而不是使用CharNext()，你可能会发出不正确的错误警告如果恰巧有一个trail byte它的值的等于'':''的值。

与规则2相关的关于字符串索引的规则：

2a. 永远不要使用减法去得到一个字符串的索引。

违背这条规则的代码和违背规则2的代码很相似。例如，

char\* pLastChar = &szConfigFilename [strlen(szConfigFilename) - 1];

这和向后移动一个指针是同样的效果。

回到关于str\*\*\*()和\_mbs\*\*\*()的区别

　　现在，我们应该很清楚为什么\_mbs\*\*\*()函数是必需的。Str\*\*\*()函数根本不考虑DBCS字符，而\_mbs\*\*\*()考虑。如果，你调用strrchr("C:\\ ", ''\\'')，返回结果可能是错误的，然而\_mbsrchr()将会认出最后的双字节字符，返回一个指向真的''\\''的指针。

　　关于字符串函数的最后一点：str\*\*\*()和\_mbs\*\*\*()函数认为字符串的长度都是以char来计算的。所以，如果一个字符串包含3个双字节字符，\_mbslen()将会返回6。Unicode函数返回的长度是按wchar\_t来计算的。例如，wcslen(L"Bob")返回3。

**Win32 API中的MBCS和Unicode**

两组 APIs：

　　尽管你也许从来没有注意过，Win32中的每个与字符串相关的API和message都有两个版本。一个版本接受MBCS字符串，另一个接受Unicode字符串。例如，根本没有SetWindowText()这个API，相反，有SetWindowTextA()和SetWindowTextW()。后缀A表明这是MBCS函数，后缀W表示这是Unicode版本的函数。

　　当你 build 一个 Windows 程序，你可以选择是用 MBCS 或者 Unicode APIs。如果，你曾经用过VC向导并且没有改过预处理的设置，那表明你用的是MBCS版本。那么，既然没有 SetWindowText() API，我们为什么可以使用它呢？winuser.h头文件包含了一些宏，例如：

BOOL WINAPI SetWindowText**A** ( HWND hWnd, LPCSTR lpString ); **//MBCS版本**

BOOL WINAPI SetWindowText**W** ( HWND hWnd, LPCWSTR lpString ); **//Unicode版本**

#ifdef UNICODE **//定义了UNICODE**

#define SetWindowText SetWindowTextW

#else **//没有定义UNICODE,即定义了MBCS（VC++环境的缺省设置）**

#define SetWindowText SetWindowTextA

#endif

当使用MBCS APIs来build程序时，UNICODE没有被定义，所以预处理器看到：

#define SetWindowText SetWindowTextA

　　这个宏定义把所有对SetWindowText的调用都转换成真正的API函数SetWindowTextA。（当然，你可以直接调用SetWindowTextA() 或者 SetWindowTextW()，虽然你不必那么做。）

　　所以，如果你想把默认使用的API函数变成Unicode版的，你可以在预处理器设置中，把\_MBCS从预定义的宏列表中删除，然后添加**UNICODE和\_UNICODE。(你需要两个都定义，因为不同的头文件可能使用不同的宏。)** 然而，如果你用char来定义你的字符串，你将会陷入一个尴尬的境地。考虑下面的代码：

HWND hwnd = GetSomeWindowHandle();

char szNewText[] = "we love Bob!"; //ANSI版本的字符串

SetWindowText ( hwnd, szNewText );

在预处理器把SetWindowText用SetWindowTextW来替换后，代码变成：

HWND hwnd = GetSomeWindowHandle();

char szNewText[] = "we love Bob!"; //ANSI版本的字符串

SetWindowTextW ( hwnd, szNewText );

　　看到问题了吗？我们把单字节字符串传给了一个以Unicode字符串做参数的函数。解决这个问题的第一个方案是使用 #ifdef 来包含字符串变量的定义：

HWND hwnd = GetSomeWindowHandle();

#ifdef UNICODE

**wchar\_t** szNewText[] = L"we love Bob!"; //wchar\_t为Unicode字符

#else

char szNewText[] = "we love Bob!"; //char为单字节字符，即ANSI

#endif

SetWindowText ( hwnd, szNewText );

你可能已经感受到了这样做将会使你多么的头疼。完美的解决方案是使用TCHAR.

**使用TCHAR**

　　TCHAR是一种字符串类型，它让你在以MBCS和UNNICODE来build程序时可以使用同样的代码，不需要使用繁琐的宏定义来包含你的代码。TCHAR的定义如下：

#ifdef UNICODE

typedef wchar\_t **TCHAR**;

#else //ifdef MBCS

typedef char **TCHAR**;

#endif

所以**用MBCS来build时，TCHAR是char，使用UNICODE时，TCHAR是wchar\_t**。还有一个宏来处理定义Unicode字符串常量时所需的L前缀。

#ifdef UNICODE

#define **\_T**(x) L##x

#else

#define **\_T**(x) x

#endif

　　##是一个预处理操作符，它可以把两个参数连在一起。如果你的代码中需要字符串常量，在它前面加上\_T宏。如果你使用Unicode来build，它会在字符串常量前加上L前缀。

**TCHAR** szNewText[] = **\_T**("we love Bob!");

　　像是用宏来隐藏SetWindowTextA/W的细节一样，还有很多可以供你使用的宏来实现str\*\*\*()和\_mbs\*\*\*()等字符串函数。例如，你可以使用\_tcsrchr宏来替换strrchr()、\_mbsrchr()和wcsrchr()。\_tcsrchr根据你预定义的宏是\_MBCS还是UNICODE来扩展成正确的函数，就像SetWindowText所作的一样。

　　不仅str\*\*\*()函数有TCHAR宏。其他的函数如， \_stprintf（代替sprinft()和swprintf()）,\_tfopen（代替fopen()和\_wfopen()）。 MSDN中"Generic-Text Routine Mappings."标题下有完整的宏列表。

**字符串和TCHAR typedefs**

　　由于Win32 API文档的函数列表使用函数的常用名字（例如，"SetWindowText"），所有的字符串都是用TCHAR来定义的。（除了XP中引入的只适用于Unicode的API）。下面列出一些常用的typedefs，你可以在msdn中看到他们。

type Meaning in MBCS builds Meaning in Unicode builds

WCHAR wchar\_t wchar\_t

LPSTR zero-terminated string of char (char\*) zero-terminated string of char (char\*)

LPCSTR constant zero-terminated string of char (const char\*) constant zero-terminated string of char (const char\*)

LPWSTR zero-terminated Unicode string (wchar\_t\*) zero-terminated Unicode string (wchar\_t\*)

LPCWSTR constant zero-terminated Unicode string (const wchar\_t\*) constant zero-terminated Unicode string (const wchar\_t\*)

TCHAR char wchar\_t

LPTSTR zero-terminated string of TCHAR (TCHAR\*) zero-terminated string of TCHAR (TCHAR\*)

LPCTSTR constant zero-terminated string of TCHAR (const TCHAR\*) constant zero-terminated string of TCHAR (const TCHAR\*)

**何时使用 TCHAR 和 Unicode**

　　到现在，你可能会问，我们为什么要使用Unicode。我已经用了很多年的char。下列3种情况下，使用Unicode将会使你受益：

1．你的程序只运行在Windows NT系统中。

2． 你的程序需要处理超过MAX\_PATH个字符长的文件名。

3． 你的程序需要使用XP中引入的只有Unicode版本的API.

　　Windows 9x 中大多数的 API 没有实现 Unicode 版本。所以，如果你的程序要在windows 9x中运行，你必须使用MBCS APIs。然而，由于**NT系统内部都使用Unicode**，所以使用Unicode APIs将会加快你的程序的运行速度。**每次，你传递一个字符串调用MBCS API，操作系统会把这个字符串转换成Unicode字符串，然后调用对应的Unicode API**。如果一个字符串被返回，操作系统还要把它转变回去。尽管这个转换过程被高度优化了，但它对速度造成的损失是无法避免的。

　　只要你使用Unicode API，NT系统允许使用非常长的文件名（突破了MAX\_PATH的限制，MAX\_PATH=260）。使用Unicode API的另一个优点是你的程序会自动处理用户输入的各种语言。所以一个用户可以输入英文，中文或者日文，而你不需要额外编写代码去处理它们。

　　最后，随着windows 9x产品的淡出，微软似乎正在抛弃MBCS APIs。例如，包含两个字符串参数的SetWindowTheme() API只有Unicode版本的。使用Unicode来build你的程序将会简化字符串的处理，你不必在MBCS和Unicdoe之间相互转换。

即使你现在不使用Unicode来build你的程序，你也应该使用TCHAR及其相关的宏。这样做不仅可以的代码可以很好地处理DBCS，而且如果将来你想用Unicode来build你的程序，你只需要改变一下预处理器中的设置就可以实现了。

**C++字符串完全指引之二 —— 字符串封装类**

**引言**

　　因为C语言风格的字符串容易出错且不易管理，黑客们甚至利用可能存在的缓冲区溢出bug把C语言风格的字符串作为攻击目标，所以出现了很多字符串封装类。不幸的是，在某些场合下我们不知道该使用哪个字符串类，也不知道怎样把一个C风格的字符串转换成一个字符串封装类。

　　这篇文章将介绍所有在Win32 API, MFC, STL, WTL 和 Visual C++ 运行库中出现的字符串类型。我将描述每一个类的用法，告诉大家怎样创建每一个类的对象以及怎样把一个类转换成其他类。

　　为了更好的从这篇文章中受益，你必须要明白不同的字符类型和编码，这些内容我在第一部分中介绍过。

**Rule #1 of string classes**

**使用cast来实现类型转换是不好的做法，除非有文档明确指出这种转换可以使用**。

促使我写这两篇文章的原因是字符串类型转换中经常遇到的一些问题。当我们使用cast把字符串从类型X转换到类型Z的时候，我们不知道为什么代码不能正常工作。各种各样的字符串类型，尤其是BSTR，几乎没有在任何一个地方的文档中被明确的指出可以用cast来实现类型转换。所以我想一些人可能会使用cast来实现类型转换并希望这种转换能够正常工作。

　　除非源字符串是一个被明确指明支持转换操作符的字符串包装类，否则cast不对字符串做任何转换。对常量字符串使用cast不会起到任何作用，所以下面的代码：

void SomeFunc ( LPCWSTR widestr );

main()

{

SomeFunc ( (LPCWSTR) "C:\\foo.txt" ); // WRONG!

}

　　肯定会失败。它可以被编译，因为**cast操作会撤消编译器的类型检查**。但是，编译可以通过并不能说明代码是正确的。

　　在下面的例子中，我将会指明cast在什么时候使用是合法的。

**C-style strings and typedefs**

　　正如我在第一部分中提到的，windows APIs 是用TCHARs来定义的，在编译时，它可以根据你是否定义\_MBCS或者\_UNICODE被编译成MBCS或者Unicode字符。你可以参看第一部分中对TCHAR的完整描述，这里为了方便，我列出了字符的typedefs

Type Meaning

WCHAR Unicode character (wchar\_t)

TCHAR MBCS or Unicode character, depending on preprocessor settings

LPSTR string of char (char\*)

LPCSTR constant string of char (const char\*)

LPWSTR string of WCHAR (WCHAR\*)

LPCWSTR constant string of WCHAR (const WCHAR\*)

LPTSTR string of TCHAR (TCHAR\*)

LPCTSTR constant string of TCHAR (const TCHAR\*)

　　一个增加的字符类型是OLETYPE。它表示自动化接口（如word提供的可以使你操作文档的接口）中使用的字符类型。这种类型一般被定义成wchar\_t，然而如果你定义了OLE2ANSI预处理标记，OLECHAR将会被定义成char类型。我知道现在已经没有理由定义OLE2ANSI（从MFC3以后，微软已经不使用它了），所以从现在起我将把OLECHAR当作Unicode字符。

这里给出你将会看到的一些OLECHAR相关的typedefs：

Type Meaning

OLECHAR Unicode character (wchar\_t)

LPOLESTR string of OLECHAR (OLECHAR\*)

LPCOLESTR constant string of OLECHAR (const OLECHAR\*)

　　还有两个用于包围字符串和字符常量的宏定义，它们可以使同样的代码被用于MBCS和Unicode builds ：

Type Meaning

\_T(x) Prepends **L** to the literal in Unicode builds.

OLESTR(x) Prepends L to the literal to make it an LPCOLESTR.

　　在文档或例程中，你还会看到好多\_T的变体。有四个等价的宏定义，它们是**TEXT**, \_**TEXT**, \_\_**TEXT**和**\_\_T**，它们都起同样的做用。

**COM 中的字符串 —— BSTR 和 VARIANT**

　　很多自动化和COM接口使用BSTR来定义字符串。BSTRs中有几个"陷阱"，所以这里我用单独的部分来说明它。

　　BSTR 是 Pascal-style 字符串（字符串长度被明确指出）和C-style字符串（字符串的长度要通过寻找结束符来计算）的混合产物。一个BSTR是一个Unicode字符串，它的长度是预先考虑的，并且它还有一个0字符作为结束标记。下面是一个BSTR的示例：

06 00 00 00 42 00 6F 00 62 00 00 00

--length-- B o b EOS

　　注意字符串的长度是如何被加到字符串数据中的。**长度是DWORD类型的，保存了字符串中包含的字节数，但不包括结束标记**。在这个例子中，"Bob"包含3个Unicode字符（不包括结束符），总共6个字节。字符串的长度被预先存储好，以便当一个BSTR在进程或者计算机之间被传递时，COM库知道多少数据需要传送。（另一方面，一个BSTR能够存储任意数据块，而不仅仅是字符，它还可以包含嵌入在数据中的0字符。然而，由于这篇文章的目的，我将不考虑那些情况）。

　　在 C++ 中，一个 BSTR 实际上就是一个指向字符串中第一个字符的指针。它的定义如下：

BSTR bstr = NULL;

bstr = SysAllocString ( L"Hi Bob!" );

if ( NULL == bstr )

// out of memory error

// Use bstr here...

SysFreeString ( bstr );

自然的，各种各样的BSTR封装类为你实现内存管理。

　　另外一个用在自动化接口中的变量类型是**VARIANT**。它被用来在无类型（typeless）语言，如Jscript和VBScript，来传递数据。一个VARIANT可能含有很多不同类型的数据，例如long和IDispatch\*。当一个VARIANT包含一个字符串，字符串被存成一个BSTR。当我后面讲到VARIANT封装类时，我会对VARIANT多些介绍。

**字符串封装类**

　　到目前为止，我已经介绍了各种各样的字符串。下面，我将说明封装类。对于每个封装类，我将展示怎样创建一个对象及怎样把它转换成一个C语言风格的字符串指针。C语言风格的字符串指针对于API的调用，或者创建一个不同的字符串类对象经常是必需的。我不会介绍字符串类提供的其他操作，比如排序和比较。

　　重复一遍，除非你确切的明白结果代码将会做什么，否则不要盲目地使用cast来实现类型转换。

**CRT提供的类**

**\_bstr\_t**

　　\_bstr\_t是一个对BSTR的完整封装类，实际上它隐藏了底层的BSTR。它提供各种构造函数和操作符来访问底层的C语言风格的字符串。然而，\_bstr\_t却没有访问BSTR本身的操作符，所以一个\_bstr\_t类型的字符串不能被作为输出参数传给一个COM方法。如果你需要一个BSTR\*参数，使用ATL类CComBSTR是比较容易的方式。

　　一个\_bstr\_t字符串能够传给一个接收参数类型为BSTR的函数，只是因为下列3个条件同时满足。首先，\_bstr\_t有一个向wchar\_t\*转换的转换函数；其次，对编译器而言，因为BSTR的定义，wchar\_t\*和BSTR有同样的含义；第三，\_bstr\_t内部含有的wchar\_t\*指向一片按BSTR的形式存储数据的内存。所以，即使没有文档说明，\_bstr\_t可以转换成BSTR，这种转换仍然可以正常进行。

// Constructing

\_bstr\_t bs1 = "char string"; // construct from a LPCSTR

\_bstr\_t bs2 = L"wide char string"; // construct from a LPCWSTR

\_bstr\_t bs3 = bs1; // copy from another \_bstr\_t

\_variant\_t v = "Bob";

\_bstr\_t bs4 = v; // construct from a \_variant\_t that has a string

// Extracting data

LPCSTR psz1 = bs1; // automatically converts to MBCS string

LPCSTR psz2 = (LPCSTR) bs1; // cast OK, same as previous line

LPCWSTR pwsz1 = bs1; // returns the internal Unicode string

LPCWSTR pwsz2 = (LPCWSTR) bs1; // cast OK, same as previous line

BSTR bstr = bs1.copy(); // copies bs1, returns it as a BSTR

// ...

SysFreeString ( bstr );

　　注意\_bstr\_t也提供char\*和wchar\_t\*之间的转换操作符。这是一个值得怀疑的设计，因为即使它们是非常量字符串指针，你也一定不能使用这些指针去修改它们指向的缓冲区的内容，因为那将破坏内部的BSTR结构。

**\_variant\_t**

　　\_variant\_t是一个对VARIANT的完整封装，它提供很多构造函数和转换函数来操作一个VARIANT可能包含的大量的数据类型。这里，我将只介绍与字符串有关的操作。

// Constructing

\_variant\_t v1 = "char string"; // construct from a LPCSTR

\_variant\_t v2 = L"wide char string"; // construct from a LPCWSTR

\_bstr\_t bs1 = "Bob";

\_variant\_t v3 = bs1; // copy from a \_bstr\_t object

// Extracting data

\_bstr\_t bs2 = v1; // extract BSTR from the VARIANT

\_bstr\_t bs3 = (\_bstr\_t) v1; // cast OK, same as previous line

注意：

　　如果类型转换不能被执行，\_variant\_t方法能够抛出异常，所以应该准备捕获\_com\_error异常。

还需要注意的是：

　　没有从一个\_variant\_t变量到一个MBCS字符串的直接转换。你需要创建一个临时的\_bstr\_t变量，使用提供Unicode到MBCS转换的另一个字符串类或者使用一个ATL转换宏。

　　不像\_bstr\_t，一个\_variant\_t变量可以被直接作为参数传递给一个COM方法。\_variant\_t

　　继承自VARIANT类型，所以传递一个\_variant\_t来代替VARIANT变量是C++语言所允许的。

**STL 类**

　　STL只有一个字符串类，basic\_string。一个basic\_string管理一个以0做结束符的字符数组。字符的类型是basic\_string模般的参数。总的来说，一个basic\_string类型的变量应该被当作不透明的对象。你可以得到一个指向内部缓冲区的只读指针，但是任何写操作必须使用basic\_string的操作符和方法。

　　basic\_string有两个预定义的类型：包含char的string类型和包含wchar\_t的wstring类型。这里没有内置的包含TCHAR的类型，但是你可以使用下面列出的代码来实现。

// Specializations

typedef basic\_string tstring; // string of TCHARs

// Constructing

string str = "char string"; // construct from a LPCSTR

wstring wstr = L"wide char string"; // construct from a LPCWSTR

tstring tstr = \_T("TCHAR string"); // construct from a LPCTSTR

// Extracting data

LPCSTR psz = str.**c\_str**(); // **read-only** pointer to str''s buffer

LPCWSTR pwsz = wstr.c\_str(); // read-only pointer to wstr''s buffer

LPCTSTR ptsz = tstr.c\_str(); // read-only pointer to tstr''s buffer

　　不像\_bstr\_t，一个basic\_string变量不能在字符集之间直接转换。然而，你可以传递由c\_str()返回的指针给另外一个类的构造函数（如果这个类的构造函数接受这种字符类型）。例如：

// Example, construct \_bstr\_t from basic\_string

\_bstr\_t bs1 = str.c\_str(); // construct a \_bstr\_t from a LPCSTR

\_bstr\_t bs2 = wstr.c\_str(); // construct a \_bstr\_t from a LPCWSTR

**ATL 类**

CComBSTR

　　CComBSTR 是 ATL 中的 BSTR 封装类，它在某些情况下比\_bstr\_t有用的多。最引人注意的是CComBSTR允许访问底层的BSTR，这意味着你可以传递一个CComBSTR对象给COM的方法。CComBSTR对象能够替你自动的管理BSTR的内存。例如，假设你想调用下面这个接口的方法：

// Sample interface:

struct IStuff : public IUnknown

{

// Boilerplate COM stuff omitted...

STDMETHOD(SetText)(BSTR bsText);

STDMETHOD(GetText)(BSTR\* pbsText);

};

　　CComBSTR有一个操作符--BSTR方法，所以它能直接被传给SetText()函数。还有另外一个操作--&，这个操作符返回一个BSTR\*。所以，你可以对一个CComBSTR对象使用&操作符，然后把它传给需要BSTR\*参数的函数。

CComBSTR bs1;

CComBSTR bs2 = "new text";

pStuff->GetText ( &bs1 ); // ok, takes address of internal BSTR

pStuff->SetText ( bs2 ); // ok, calls BSTR converter

pStuff->SetText ( (BSTR) bs2 ); // cast ok, same as previous line

　　CComBSTR有和\_bstr\_t相似的构造函数，然而却没有内置的向MBCS字符串转换的函数。因此，你需要使用一个ATL转换宏。

// Constructing

CComBSTR bs1 = "char string"; // construct from a LPCSTR

CComBSTR bs2 = L"wide char string"; // construct from a LPCWSTR

CComBSTR bs3 = bs1; // copy from another CComBSTR

CComBSTR bs4;

bs4.LoadString ( IDS\_SOME\_STR ); // load string from string table

// Extracting data

BSTR bstr1 = bs1; // returns internal BSTR, but don''t modify it!

BSTR bstr2 = (BSTR) bs1; // cast ok, same as previous line

BSTR bstr3 = bs1.Copy(); // copies bs1, returns it as a BSTR

BSTR bstr4;

bstr4 = bs1.Detach(); // bs1 no longer manages its BSTR

// ...

SysFreeString ( bstr3 );

SysFreeString ( bstr4 );

　　注意在上个例子中使用了Detach()方法。调用这个方法后，CComBSTR对象不再管理它的BSTR字符串或者说它对应的内存。这就是bstr4需要调用SysFreeString()的原因。

　　做一个补充说明：重载的&操作符意味着在一些STL容器中你不能直接使用CComBSTR变量，比如list。容器要求&操作符返回一个指向容器包含的类的指针，但是对CComBSTR变量使用&操作符返回的是BSTR\*，而不是CComBSTR\*。然而，有一个ATL类可以解决这个问题，这个类是CAdapt。例如，你可以这样声明一个CComBSTR的list：

std::list< CAdapt > bstr\_list;

　　CAdapt提供容器所需要的操作符，但这些操作符对你的代码是透明的。你可以把一个bstr\_list当作一个CComBSTR的list来使用。

**CComVariant**

　　CComVariant是VARIANT的封装类。然而，不像\_variant\_t，在CComVariant中VARIANT没有被隐藏。事实上你需要直接访问VARIANT的成员。CComVariant提供了很多构造函数来对VARIANT能够包含的多种类型进行处理。这里，我将只介绍和字符串相关的操作。

// Constructing

CComVariant v1 = "char string"; // construct from a LPCSTR

CComVariant v2 = L"wide char string"; // construct from a LPCWSTR

CComBSTR bs1 = "BSTR bob";

CComVariant v3 = (BSTR) bs1; // copy from a BSTR

// Extracting data

CComBSTR bs2 = v1.bstrVal; // extract BSTR from the VARIANT

　　不像\_variant\_t，这里没有提供针对VARIANT包含的各种类型的转换操作符。正如上面介绍的，你必须直接访问VARIANT的成员并且确保这个VARIANT变量保存着你期望的类型。如果你需要把一个CComVariant类型的数据转换成一个BSTR类型的数据，你可以调用ChangeType()方法。

CComVariant v4 = ... // Init v4 from somewhere

CComBSTR bs3;

if ( SUCCEEDED( v4.ChangeType ( VT\_BSTR ) ))

bs3 = v4.bstrVal;

　　像\_variant\_t一样，CComVariant也没有提供向MBCS字符串转换的转换操作。你需要创建一个\_bstr\_t类型的中间变量，使用提供从Unicode到MBCS转换的另一个字符串类，或者使用一个ATL的转换宏。

**ATL转换宏**

　　ATL：转换宏是各种字符编码之间进行转换的一种很方便的方式，在函数调用时，它们显得非常有用。ATL转换宏的名称是根据下面的模式来命名的[源类型]2[新类型]或者[源类型]2C[新类型]。据有第二种形式的名字的宏的转换结果是常量指针（对应名字中的"C"）。各种类型的简称如下：

A: MBCS string, char\* (A for ANSI)

W: Unicode string, wchar\_t\* (W for wide)

T: TCHAR string, TCHAR\*

OLE: OLECHAR string, OLECHAR\* (in practice, equivalent to W)

BSTR: BSTR (used as the destination type only)

　　所以，W2A()宏把一个Unicode字符串转换成一个MBCS字符串。T2CW()宏把一个TCHAR字符串转转成一个Unicode字符串常量。

　　为了使用这些宏，需要先包含atlconv.h头文件。你甚至可以在非ATL工程中包含这个头文件来使用其中定义的宏，因为这个头文件独立于ATL中的其他部分，不需要一个\_Module全局变量。当你在一个函数中使用转换宏时，需要把USES\_CONVERSION宏放在函数的开头。它定义了转换宏所需的一些局部变量。

　　当转换的目的类型是除了BSTR以外的其他类型时，被转换的字符串是存在栈中的。所以，如果你想让字符串的生命周期比当前的函数长，你需要把这个字符串拷贝到其他的字符串类中。当目的类型是BSTR时，内存不会自动被释放，你必须把返回值赋给一个BSTR变量或者一个BSTR封装类以避免内存泄漏。

　　下面是一些各种转换宏的使用例子：

// Functions taking various strings:

void Foo ( LPCWSTR wstr );

void Bar ( BSTR bstr );

// Functions returning strings:

void Baz ( BSTR\* pbstr );

#include

main()

{

using std::string;

USES\_CONVERSION; // declare locals used by the ATL macros

// Example 1: Send an MBCS string to Foo()

LPCSTR psz1 = "Bob";

string str1 = "Bob";

Foo ( A2CW(psz1) );

Foo ( A2CW(str1.c\_str()) );

// Example 2: Send a MBCS and Unicode string to Bar()

LPCSTR psz2 = "Bob";

LPCWSTR wsz = L"Bob";

BSTR bs1;

CComBSTR bs2;

bs1 = A2BSTR(psz2); // create a BSTR

bs2.Attach ( W2BSTR(wsz) ); // ditto, assign to a CComBSTR

Bar ( bs1 );

Bar ( bs2 );

SysFreeString ( bs1 ); // free bs1 memory

// No need to free bs2 since CComBSTR will do it for us.

// Example 3: Convert the BSTR returned by Baz()

BSTR bs3 = NULL;

string str2;

Baz ( &bs3 ); // Baz() fills in bs3

str2 = W2CA(bs3); // convert to an MBCS string

SysFreeString ( bs3 ); // free bs3 memory

}

　　正如你所看见的，当你有一个和函数所需的参数类型不同的字符串时，使用这些转换宏是非常方便的。

**MFC类**

CString

　　因为一个MFC CString类的对象包含**TCHAR类型的字符，所以确切的字符类型取决于你所定义的预处理符号**。大体来说，CString 很像STL string，这意味着你必须把它当成不透明的对象，**只能使用CString提供的方法来修改CString对象**。CString有一个string所不具备的优点：CString具有接收MBCS和Unicode两种字符串的构造函数，它还有一个LPCTSTR转换符，所以你可以把CString对象直接传给一个接收LPCTSTR的函数而不需要调用c\_str()函数。

// Constructing

CString s1 = "char string"; // construct from a LPCSTR

CString s2 = **L**"wide char string"; // construct from a LPCWSTR

CString s3 ( '' '', 100 ); // pre-allocate a 100-byte buffer, fill with spaces

CString s4 = "New window text";

// You can pass a CString in place of an **LPCTSTR**:

SetWindowText ( hwndSomeWindow, s4 );

// Or, equivalently, explicitly cast the CString:

SetWindowText ( hwndSomeWindow, (LPCTSTR) s4 );

　　你可以从你的字符串表中装载一个字符串，CString的一个构造函数和LoadString()函数可以完成它。Format()方法能够从字符串表中随意的读取一个具有一定格式的字符串。

// Constructing/loading from string table

CString s5 ( (LPCTSTR) IDS\_SOME\_STR ); // load from string table

CString s6, s7;

// Load from string table.

s6.LoadString ( IDS\_SOME\_STR );

// Load printf-style format string from the string table:

s7.Format ( IDS\_SOME\_FORMAT, "bob", nSomeStuff, ... );

　　第一个构造函数看起来有点奇怪，但是这实际上是文档说明的装入一个字符串的方法。 注意，对一个CString变量，你可以使用的唯一合法转换符是LPCTSTR。转换成LPTSTR（非常量指针）是错误的。养成把一个CString变量转换成LPTSTR的习惯将会给你带来伤害，因为当你的程序后来崩溃时，你可能不知道为什么，因为你到处都使用同样的代码而那时它们都恰巧正常工作。正确的得到一个指向缓冲区的非常量指针的方法是调用GetBuffer()方法。下面是正确的用法的一个例子，这段代码是给一个列表控件中的项设定文字：

CString str = \_T("new text");

LVITEM item = {0};

item.mask = LVIF\_TEXT;

item.iItem = 1;

item.pszText = (LPTSTR)(LPCTSTR) str; // WRONG!

item.pszText = str.GetBuffer(0); // correct

ListView\_SetItem ( &item );

str.ReleaseBuffer(); // return control of the buffer to str

　　pszText成员是一个LPTSTR变量，一个非常量指针，因此你需要对str调用GetBuffer()。GetBuffer()的参数是你需要CString为缓冲区分配的最小长度。如果因为某些原因，你需要一个可修改的缓冲区来存放1K TCHARs，你需要调用GetBuffer(1024)。把0作为参数时，GetBuffer()返回的是指向字符串当前内容的指针。

　　上面划线的语句可以被编译，在这种情况下，甚至可以正常起作用。但这并不意味着这行代码是正确的。通过使用非常量转换，你已经破坏了面向对象的封装，并对CString的内部实现作了某些假定。如果你有这样的转换习惯，你终将会陷入代码崩溃的境地。你会想代码为什么不能正常工作了，因为你到处都使用同样的代码而那些代码看起来是正确的。

　　你知道人们总是抱怨现在的软件的bug是多么的多吗？软件中的bug是因为程序员写了不正确的代码。难道你真的想写一些你知道是错误的代码来为所有的软件都满是bug这种认识做贡献吗？花些时间来学习使用CString的正确方法让你的代码在任何时间都正常工作把。

　　CString 有两个函数来从一个 CString 创建一个 BSTR。它们是 AllocSysString() 和SetSysString()。

// Converting to BSTR

CString s5 = "Bob!";

BSTR bs1 = NULL, bs2 = NULL;

bs1 = s5.AllocSysString();

s5.SetSysString ( &bs2 );

SysFreeString ( bs1 );

SysFreeString ( bs2 );

**COleVariant**

　　COleVariant和CComVariant.很相似。COleVariant继承自VARIANT，所以它可以传给接收VARIANT的函数。然而，不像CComVariant，COleVariant只有一个LPCTSTR构造函数。没有对LPCSTR 和LPCWSTR的构造函数。在大多数情况下这不是一个问题，因为不管怎样你的字符串很可能是LPCTSTRs，但这是一个需要意识到的问题。COleVariant还有一个接收CString参数的构造函数。

// Constructing

CString s1 = \_T("tchar string");

COleVariant v1 = \_T("Bob"); // construct from an LPCTSTR

COleVariant v2 = s1; // copy from a CString

　　像CComVariant一样，你必须直接访问VARIANT的成员。如果需要把VARIANT转换成一个字符串，你应该使用ChangeType()方法。然而，COleVariant::ChangeType()如果失败会抛出异常，而不是返回一个表示失败的HRESULT代码。

// Extracting data

COleVariant v3 = ...; // fill in v3 from somewhere

BSTR bs = NULL;

try

{

v3.ChangeType ( VT\_BSTR );

bs = v3.bstrVal;

}

catch ( COleException\* e )

{

// error, couldn''t convert

}

SysFreeString ( bs );

**WTL 类**

CString

　　WTL的CString的行为和MFC的 CString完全一样，所以你可以参考上面关于MFC的 CString的介绍。

**CLR 和 VC 7 类**

　　System::String是用来处理字符串的.NET类。在内部，一个String对象包含一个不可改变的字符串序列。任何对String对象的操作实际上都是返回了一个新的String对象，因为原始的对象是不可改变的。String的一个特性是如果你有不止一个String对象包含相同的字符序列，它们实际上是指向相同的对象的。相对于C++的使用扩展是增加了一个新的字符串常量前缀S，S用来代表一个受控的字符串常量（a managed string literal）。

// Constructing

String\* ms = S"This is a nice managed string";

　　你可以传递一个非受控的字符串来创建一个String对象，但是样会比使用受控字符串来创建String对象造成效率的微小损失。这是因为所有以S作为前缀的相同的字符串实例都代表同样的对象，但这对非受控对象是不适用的。下面的代码清楚地阐明了这一点：

String\* ms1 = S"this is nice";

String\* ms2 = S"this is nice";

String\* ms3 = L"this is nice";

Console::WriteLine ( ms1 == ms2 ); // prints true

Console::WriteLine ( ms1 == ms3); // prints false

正确的比较可能没有使用S前缀的字符串的方法是使用String::CompareTo()

Console::WriteLine ( ms1->CompareTo(ms2) );

Console::WriteLine ( ms1->CompareTo(ms3) );

　　上面的两行代码都会打印0，0表示两个字符串相等。 String和MFC 7 CString之间的转换是很容易的。CString有一个向LPCTSTR的转换操作，而String有两个接收char\* 和 wchar\_t\*的构造函数，因此你可以把一个CString变量直接传给一个String的构造函数。

CString s1 ( "hello world" );

String\* s2 ( s1 ); // copy from a CString

反方向的转换也很类似

String\* s1 = S"Three cats";

CString s2 ( s1 );

　　这也许会使你感到一点迷惑，但是它确实是起作用的。因为从VS.NET 开始，CString 有了一个接收String 对象的构造函数。

CStringT ( System::String\* pString );

对于一些快速操作，你可能想访问底层的字符串：

String\* s1 = S"Three cats";

Console::WriteLine ( s1 );

const \_\_wchar\_t \_\_pin\* pstr = PtrToStringChars(s1);

for ( int i = 0; i < wcslen(pstr); i++ )

(\*const\_cast<\_\_wchar\_t\*>(pstr+i))++;

Console::WriteLine ( s1 );

　　PtrToStringChars()返回一个指向底层字符串的const \_\_wchar\_t\* ，我们需要固定它，否则垃圾收集器或许会在我们正在管理它的内容的时候移动了它。

**在 printf-style 格式函数中使用字符串类**

　　当你在printf()或者类似的函数中使用字符串封装类时你必须十分小心。这些函数包括sprintf()和它的变体，还有TRACE和ATLTRACE宏。因为这些函数没有对添加的参数的类型检查，你必须小心，只能传给它们C语言风格的字符串指针，而不是一个完整的字符串类。

　　例如，要把一个\_bstr\_t 字符串传给ATLTRACE()，你必须使用显式转换(LPCSTR) 或者(LPCWSTR)：

\_bstr\_t bs = L"Bob!";

ATLTRACE("The string is: %s in line %d\n", (LPCSTR) bs, nLine);

　　如果你忘了使用转换符而把整个\_bstr\_t对象传给了函数，将会显示一些毫无意义的输出，因为\_bstr\_t保存的内部数据会全部被输出。

**所有类的总结**

　　两个字符串类之间进行转换的常用方式是：先把源字符串转换成一个C语言风格的字符串指针，然后把这个指针传递给目的类型的构造函数。下面这张表显示了怎样把一个字符串转换成一个C语言风格的字符串指针以及哪些类具有接收C语言风格的字符串指针的构造函数。

Class string type convert to char\*? convert to const char\*? convert to wchar\_t\*? convert to const wchar\_t\*? convert to BSTR? construct from char\*? construct from wchar\_t\*?

\_bstr\_t BSTR yes cast1 yes cast yes cast1 yes cast yes2 yes yes

\_variant\_t BSTR no no no cast to

\_bstr\_t3 cast to

\_bstr\_t3 yes yes

string MBCS no yes c\_str() method no no no yes no

wstring Unicode no no no yes c\_str() method no no yes

CComBSTR BSTR no no no yes cast to BSTR yes cast yes yes

CComVariant BSTR no no no yes4 yes4 yes yes

CString TCHAR no6 in MBCS

builds, cast no6 in Unicode

builds, cast no5 yes yes

COleVariant BSTR no no no yes4 yes4 in MBCS

**builds in Unicode**

builds

1、即使 \_bstr\_t 提供了向非常量指针的转换操作符，修改底层的缓冲区也会已引起GPF如果你溢出了缓冲区或者造成内存泄漏。

2、\_bstr\_t 在内部用一个 wchar\_t\* 来保存 BSTR，所以你可以使用 const wchar\_t\* 来访问BSTR。这是一个实现细节，你可以小心的使用它，将来这个细节也许会改变。

3、如果数据不能转换成BSTR会抛出一个异常。

4、使用 ChangeType()，然后访问 VARIANT 的 bstrVal 成员。在MFC中，如果数据转换不成功将会抛出异常。

5、这里没有转换 BSTR 函数，然而 AllocSysString() 返回一个新的BSTR。

6、使用 GetBuffer() 方法，你可以暂时地得到一个非常量的TCHAR指针。