**CString 操作指南**  
原著：Joseph M. Newcomer  
  
翻译：[littleloach](mailto:littleloach@sina.com)

原文出处：[codeproject：CString Management](http://www.codeproject.com/string/cstringmgmt.asp)  
  
  
通过阅读本文你可以学习如何有效地使用 CString。  
  
　　CString 是一种很有用的数据类型。它们很大程度上简化了MFC中的许多操作，使得MFC在做字符串操作的时候方便了很多。不管怎样，使用CString有很多特殊的技巧，特别是对于纯C背景下走出来的程序员来说有点难以学习。这篇文章就来讨论这些技巧。  
　　使用CString可以让你对字符串的操作更加直截了当。这篇文章不是CString的完全手册，但囊括了大部分常见基本问题。  
  
这篇文章包括以下内容：

1. [CString 对象的连接](http://www.vckbase.com/document/viewdoc/?id=1094#CString 对象的连接)
2. [格式化字符串（包括 int 型转化为 CString ）](http://www.vckbase.com/document/viewdoc/?id=1094#格式化字符串)
3. [CString 型转化成 int 型](http://www.vckbase.com/document/viewdoc/?id=1094#CString 型转化成 int 型)
4. [CString 型和 char\* 类型的相互转化](http://www.vckbase.com/document/viewdoc/?id=1094#CString 型和 char* 类型的相互转化)
5. [char\* 转化成 CString](http://www.vckbase.com/document/viewdoc/?id=1094#char* 转化为 CString)
6. [CString 转化成 char\* 之一：使用LPCTSTR强制转化](http://www.vckbase.com/document/viewdoc/?id=1094#CString转化成char* 之一)
7. [CString 转化成 char\* 之二：使用CString对象的GetBuffer方法](http://www.vckbase.com/document/viewdoc/?id=1094#CString转化成char* 之二)
8. [CString 转化成 char\* 之三: 和控件的接口](http://www.vckbase.com/document/viewdoc/?id=1094#CString to char * 之三)
9. [CString 型转化成 BSTR 型](http://www.vckbase.com/document/viewdoc/?id=1094#CString 型转化成 BSTR 型)；
10. [BSTR 型转化成 CString 型](http://www.vckbase.com/document/viewdoc/?id=1094#BSTR 型转化成 CString 型)；
11. [VARIANT 型转化成 CString 型](http://www.vckbase.com/document/viewdoc/?id=1094#VARIANT 型转化成 CString 型)；
12. [载入字符串表资源](http://www.vckbase.com/document/viewdoc/?id=1094#载入字符串表资源)；
13. [CString 和临时对象](http://www.vckbase.com/document/viewdoc/?id=1094#CString 和临时对象)；
14. [CString 的效率](http://www.vckbase.com/document/viewdoc/?id=1094#CString的效率)；
15. [总结](http://www.vckbase.com/document/viewdoc/?id=1094#总结)

下面我分别讨论。  
  
http://www.vckbase.com/document/image/paragraph.gif 1、**CString 对象的连接**  
  
　　能体现出 CString 类型方便性特点的一个方面就字符串的连接，使用 CString 类型，你能很方便地连接两个字符串，正如下面的例子：

CString gray("Gray");

CString cat("Cat");

CString graycat = gray + cat;

要比用下面的方法好得多：

char gray[] = "Gray";

char cat[] = "Cat";

char \* graycat = malloc(strlen(gray) + strlen(cat) + 1);

strcpy(graycat, gray);

strcat(graycat, cat);

http://www.vckbase.com/document/image/paragraph.gif 2、**格式化字符串**  
  
　　与其用 sprintf() 函数或 wsprintf() 函数来格式化一个字符串，还不如用 CString 对象的Format()方法：

CString s;

s.Format(\_T("The total is %d"), total);

　　用这种方法的好处是你不用担心用来存放格式化后数据的缓冲区是否足够大，这些工作由CString类替你完成。  
　　格式化是一种把其它不是字符串类型的数据转化为CString类型的最常用技巧，比如，把一个整数转化成CString类型，可用如下方法：

CString s;

s.Format(\_T("%d"), total);

　　我总是对我的字符串使用\_T()宏，这是为了让我的代码至少有Unicode的意识，当然，关于Unicode的话题不在这篇文章的讨论范围。\_T()宏在8位字符环境下是如下定义的：

#define \_T(x) x // 非Unicode版本（non-Unicode version）

而在Unicode环境下是如下定义的：

#define \_T(x) L##x // Unicode版本（Unicode version）

所以在Unicode环境下，它的效果就相当于：

s.Format(L"%d", total);

　　如果你认为你的程序可能在Unicode的环境下运行，那么开始在意用 Unicode 编码。比如说，不要用 sizeof() 操作符来获得字符串的长度，因为在Unicode环境下就会有2倍的误差。我们可以用一些方法来隐藏Unicode的一些细节，比如在我需要获得字符长度的时候，我会用一个叫做DIM的宏，这个宏是在我的dim.h文件中定义的，我会在我写的所有程序中都包含这个文件：

#define DIM(x) ( sizeof((x)) / sizeof((x)[0]) )

　　这个宏不仅可以用来解决Unicode的字符串长度的问题，也可以用在编译时定义的表格上，它可以获得表格的项数，如下：

class Whatever { ... };

Whatever data[] = {

{ ... },

...

{ ... },

};

for(int i = 0; i < DIM(data); i++) // 扫描表格寻找匹配项。

　　这里要提醒你的就是一定要注意那些在参数中需要真实字节数的API函数调用，如果你传递字符个数给它，它将不能正常工作。如下：

TCHAR data[20];

lstrcpyn(data, longstring, sizeof(data) - 1); // WRONG!

lstrcpyn(data, longstring, DIM(data) - 1); // RIGHT

WriteFile(f, data, DIM(data), &bytesWritten, NULL); // WRONG!

WriteFile(f, data, sizeof(data), &bytesWritten, NULL); // RIGHT

造成以上原因是因为lstrcpyn需要一个字符个数作为参数，但是WriteFile却需要字节数作为参数。  
同样需要注意的是有时候需要写出数据的所有内容。如果你仅仅只想写出数据的真实长度，你可能会认为你应该这样做：

WriteFile(f, data, lstrlen(data), &bytesWritten, NULL); // WRONG

但是在Unicode环境下，它不会正常工作。正确的做法应该是这样：

WriteFile(f, data, lstrlen(data) \* sizeof(TCHAR), &bytesWritten, NULL); // RIGHT

　　因为WriteFile需要的是一个以字节为单位的长度。（可能有些人会想“在非Unicode的环境下运行这行代码，就意味着总是在做一个多余的乘1操作，这样不会降低程序的效率吗？”这种想法是多余的，你必须要了解编译器实际上做了什么，没有哪一个C或C++编译器会把这种无聊的乘1操作留在代码中。在Unicode环境下运行的时候，你也不必担心那个乘2操作会降低程序的效率，记住，这只是一个左移一位的操作而已，编译器也很乐意为你做这种替换。）  
　　使用\_T宏并不是意味着你已经创建了一个Unicode的程序，你只是创建了一个有Unicode意识的程序而已。如果你在默认的8-bit模式下编译你的程序的话，得到的将是一个普通的8-bit的应用程序（这里的8-bit指的只是8位的字符编码，并不是指8位的计算机系统）；当你在Unicode环境下编译你的程序时，你才会得到一个Unicode的程序。记住，CString 在 Unicode 环境下，里面包含的可都是16位的字符哦。  
  
http://www.vckbase.com/document/image/paragraph.gif 3、**CString 型转化成 int 型**  
  
　　把 CString 类型的数据转化成整数类型最简单的方法就是使用标准的字符串到整数转换例程。  
　　虽然通常你怀疑使用\_atoi()函数是一个好的选择，它也很少会是一个正确的选择。如果你准备使用 Unicode 字符，你应该用\_ttoi()，它在 ANSI 编码系统中被编译成\_atoi()，而在 Unicode 编码系统中编译成\_wtoi()。你也可以考虑使用\_tcstoul()或者\_tcstol()，它们都能把字符串转化成任意进制的长整数（如二进制、八进制、十进制或十六进制），不同点在于前者转化后的数据是无符号的（unsigned），而后者相反。看下面的例子：

CString hex = \_T("FAB");

CString decimal = \_T("4011");

ASSERT(\_tcstoul(hex, 0, 16) == \_ttoi(decimal));

http://www.vckbase.com/document/image/paragraph.gif 4、**CString 型和 char\* 类型的相互转化**  
  
　　这是初学者使用 CString 时最常见的问题。有了 C++ 的帮助，很多问题你不需要深入的去考虑它，直接拿来用就行了，但是如果你不能深入了解它的运行机制，又会有很多问题让你迷惑，特别是有些看起来没有问题的代码，却偏偏不能正常工作。  
比如，你会奇怪为什么不能写向下面这样的代码呢：

CString graycat = "Gray" + "Cat";

或者这样：

CString graycat("Gray" + "Cat");

　　事实上，编译器将抱怨上面的这些尝试。为什么呢？因为针对CString 和 LPCTSTR数据类型的各种各样的组合，“ +” 运算符 被定义成一个重载操作符。而不是两个 LPCTSTR 数据类型，它是底层数据类型。你不能对基本数据（如 int、char 或者 char\*）类型重载 C++ 的运算符。你可以象下面这样做：

CString graycat = CString("Gray") + CString("Cat");

或者这样：

CString graycat = CString("Gray") + "Cat";

研究一番就会发现：“ +”总是使用在至少有一个 CString 对象和一个 LPCSTR 的场合。  
  
注意，编写有 Unicode 意识的代码总是一件好事，比如：

CString graycat = CString(\_T("Gray")) + \_T("Cat");

这将使得你的代码可以直接移植。  
  
***char\* 转化为 CString***  
  
　　现在你有一个 char\* 类型的数据，或者说一个字符串。怎么样创建 CString 对象呢？这里有一些例子：

char \* p = "This is a test";

或者象下面这样更具有 Unicode 意识：

TCHAR \* p = \_T("This is a test")

或

LPTSTR p = \_T("This is a test");

你可以使用下面任意一种写法：

CString s = "This is a test"; // 8-bit only

CString s = \_T("This is a test"); // Unicode-aware

CString s("This is a test"); // 8-bit only

CString s(\_T("This is a test")); // Unicode-aware

CString s = p;

CString s(p);

　　用这些方法可以轻松将常量字符串或指针转换成 CString。需要注意的是，字符的赋值总是被拷贝到 CString 对象中去的，所以你可以象下面这样操作：

TCHAR \* p = \_T("Gray");

CString s(p);

p = \_T("Cat");

s += p;

结果字符串肯定是“GrayCat”。  
  
CString 类还有几个其它的构造函数，但是这里我们不考虑它，如果你有兴趣可以自己查看相关文档。  
  
事实上，CString 类的构造函数比我展示的要复杂，比如：

CString s = "This is a test";

　　这是很草率的编码，但是实际上它在 Unicode 环境下能编译通过。它在运行时调用构造函数的 MultiByteToWideChar 操作将 8 位字符串转换成 16 位字符串。不管怎样，如果 char \* 指针是网络上传输的 8 位数据，这种转换是很有用的。  
  
***CString 转化成 char\* 之一：***强制类型转换为 LPCTSTR；  
  
　　这是一种略微硬性的转换，有关“正确”的做法，人们在认识上还存在许多混乱，正确的使用方法有很多，但错误的使用方法可能与正确的使用方法一样多。  
　　我们首先要了解 CString 是一种很特殊的 C++ 对象，它里面包含了三个值：一个指向某个数据缓冲区的指针、一个是该缓冲中有效的字符记数以及一个缓冲区长度。 有效字符数的大小可以是从0到该缓冲最大长度值减1之间的任何数（因为字符串结尾有一个NULL字符）。字符记数和缓冲区长度被巧妙隐藏。  
　　除非你做一些特殊的操作，否则你不可能知道给CString对象分配的缓冲区的长度。这样，即使你获得了该0缓冲的地址，你也无法更改其中的内容，不能截短字符串，也 绝对没有办法加长它的内容，否则第一时间就会看到溢出。  
　　LPCTSTR 操作符（或者更明确地说就是 TCHAR \* 操作符）在 CString 类中被重载了，该操作符的定义是返回缓冲区的地址，因此，如果你需要一个指向 CString 的 字符串指针的话，可以这样做：

CString s("GrayCat");

LPCTSTR p = s;

　　它可以正确地运行。这是由C语言的强制类型转化规则实现的。当需要强制类型转化时，C++规测容许这种选择。比如，你可以将（浮点数）定义为将某个复数 （有一对浮点数）进行强制类型转换后只返回该复数的第一个浮点数（也就是其实部）。可以象下面这样：

Complex c(1.2f, 4.8f);

float realpart = c;

如果(float)操作符定义正确的话，那么实部的的值应该是1.2。  
　　这种强制转化适合所有这种情况，例如，任何带有 LPCTSTR 类型参数的函数都会强制执行这种转换。 于是，你可能有这样一个函数（也许在某个你买来的DLL中）：

BOOL DoSomethingCool(LPCTSTR s);

你象下面这样调用它：

CString file("c:\\myfiles\\coolstuff")

BOOL result = DoSomethingCool(file);

　　它能正确运行。因为 DoSomethingCool 函数已经说明了需要一个 LPCTSTR 类型的参数，因此 LPCTSTR 被应用于该参数，在 MFC 中就是返回的串地址。  
  
如果你要格式化字符串怎么办呢？

CString graycat("GrayCat");

CString s;

s.Format("Mew! I love %s", graycat);

　　注意由于在可变参数列表中的值（在函数说明中是以“...”表示的）并没有隐含一个强制类型转换操作符。你会得到什么结果呢？  
　　一个令人惊讶的结果，我们得到的实际结果串是：

"Mew! I love GrayCat"。

　　因为 MFC 的设计者们在设计 CString 数据类型时非常小心， CString 类型表达式求值后指向了字符串，所以这里看不到任何象 Format 或 sprintf 中的强制类型转换，你仍然可以得到正确的行为。描述 CString 的附加数据实际上在 CString 名义地址之后。  
　　有一件事情你是不能做的，那就是修改字符串。比如，你可能会尝试用“,”代替“.”（不要做这样的，如果你在乎国际化问题，你应该使用十进制转换的 National Language Support 特性，），下面是个简单的例子：

CString v("1.00"); // 货币金额，两位小数

LPCTSTR p = v;

p[lstrlen(p) - 3] = '','';

　　这时编译器会报错，因为你赋值了一个常量串。如果你做如下尝试，编译器也会错：

strcat(p, "each");

　　因为 strcat 的第一个参数应该是 LPTSTR 类型的数据，而你却给了一个 LPCTSTR。  
  
　　不要试图钻这个错误消息的牛角尖，这只会使你自己陷入麻烦！  
  
　　原因是缓冲有一个计数，它是不可存取的（它位于 CString 地址之下的一个隐藏区域），如果你改变这个串，缓冲中的字符计数不会反映所做的修改。此外，如果字符串长度恰好是该字符串物理限制的长度（梢后还会讲到这个问题），那么扩展该字符串将改写缓冲以外的任何数据，那是你无权进行写操作的内存（不对吗？），你会毁换坏不属于你的内存。这是应用程序真正的死亡处方。

***CString转化成char\* 之二：***使用 CString 对象的 GetBuffer 方法；  
  
　　如果你需要修改 CString 中的内容，它有一个特殊的方法可以使用，那就是 GetBuffer，它的作用是返回一个可写的缓冲指针。 如果你只是打算修改字符或者截短字符串，你完全可以这样做：

CString s(\_T("File.ext"));

LPTSTR p = s.GetBuffer();

LPTSTR dot = strchr(p, ''.''); // OK, should have used s.Find...

if(p != NULL)

\*p = \_T(''\0'');

s.ReleaseBuffer();

　　这是 GetBuffer 的第一种用法，也是最简单的一种，不用给它传递参数，它使用默认值 0，意思是：“给我这个字符串的指针，我保证不加长它”。当你调用 ReleaseBuffer 时，字符串的实际长度会被重新计算，然后存入 CString 对象中。  
　　必须强调一点，在 GetBuffer 和 ReleaseBuffer 之间这个范围，一定不能使用你要操作的这个缓冲的 CString 对象的任何方法。因为 ReleaseBuffer 被调用之前，该 CString 对象的完整性得不到保障。研究以下代码：

CString s(...);

LPTSTR p = s.GetBuffer();

//... 这个指针 p 发生了很多事情

int n = s.GetLength(); // 很糟D!!!!! 有可能给出错误的答案!!!

s.TrimRight(); // 很糟!!!!! 不能保证能正常工作!!!!

s.ReleaseBuffer(); // 现在应该 OK

int m = s.GetLength(); // 这个结果可以保证是正确的。

s.TrimRight(); // 将正常工作。

　　假设你想增加字符串的长度，你首先要知道这个字符串可能会有多长，好比是声明字符串数组的时候用：

char buffer[1024];

表示 1024 个字符空间足以让你做任何想做得事情。在 CString 中与之意义相等的表示法：

LPTSTR p = s.GetBuffer(1024);

　　调用这个函数后，你不仅获得了字符串缓冲区的指针，而且同时还获得了长度至少为 1024 个字符的空间（注意，我说的是“字符”，而不是“字节”，因为 CString 是以隐含方式感知 Unicode 的）。  
　　同时，还应该注意的是，如果你有一个常量串指针，这个串本身的值被存储在只读内存中，如果试图存储它，即使你已经调用了 GetBuffer ，并获得一个只读内存的指针，存入操作会失败，并报告存取错误。我没有在 CString 上证明这一点，但我看到过大把的 C 程序员经常犯这个错误。  
　　C 程序员有一个通病是分配一个固定长度的缓冲，对它进行 sprintf 操作，然后将它赋值给一个 CString：

char buffer[256];

sprintf(buffer, "%......", args, ...); // ... 部分省略许多细节

CString s = buffer;

虽然更好的形式可以这么做：

CString s;

s.Format(\_T("%...."), args, ...);

如果你的字符串长度万一超过 256 个字符的时候，不会破坏堆栈。  
  
　　另外一个常见的错误是：既然固定大小的内存不工作，那么就采用动态分配字节，这种做法弊端更大：

int len = lstrlen(parm1) + 13 lstrlen(parm2) + 10 + 100;

char \* buffer = new char[len];

sprintf(buffer, "%s is equal to %s, valid data", parm1, parm2);

CString s = buffer;

......

delete [] buffer;

它可以能被简单地写成：

CString s;

s.Format(\_T("%s is equal to %s, valid data"), parm1, parm2);

　　需要注意 sprintf 例子都不是 Unicode 就绪的，尽管你可以使用 tsprintf 以及用 \_T() 来包围格式化字符串，但是基本 思路仍然是在走弯路，这这样很容易出错。  
  
***CString to char \* 之三：***和控件的接口；  
  
　　我们经常需要把一个 CString 的值传递给一个控件，比如，CTreeCtrl。MFC为我们提供了很多便利来重载这个操作，但是 在大多数情况下，你使用“原始”形式的更新，因此需要将墨某个串指针存储到 TVINSERTITEMSTRUCT 结构的 TVITEM 成员中。如下：

TVINSERTITEMSTRUCT tvi;

CString s;

// ... 为s赋一些值。

tvi.item.pszText = s; // Compiler yells at you here

// ... 填写tvi的其他域

HTREEITEM ti = c\_MyTree.InsertItem(&tvi);

　　为什么编译器会报错呢？明明看起来很完美的用法啊！但是事实上如果你看看 TVITEM 结构的定义你就会明白，在 TVITEM 结构中 pszText 成员的声明如下：

LPTSTR pszText;

int cchTextMax;

　　因此，赋值不是赋给一个 LPCTSTR 类型的变量，而且编译器无法知道如何将赋值语句右边强制转换成 LPCTSTR。好吧，你说，那我就改成这样：

tvi.item.pszText = (LPCTSTR)s; //编译器依然会报错。

　　编译器之所以依然报错是因为你试图把一个 LPCTSTR 类型的变量赋值给一个 LPTSTR 类型的变量，这种操作在C或C++中是被禁止的。你不能用这种方法 来滥用常量指针与非常量指针概念，否则，会扰乱编译器的优化机制，使之不知如何优化你的程序。比如，如果你这么做：

const int i = ...;

//... do lots of stuff

... = a[i]; // usage 1

// ... lots more stuff

... = a[i]; // usage 2

　　那么，编译器会以为既然 i 是 const ，所以 usage1和usage2的值是相同的，并且它甚至能事先计算好 usage1 处的 a[i] 的地址，然后保留着在后面的 usage2 处使用，而不是重新计算。如果你按如下方式写的话：

const int i = ...;

int \* p = &i;

//... do lots of stuff

... = a[i]; // usage 1

// ... lots more stuff

(\*p)++; // mess over compiler''s assumption

// ... and other stuff

... = a[i]; // usage 2

　　编译器将认为 i 是常量，从而 a[i] 的位置也是常量，这样间接地破坏了先前的假设。因此，你的程序将会在 debug 编译模式（没有优化）和 release 编译模式（完全优化）中反映出不同的行为，这种情况可不好，所以当你试图把指向 i 的指针赋值给一个 可修改的引用时，会被编译器诊断为这是一种伪造。这就是为什么（LPCTSTR）强制类型转化不起作用的原因。  
　　为什么不把该成员声明成 LPCTSTR 类型呢？因为这个结构被用于读写控件。当你向控件写数据时，文本指针实际上被当成 LPCTSTR，而当你从控件读数据 时，你必须有一个可写的字符串。这个结构无法区分它是用来读还是用来写。  
  
因此，你会常常在我的代码中看到如下的用法：

tvi.item.pszText = (LPTSTR)(LPCTSTR)s;

　　它把 CString 强制类型转化成 LPCTSTR，也就是说先获得改字符串的地址，然后再强制类型转化成 LPTSTR，以便可以对之进行赋值操作。 注意这只有在使用 Set 或 Insert 之类的方法才有效！如果你试图获取数据，则不能这么做。  
　　如果你打算获取存储在控件中的数据，则方法稍有不同，例如，对某个 CTreeCtrl 使用 GetItem 方法，我想获取项目的文本。我知道这些 文本的长度不会超过 MY\_LIMIT，因此我可以这样写：

TVITEM tvi;

// ... assorted initialization of other fields of tvi

tvi.pszText = s.GetBuffer(MY\_LIMIT);

tvi.cchTextMax = MY\_LIMIT;

c\_MyTree.GetItem(&tvi);

s.ReleaseBuffer();

　　可以看出来，其实上面的代码对所有类型的 Set 方法都适用，但是并不需要这么做，因为所有的类 Set 方法（包括 Insert方法）不会改变字符串的内容。但是当你需要写 CString 对象时，必须保证缓冲是可写的，这正是 GetBuffer 所做的事情。再次强调： 一旦做了一次 GetBuffer 调用，那么在调用 ReleaseBuffer 之前不要对这个 CString 对象做任何操作。  
  
http://www.vckbase.com/document/image/paragraph.gif 5、**CString 型转化成 BSTR 型**  
  
　　当我们使用 ActiveX 控件编程时，经常需要用到将某个值表示成 BSTR 类型。BSTR 是一种记数字符串，Intel平台上的宽字符串（Unicode），并且 可以包含嵌入的 NULL 字符。  
  
你可以调用 CString 对象的 AllocSysString 方法将 CString 转化成 BSTR：

CString s;

s = ... ; // whatever

BSTR b = s.AllocSysString();

　　现在指针 b 指向的就是一个新分配的 BSTR 对象，该对象是 CString 的一个拷贝，包含终结 NULL字符。现在你可以将它传递给任何需要 BSTR 的接口。通常，BSTR 由接收它的组件来释放，如果你需要自己释放 BSTR 的话，可以这么做：

::SysFreeString(b);

　　对于如何表示传递给 ActiveX 控件的字符串，在微软内部曾一度争论不休，最后 Visual Basic 的人占了上风，BSTR（“Basic String”的首字母缩写）就是这场争论的结果。  
  
http://www.vckbase.com/document/image/paragraph.gif 6、**BSTR 型转化成 CString 型**  
  
　　由于 BSTR 是记数 Unicode 字符串，你可以用标准转换方法来创建 8 位的 CString。实际上，这是 CString 内建的功能。在 CString 中 有特殊的构造函数可以把 ANSI 转化成 Unicode，也可以把Unicode 转化成 ANSI。你同样可以从 VARIANT 类型的变量中获得 BSTR 类型的字符串，VARIANT 类型是 由各种 COM 和 Automation (自动化)调用返回的类型。  
  
例如，在一个ANSI程序中：

BSTR b;

b = ...; // whatever

CString s(b == NULL ? L"" : b)

　　对于单个的 BSTR 串来说，这种用法可以工作得很好，这是因为 CString 有一个特殊的构造函数以LPCWSTR（BSTR正是这种类型） 为参数，并将它转化成 ANSI 类型。专门检查是必须的，因为 BSTR 可能为空值，而 CString 的构造函数对于 NULL 值情况考虑的不是很周到，（感谢 Brian Ross 指出这一点!）。这种用法也只能处理包含 NUL 终结字符的单字符串；如果要转化含有多个 NULL 字符 串，你得额外做一些工作才行。在 CString 中内嵌的 NULL 字符通常表现不尽如人意，应该尽量避免。  
　　根据 C/C++ 规则，如果你有一个 LPWSTR，那么它别无选择，只能和 LPCWSTR 参数匹配。  
  
在 Unicode 模式下，它的构造函数是：

CString::CString(LPCTSTR);

正如上面所表示的，在 ANSI 模式下，它有一个特殊的构造函数：

CString::CString(LPCWSTR);

　　它会调用一个内部的函数将 Unicode 字符串转换成 ANSI 字符串。（在Unicode模式下，有一个专门的构造函数，该函数有一个参数是LPCSTR类型——一个8位 ANSI 字符串 指针，该函数将它加宽为 Unicode 的字符串！）再次强调：一定要检查 BSTR 的值是否为 NULL。  
　　另外还有一个问题，正如上文提到的：BSTRs可以含有多个内嵌的NULL字符，但是 CString 的构造函数只能处理某个串中单个 NULL 字符。 也就是说，如果串中含有嵌入的 NUL字节，CString 将会计算出错误的串长度。你必须自己处理它。如果你看看 strcore.cpp 中的构造函数，你会发现 它们都调用了lstrlen，也就是计算字符串的长度。  
　　注意从 Unicode 到 ANSI 的转换使用带专门参数的 ::WideCharToMultiByte，如果你不想使用这种默认的转换方式，则必须编写自己的转化代码。  
　　如果你在 UNICODE 模式下编译代码，你可以简单地写成：

CString convert(BSTR b)

{

if(b == NULL)

return CString(\_T(""));

CString s(b); // in UNICODE mode

return s;

}

　　如果是 ANSI 模式，则需要更复杂的过程来转换。注意这个代码使用与 ::WideCharToMultiByte 相同的参数值。所以你 只能在想要改变这些参数进行转换时使用该技术。例如，指定不同的默认字符，不同的标志集等。

CString convert(BSTR b)

{

CString s;

if(b == NULL)

return s; // empty for NULL BSTR

#ifdef UNICODE

s = b;

#else

LPSTR p = s.GetBuffer(SysStringLen(b) + 1);

::WideCharToMultiByte(CP\_ACP, // ANSI Code Page

0, // no flags

b, // source widechar string

-1, // assume NUL-terminated

p, // target buffer

SysStringLen(b)+1, // target buffer length

NULL, // use system default char

NULL); // don''t care if default used

s.ReleaseBuffer();

#endif

return s;

}

　　我并不担心如果 BSTR 包含没有映射到 8 位字符集的 Unicode 字符时会发生什么，因为我指定了::WideCharToMultiByte 的最后两个参数为 NULL。这就是你可能需要改变的地方。   
  
http://www.vckbase.com/document/image/paragraph.gif 7、**VARIANT 型转化成 CString 型**  
  
　　事实上，我从来没有这么做过，因为我没有用 COM/OLE/ActiveX 编写过程序。但是我在microsoft.public.vc.mfc 新闻组上看到了 Robert Quirk 的一篇帖子谈到了这种转化，我觉得把他的文章包含在我的文章里是不太好的做法，所以在这里多做一些解释和演示。如果和他的文章有相孛的地方可能是我的疏忽。  
　　VARIANT 类型经常用来给 COM 对象传递参数，或者接收从 COM 对象返回的值。你也能自己编写返回 VARIANT 类型的方法，函数返回什么类型 依赖可能（并且常常）方法的输入参数（比如，在自动化操作中，依赖与你调用哪个方法。IDispatch::Invoke 可能返回（通过其一个参数）一个 包含有BYTE、WORD、float、double、date、BSTR 等等 VARIANT 类型的结果，（详见 MSDN 上的 VARIANT 结构的定义）。在下面的例子中，假设 类型是一个BSTR的变体，也就是说在串中的值是通过 bsrtVal 来引用，其优点是在 ANSI 应用中，有一个构造函数会把 LPCWCHAR 引用的值转换为一个 CString（见 BSTR-to-CString 部分）。在 Unicode 模式中，将成为标准的 CString 构造函数，参见对缺省::WideCharToMultiByte 转换的告诫，以及你觉得是否可以接受（大多数情况下，你会满意的）。

VARIANT vaData;

vaData = m\_com.YourMethodHere();

ASSERT(vaData.vt == VT\_BSTR);

CString strData(vaData.bstrVal);

你还可以根据 vt 域的不同来建立更通用的转换例程。为此你可能会考虑：

CString VariantToString(VARIANT \* va)

{

CString s;

switch(va->vt)

{ /\* vt \*/

case VT\_BSTR:

return CString(vaData->bstrVal);

case VT\_BSTR | VT\_BYREF:

return CString(\*vaData->pbstrVal);

case VT\_I4:

s.Format(\_T("%d"), va->lVal);

return s;

case VT\_I4 | VT\_BYREF:

s.Format(\_T("%d"), \*va->plVal);

case VT\_R8:

s.Format(\_T("%f"), va->dblVal);

return s;

... 剩下的类型转换由读者自己完成

default:

ASSERT(FALSE); // unknown VARIANT type (this ASSERT is optional)

return CString("");

} /\* vt \*/

}

http://www.vckbase.com/document/image/paragraph.gif 8、**载入字符串表资源**  
  
　　如果你想创建一个容易进行语言版本移植的应用程序，你就不能在你的源代码中直接包含本土语言字符串 （下面这些例子我用的语言都是英语，因为我的本土语是英语），比如下面这种写法就很糟：

CString s = "There is an error";

　　你应该把你所有特定语言的字符串单独摆放（调试信息、在发布版本中不出现的信息除外）。这意味着向下面这样写比较好：

s.Format(\_T("%d - %s"), code, text);

　　在你的程序中，文字字符串不是语言敏感的。不管怎样，你必须很小心，不要使用下面这样的串：

// fmt is "Error in %s file %s"

// readorwrite is "reading" or "writing"

s.Format(fmt, readorwrite, filename);

　　这是我的切身体会。在我的第一个国际化的应用程序中我犯了这个错误，尽管我懂德语，知道在德语的语法中动词放在句子的最后面，我们的德国方面的发行人还是苦苦的抱怨他们不得不提取那些不可思议的德语错误提示信息然后重新格式化以让它们能正常工作。比较好的办法（也是我现在使用的办法）是使用两个字符串，一个用 于读，一个用于写，在使用时加载合适的版本，使得它们对字符串参数是非敏感的。也就是说加载整个格式，而不是加载串“reading”，“writing”：

// fmt is "Error in reading file %s"

// "Error in writing file %s"

s.Format(fmt, filename);

　　一定要注意，如果你有好几个地方需要替换，你一定要保证替换后句子的结构不会出现问题，比如在英语中，可以是主语-宾语，主语-谓语，动词-宾语的结构等等。  
　　在这里，我们并不讨论 FormatMessage，其实它比 sprintf/Format 还要有优势，但是不太容易和CString 结合使用。解决这种问题的办法就是我们按照参数出现在参数表中的位置给参数取名字，这样在你输出的时候就不会把他们的位置排错了。  
　　接下来我们讨论我们这些独立的字符串放在什么地方。我们可以把字符串的值放入资源文件中的一个称为 STRINGTABLE 的段中。过程如下：首先使用 Visual Studio 的资源编辑器创建一个字符串，然后给每一个字符串取一个ID，一般我们给它取名字都以 IDS\_开头。所以如果你有一个信息，你可以创建一个字符串资源然后取名为 IDS\_READING\_FILE，另外一个就取名为 IDS\_WRITING\_FILE。它们以下面的形式出现在你的 .rc 文件中：

STRINGTABLE

IDS\_READING\_FILE "Reading file %s"

IDS\_WRITING\_FILE "Writing file %s"

END

**注意：**这些资源都以 Unicode 的格式保存，不管你是在什么环境下编译。他们在Win9x系统上也是以Unicode 的形式存在，虽然 Win9x 不能真正处理 Unicode。  
然后你可以这样使用这些资源：  
// 在使用资源串表之前，程序是这样写的：

CString fmt;

if(...)

fmt = "Reading file %s";

else

fmt = "Writing file %s";

...

// much later

CString s;

s.Format(fmt, filename);

// 使用资源串表之后，程序这样写：

CString fmt;

if(...)

fmt.LoadString(IDS\_READING\_FILE);

else

fmt.LoadString(DS\_WRITING\_FILE);

...

// much later

CString s;

s.Format(fmt, filename);

　　现在，你的代码可以移植到任何语言中去。LoadString 方法需要一个字符串资源的 ID 作为参数，然后它从 STRINGTABLE 中取出它对应的字符串，赋值给 CString 对象。 CString 对象的构造函数还有一个更加聪明的特征可以简化 STRINGTABLE 的使用。这个用法在 CString::CString 的文档中没有指出，但是在 构造函数的示例程序中使用了。（为什么这个特性没有成为正式文档的一部分，而是放在了一个例子中，我记不得了！）——【**译者注**：从这句话看，作者可能是CString的设计者。其实前面还有一句类似的话。说他没有对使用GetBuffer(0)获得的指针指向的地址是否可读做有效性检查 】。这个特征就是：如果你将一个字符串资源的ID强制类型转换为 LPCTSTR，将会隐含调用 LoadString。因此，下面两个构造字符串的例子具有相同的效果，而且其 ASSERT 在debug模式下不会被触发：

CString s;

s.LoadString(IDS\_WHATEVER);

CString t( (LPCTSTR)IDS\_WHATEVER );

ASSERT(s == t);//不会被触发，说明s和t是相同的。

　　现在，你可能会想：这怎么可能工作呢？我们怎么能把 STRINGTABLE ID 转化成一个指针呢？很简单：所有的字符串 ID 都在1~65535这个范围内，也就是说，它所有的高位都是0，而我们在程序中所使用的指针是不可能小于65535的，因为程序的低 64K 内存永远也不可能存在的，如果你试图访问0x00000000到0x0000FFFF之间的内存，将会引发一个内存越界错误。所以说1~65535的值不可能是一个内存地址，所以我们可以用这些值来作为字符串资源的ID。  
　　我倾向于使用 MAKEINTRESOURCE 宏显式地做这种转换。我认为这样可以让代码更加易于阅读。这是个只适合在 MFC 中使用的标准宏。你要记住，大多数的方法即可以接受一个 UINT 型的参数，也可以接受一个 LPCTSTR 型的参数，这是依赖 C++ 的重载功能做到的。C++重载函数带来的 弊端就是造成所有的强制类型转化都需要显示声明。同样，你也可以给很多种结构只传递一个资源名。

CString s;

s.LoadString(IDS\_WHATEVER);

CString t( MAKEINTRESOURCE(IDS\_WHATEVER));

ASSERT(s == t);

　　告诉你吧：我不仅只是在这里鼓吹，事实上我也是这么做的。在我的代码中，你几乎不可能找到一个字符串，当然，那些只是偶然在调试中出现的或者和语言无关的字符串除外。  
  
http://www.vckbase.com/document/image/paragraph.gif 9、**CString 和临时对象**  
  
　　这是出现在 microsoft.public.vc.mfc 新闻组中的一个小问题，我简单的提一下，这个问题是有个程序员需要往注册表中写入一个字符串，他写道：  
　　我试着用 RegSetValueEx() 设置一个注册表键的值，但是它的结果总是令我困惑。当我用char[]声明一个变量时它能正常工作，但是当我用 CString 的时候，总是得到一些垃圾："&Yacute;&Yacute;&Yacute;&Yacute;...&Yacute;&Yacute;&Yacute;&Yacute;&Yacute;&Yacute;"为了确认是不是我的 CString 数据出了问题，我试着用 GetBuffer，然后强制转化成 char\*，LPCSTR。GetBuffer 返回的值是正确的，但是当我把它赋值给 char\* 时，它就变成垃圾了。以下是我的程序段：

char\* szName = GetName().GetBuffer(20);

RegSetValueEx(hKey, "Name", 0, REG\_SZ,

(CONST BYTE \*) szName,

strlen (szName + 1));

这个 Name 字符串的长度小于 20，所以我不认为是 GetBuffer 的参数的问题。  
  
真让人困惑，请帮帮我。  
  
亲爱的 Frustrated，  
  
你犯了一个相当微妙的错误，聪明反被聪明误，正确的代码应该象下面这样：

CString Name = GetName();

RegSetValueEx(hKey, \_T("Name"), 0, REG\_SZ,

(CONST BYTE \*) (LPCTSTR)Name,

(Name.GetLength() + 1) \* sizeof(TCHAR));

　　为什么我写的代码能行而你写的就有问题呢？主要是因为当你调用 GetName 时返回的 CString 对象是一个临时对象。参见：《C++ Reference manual》§12.2  
　　在一些环境中，编译器有必要创建一个临时对象，这样引入临时对象是依赖于实现的。如果编译器引入的这个临时对象所属的类有构造函数的话，编译器要确保这个类的构造函数被调用。同样的，如果这个类声明有析构函数的话，也要保证这个临时对象的析构函数被调用。  
　　编译器必须保证这个临时对象被销毁了。被销毁的确切地点依赖于实现.....这个析构函数必须在退出创建该临时对象的范围之前被调用。  
　　大部分的编译器是这样设计的：在临时对象被创建的代码的下一个执行步骤处隐含调用这个临时对象的析构函数，实现起来，一般都是在下一个分号处。因此，这个 CString 对象在 GetBuffer 调用之后就被析构了（顺便提一句，你没有理由给 GetBuffer 函数传递一个参数，而且没有使用ReleaseBuffer 也是不对的）。所以 GetBuffer 本来返回的是指向这个临时对象中字符串的地址的指针，但是当这个临时对象被析构后，这块内存就被释放了。然后 MFC 的调试内存分配器会重新为这块内存全部填上 0xDD，显示出来刚好就是“&Yacute;”符号。在这个时候你向注册表中写数据，字符串的内容当然全被破坏了。  
　　我们不应该立即把这个临时对象转化成 char\* 类型，应该先把它保存到一个 CString 对象中，这意味着把临时对象复制了一份，所以当临时的 CString 对象被析构了之后，这个 CString 对象中的值依然保存着。这个时候再向注册表中写数据就没有问题了。  
　　此外，我的代码是具有 Unicode 意识的。那个操作注册表的函数需要一个字节大小，使用lstrlen(Name+1) 得到的实际结果对于 Unicode 字符来说比 ANSI 字符要小一半，而且它也不能从这个字符串的第二个字符起开始计算，也许你的本意是 lstrlen(Name) + 1（OK，我承认，我也犯了同样的错误！）。不论如何，在 Unicode 模式下，所有的字符都是2个字节大小，我们需要处理这个问题。微软的文档令人惊讶地对此保持缄默：REG\_SZ 的值究竟是以字节计算还是以字符计算呢？我们假设它指的是以字节为单位计算，你需要对你的代码做一些修改来计算这个字符串所含有的字节大小。  
  
http://www.vckbase.com/document/image/paragraph.gif 10、**CString 的效率**  
  
　　CString 的一个问题是它确实掩藏了一些低效率的东西。从另外一个方面讲，它也确实可以被实现得更加高效，你可能会说下面的代码：

CString s = SomeCString1;

s += SomeCString2;

s += SomeCString3;

s += ",";

s += SomeCString4;

比起下面的代码来，效率要低多了：

char s[1024];

lstrcpy(s, SomeString1);

lstrcat(s, SomeString2);

lstrcat(s, SomeString 3);

lstrcat(s, ",");

lstrcat(s, SomeString4);

　　总之，你可能会想，首先，它为 SomeCString1 分配一块内存，然后把 SomeCString1 复制到里面，然后发现它要做一个连接，则重新分配一块新的足够大的内存，大到能够放下当前的字符串加上SomeCString2，把内容复制到这块内存 ，然后把 SomeCString2 连接到后面，然后释放第一块内存，并把指针重新指向新内存。然后为每个字符串重复这个过程。把这 4 个字符串连接起来效率多低啊。事实上，在很多情况下根本就不需要复制源字符串（在 += 操作符左边的字符串）。  
　　在 VC++6.0 中，Release 模式下，所有的 CString 中的缓存都是按预定义量子分配的。所谓量子，即确定为 64、128、256 或者 512 字节。这意味着除非字符串非常长，连接字符串的操作实际上就是 strcat 经过优化后的版本（因为它知道本地的字符串应该在什么地方结束，所以不需要寻找字符串的结尾；只需要把内存中的数据拷贝到指定的地方即可）加上重新计算字符串的长度。所以它的执行效率和纯 C 的代码是一样的，但是它更容易写、更容易维护和更容易理解。  
　　如果你还是不能确定究竟发生了怎样的过程，请看看 CString 的源代码，strcore.cpp，在你 vc98的安装目录的 mfc\src 子目录中。看看 ConcatInPlace 方法，它被在所有的 += 操作符中调用。  
  
啊哈！难道 CString 真的这么"高效"吗？比如，如果我创建

CString cat("Mew!");

　　然后我并不是得到了一个高效的、精简的5个字节大小的缓冲区（4个字符加一个结束字符），系统将给我分配64个字节，而其中59个字节都被浪费了。  
　　如果你也是这么想的话，那么就请准备好接受再教育吧。可能在某个地方某个人给你讲过尽量使用少的空间是件好事情。不错，这种说法的确正确，但是他忽略了事实中一个很重要的方面。  
　　如果你编写的是运行在16K EPROMs下的嵌入式程序的话，你有理由尽量少使用空间，在这种环境下，它能使你的程序更健壮。但是在 500MHz, 256MB的机器上写 Windows 程序，如果你还是这么做，它只会比你认为的“低效”的代码运行得更糟。  
　　举例来说。字符串的大小被认为是影响效率的首要因素，使字符串尽可能小可以提高效率，反之则降低效率，这是大家一贯的想法。但是这种想法是不对的，精确的内存分配的后果要在程序运行了好几个小时后才能体现得出来，那时，程序的堆中将充满小片的内存，它们太小以至于不能用来做任何事，但是他们增加了你程序的内存用量，增加了内存页面交换的次数，当页面交换的次数增加到系统能够忍受的上限，系统则会为你的程序分配更多的页面，直到你的程序占用了所有的可用内存。由此可见，虽然内存碎片是决定效率的次要因素，但正是这些因素实际控制了系统的行为，最终，它损害了系统的可靠性，这是令人无法接受的。  
　　记住，在 debug 模式下，内存往往是精确分配的，这是为了更好的排错。  
　　假设你的应用程序通常需要连续工作好几个月。比如，我常打开 VC++，Word，PowerPoint，Frontpage，Outlook Express，Forté Agent，Internet Explorer和其它的一些程序，而且通常不关闭它们。我曾经夜以继日地连续用 PowerPoint 工作了好几天（反之，如果你不幸不得不使用像 Adobe FrameMaker 这样的程序的话，你将会体会到可靠性的重要；这个程序机会每天都要崩溃4~6次，每次都是因为用完了所有的空间并填满我所有的交换页面）。所以精确内存分配是不可取的，它会危及到系统的可靠性，并引起应用程序崩溃。  
　　按量子的倍数为字符串分配内存，内存分配器就可以回收用过的内存块，通常这些回收的内存块马上就可以被其它的 CString 对象重新用到，这样就可以保证碎片最少。分配器的功能加强了，应用程序用到的内存就能尽可能保持最小，这样的程序就可以运行几个星期或几个月而不出现问题。  
　　题外话：很多年以前，我们在 CMU 写一个交互式系统的时候，一些对内存分配器的研究显示出它往往产生很多内存碎片。Jim Mitchell，现在他在 Sun Microsystems 工作，那时侯他创造了一种内存分配器，它保留了一个内存分配状况的运行时统计表，这种技术和当时的主流分配器所用的技术都不同，且较为领先。当一个内存块需要被分割得比某一个值小的话，他并不分割它，因此可以避免产生太多小到什么事都干不了的内存碎片。事实上他在内存分配器中使用了一个浮动指针，他认为：与其让指令做长时间的存取内存操作，还不如简单的忽略那些太小的内存块而只做一些浮动指针的操作。（His observation was that the long-term saving in instructions by not having to ignore unusable small storage chunks far and away exceeded the additional cost of doing a few floating point operations on an allocation operation.）他是对的。  
　　永远不要认为所谓的“最优化”是建立在每一行代码都高速且节省内存的基础上的，事实上，高速且节省内存应该是在一个应用程序的整体水平上考虑的。在软件的整体水平上，只使用最小内存的字符串分配策略可能是最糟糕的一种方法。  
　　如果你认为优化是你在每一行代码上做的那些努力的话，你应该想一想：在每一行代码中做的优化很少能真正起作用。你可以看我的另一篇关于优化问题的文章《Your Worst Enemy for some thought-provoking ideas》。  
　　记住，+= 运算符只是一种特例，如果你写成下面这样：

CString s = SomeCString1 + SomeCString2 + SomeCString3 + "," + SomeCString4;

则每一个 + 的应用会造成一个新的字符串被创建和一次复制操作。  
  
http://www.vckbase.com/document/image/paragraph.gif **总结**  
  
　　以上是使用 CString 的一些技巧。我每天写程序的时候都会用到这些。CString 并不是一种很难使用的类，但是 MFC 没有很明显的指出这些特征，需要你自己去探索、去发现。

CString类所有成员函数详解

VC里CString是我们最常用的类之一，我们觉得对它很熟悉了，可是你知道它的所有用法吗？

还是系统的学习一下吧，认真看完本文就OK了。

下面开始：

CString::Compare

int Compare( LPCTSTR lpsz ) const;

返回值 字符串一样 返回0

小于lpsz 返回-1

大于lpsz 返回1

区分大小字符

CString s1( "abc" );

CString s2( "abd" );

ASSERT( s1.Compare( s2 ) == -1 );

ASSERT( s1.Compare( "abe" ) == -1 );

CString::CompareNoCase

int CompareNoCase( LPCTSTR lpsz ) const;

返回值 字符串一样 返回0

小于lpsz 返回-1

大于lpsz 返回1

不区分大小字符

CString::Collate

int Collate( LPCTSTR lpsz ) const;

同CString::Compare

CString::CollateNoCase

int CollateNocase( LPCTSTR lpsz ) const;

同CString::CompareNoCase

CString::CString

CString( );

CString( const CString& stringSrc );

CString( TCHAR ch, int nRepeat = 1 );

CString( LPCTSTR lpch, int nLength );

CString( const unsigned char\* psz );

CString( LPCWSTR lpsz );

CString( LPCSTR lpsz );

例子最容易说明问题

CString s1;

CString s2( "cat" );

CString s3 = s2;

CString s4( s2 + " " + s3 );

CString s5( 'x' ); // s5 = "x"

CString s6( 'x', 6 ); // s6 = "xxxxxx"

CString s7((LPCSTR)ID\_FILE\_NEW); // s7 = "Create a new document"

CString city = "Philadelphia";

CString::Delete

int Delete( int nIndex, int nCount = 1);

返回值是被删除前的字符串的长度

nIndex是第一个被删除的字符，nCount是一次删除几个字符。根据我实验得出的结果：当nCount>要删除字符串的最大长度(GetCount() - nIndex)时会出错，当nCount过大，没有足够的字符删除时，此函数不执行。

例子

CString str1,str2,str3;

char a;

str1 = "nihao";

str2 = "nIhao";

int x;

// int i=(str1 == str2);

str1.Delete(2,3);

如果nCount(3) > GetCount() – nIndex (5-2)就会执行错误

CString::Empty

Void Empty( );

没有返回值 清空操作;

例子

CString s( "abc" );

s.Empty();

ASSERT( s.GetLength( ) == 0 );

CString::Find

int Find( TCHAR ch ) const;

int Find( LPCTSTR lpszSub ) const;

int Find( TCHAR ch, int nStart ) const;

int Find( LPCTSTR lpszSub, int nStart ) const;

返回值 不匹配的话返回 -1; 索引以0 开始

nStar 代表以索引值nStart 的字符开始搜索 ,

即为包含以索引nStart字符后的字符串

例子

CString s( "abcdef" );

ASSERT( s.Find( 'c' ) == 2 );

ASSERT( s.Find( "de" ) == 3 );

Cstring str(“The stars are aligned”);

Ing n = str.Find('e',5);

ASSERT(n == 12)

CString::FindOneOf

int FindOneOf( LPCTSTR lpszCharSet ) const;

返回值 不匹配的话返回 -1; 索引以0 开始

注意::返回此字符串中第一个在lpszCharSet中 也包括字符并且从零开始的索引值

例子

CString s( "abcdef" );

ASSERT( s.FindOneOf( "xd" ) == 3 ); // 'd' is first match.

CString::Format

void Format( LPCTSTR lpszFormat, ... );

void Format( UINT nFormatID, ... );

lpszFormat 一个格式控制字符串

nFormatID 字符串标识符

例子

CString str;

Str.Format(“%d”,13);

此时Str为13

CString::GetAt

TCHAR GetAt( int nIndex ) const;

返回标号为nIndex的字符，你可以把字符串理解为一个数组，GetAt类似于[].注意nIndex的范围，如果不合适会有调试错误。

CString::GetBuffer

LPTSTR GetBuffer( int nMinBufLength );

返回值

一个指向对象的（以空字符结尾的）字符缓冲区的LPTSTR 指针。

参数

nMinBufLength

字符缓冲区的以字符数表示的最小容量。这个值不包括一个结尾的空字符的空间。

说明

此成员函数返回一个指向CString 对象的内部字符缓冲区的指针。返回的LPTSTR 不是const，因此可以允许直接修改CString 的内容。如果你使用由GetBuffer 返回的指针来改变字符串的内容，你必须在使用其它的CString 成员函数之前调用ReleaseBuffer 函数。

在调用ReleaseBuffer 之后，由GetBuffer 返回的地址也许就无效了，因为其它的CString 操作可能会导致CString 缓冲区被重新分配。如果你没有改变此CString 的长度，则缓冲区不会被重新分配。当此CString 对象被销毁时，其缓冲区内存将被自动释放。

注意，如果你自己知道字符串的长度，则你不应该添加结尾的空字符。但是，当你用ReleaseBuffer 来释放该缓冲区时，你必须指定最后的字符串长度。如果你添加了结尾的空字符， 你应该给ReleaseBuffer 的长度参数传递-1 ，ReleaseBuffer 将对该缓冲区执行strlen 来确定它的长度。

下面的例子说明了如何用CString::GetBuffer。

// CString::GetBuffer 例子

CString s( "abcd" );

#ifdef \_DEBUG

afxDump << "CString s " << s << "\n";

#endif

LPTSTR p = s.GetBuffer( 10 );

strcpy( p, "Hello" ); // 直接访问CString 对象。

s.ReleaseBuffer( );

#ifdef \_DEBUG

afxDump << "CString s " << s << "\n";

#endif

CString::GetLength

int GetLength( ) const;

返回值

返回字符串中的字节计数。

说明

此成员函数用来获取这个CString 对象中的字节计数。这个计数不包括结尾的空字符。

对于多字节字符集（MBCS），GetLength 按每一个8 位字符计数；即，在一个多字节字符中的开始和结尾字节被算作两个字节。

示例

下面的例子说明了如何使用CString::GetLength。

// CString::GetLength 示例

CString s( "abcdef" );

ASSERT( s.GetLength() == 6 );

CString::Insert

int Insert( int nIndex, TCHAR ch );

int Insert( int nIndex, LPCTSTR pstr );

返回修改后的长度，nIndex是字符（或字符串）插入后的索引号例子

CString str( “HockeyBest”);

int n = str.Insert( 6, “is” );

ASSERT( n == str.GetLength( ) );

printf( “1: %s\n”, ( LPCTSTR ) str );

n = str.Insert( 6, ' ' );

ASSERT( n == str.GetLength( ) );

printf ( “2: %s\n”, (LPCTSTR) STR );

n = str.Insert(555, ‘1’);

ASSERT( n == str.GetLength ( ) );

printf ( “3: %s\n”, ( LPCTSTR ) str );

输出

1. Hockeyis Best

2. Hockey is Best

3. Hockey is Best!

CString::IsEmpty

BOOL IsEmpty( ) const;

返回值

如果CString 对象的长度为0，则返回非零值；否则返回0。

说明

此成员函数用来测试一个CString 对象是否是空的。

示例

下面的例子说明了如何使用CString::IsEmpty。

// CString::IsEmpty 示例

CString s;

ASSERT( s.IsEmpty() );

请参阅 CString::GetLength

CString::Left

CString Left( int nCount ) const;

throw( CMemoryException );

返回的字符串是前nCount个字符。

例子

CString s( \_T("abcdef") );

ASSERT( s.Left(2) == \_T("ab") );

CString::LoadString

BOOL LoadString( UINT nID );

throw( CMemoryException );

返回值

如果加载资源成功则返回非零值；否则返回0。

nID 一个Windows 字符串资源ID。

说明 此成员函数用来读取一个由nID 标识的Windows 字符串资源，并放入一个已有CString 对象中。

示例

下面的例子说明了如何使用CString::LoadString。

// CString::LoadString 示例

#define IDS\_FILENOTFOUND 1

CString s;

if (! s.LoadString( IDS\_FILENOTFOUND ))

CString::MakeLower

void MakeLower( );

改变字符的小写

CString::MakeReverse

void MakeReverse( );

字符倒置

CString::MakeUpper

void MakeUpper( );

改变字符的大写

CString::Mid

CString Mid( int nFirst ) const;

CString Mid( int nFirst, int nCount ) const;

nCount代表要提取的字符数, nFirst代表要提取的开始索引位置

例子

CString s( \_T("abcdef") );

ASSERT( s.Mid( 2, 3 ) == \_T("cde") );

CString::ReleaseBuffer

void ReleaseBuffer( int nNewLength = -1 );

参数

nNewLength

此字符串的以字符数表示的新长度，不计算结尾的空字符。如果这个字

符串是以空字符结尾的，则参数的缺省值－1 将把CString 的大小设置为

字符串的当前长度。

说明

使用ReleaseBuffer 来结束对由GetBuffer 分配的缓冲区的使用。如果你知道缓

冲区中的字符串是以空字符结尾的，则可以省略nNewLength 参数。如果字符

串不是以空字符结尾的，则可以使用nNewLength 指定字符串的长度。在调用

ReleaseBuffer 或其它CString 操作之后，由GetBuffer 返回的地址是无效的。

示例

下面的例子说明了如何使用CString::ReleaseBuffer。

// CString::ReleaseBuffer 示例

CString s;

s = "abc";

LPTSTR p = s.GetBuffer( 1024 );

strcpy(p, "abc"); // 直接使用该缓冲区

ASSERT( s.GetLength() == 3 ); // 字符串长度 = 3

s.ReleaseBuffer(); // 释放多余的内存，现在p 无效。

ASSERT( s.GetLength() == 3 ); // 长度仍然是3

CString::Remove

int CString::Remove ( TCHAR ch );

返回值

返回从字符串中移走的字符数。如果字符串没有改变则返回零。

参数

ch

要从一个字符串中移走的字符。

说明

此成员函数用来将ch 实例从字符串中移走。与这个字符的比较是区分大小写

的。

示例

// 从一个句子中移走小写字母'c':

CString str (“This is a test.”);

int n = str.Remove( 't' );

ASSERT( n == 2 );

ASSERT( str ==“This is a es. ” );

CString::Replace

int Replace( TCHAR chOld, TCHAR chNew );

int Replace( LPCTSTR lpszOld, LPCTSTR lpszNew );

返回值

返回被替换的字符数。如果这个字符串没有改变则返回零。

参数

chOld

要被chNew 替换的字符

chNew

要用来替换chOld 的字符。

lpszOld

一个指向字符串的指针，该字符串包含了要被lpszNew 替换的字符。

lpszNew

一个指向字符串的指针，该字符串包含了要用来替换lpszOld 的字符。

说明

此成员函数用一个字符替换另一个字符。函数的第一个原形在字符串中用chNew

现场替换chOld。函数的第二个原形用lpszNew 指定的字符串替换lpszOld 指定

的子串。

在替换之后，该字符串有可能增长或缩短；那是因为lpszNew 和lpszOld 的长度

不需要是相等的。两种版本形式都进行区分大小写的匹配。

关于CString总结    
   
前言:串操作是编程中最常用也最基本的操作之一.  做为VC程序员,无论是菜鸟或高手都曾用过CString.而且好像实际编程中很难离得开它（虽然它不是标准Ｃ++中的库）.因为MFC中提供的这个类对我们操作字串实在太方便了,CString不仅提供各种丰富的操作函数、操作符重载，使我们使用起串起来更象basic中那样直观；而且它还提供了动态内存分配，使我们减少了多少字符串数组越界的隐患。但是，我们在使用过程中也体会到CString简直太容易出错了，而且有的不可捉摸。所以有许多高人站过来，建议抛弃它。    
       在此，我个人认为：CString封装得确实很完美，它有许多优点，如“容易使用  ，功能强，动态分配内存，大量进行拷贝时它很能节省内存资源并且执行效率高，与标准Ｃ完全兼容，同时支持多字节与宽字节，由于有异常机制所以使用它安全方便”  其实，使用过程中之所以容易出错，那是因为我们对它了解得还不够，特别是它的实现机制。因为我们中的大多数人，在工作中并不爱那么深入地去看关于它的文档，何况它还是英文的。      
     由于前几天我在工作中遇到了一个本不是问题但却特别棘手、特别难解决而且莫名惊诧的问题。最后发现是由于CString引发的，后来，没办法，我把整个CString的实现全部看了一遍，才慌然大悟，并彻底弄清了问题的原因(这个问题，我已在csdn上开贴)。在此，我想把我的一些关于CString的知识总结一番，以供他（她）人借鉴，也许其中有我理解上的错误，望发现者能通知我，不胜感谢。    
   
1  CString实现的机制.    
     CString是通过“引用”来管理串的，“引用”这个词我相信大家并不陌生，象Window内核对象、COM对象等都是通过引用来实现的。而CString也是通过这样的机制来管理分配的内存块。实际上CString对象只有一个指针成员变量,所以任何CString实例的长度只有4字节.    
             即:  int  len  =  sizeof(CString);//len等于4    
这个指针指向一个相关的引用内存块，如图:  CString  str("abcd");    
                                                                                       \_\_\_    
         \_\_\_\_\_\_\_\_\_\_\_\_                                                      ¦        ¦      
         ¦                          ¦                                                    ¦        ¦      
         ¦  0x04040404    ¦                                                    ¦        ¦    head部，为引用内存块相关信息    
         ¦\_\_\_\_\_\_\_\_\_\_\_\_  ¦                                                    ¦        ¦    
                 str                                                                ¦\_\_\_  ¦    
                                                                                       ¦'a'  ¦  0x40404040    
                                                                                       ¦'b'  ¦    
                                                                                       ¦'c'  ¦    
                                                                                       ¦'d'  ¦    
                                                                                       ¦  0    ¦    
   
正因为如此，一个这样的内存块可被多个CString所引用，例如下列代码：    
CString  str("abcd");    
CString  a  =  str;    
CString  b(str);    
CString  c;    
c  =  b;    
上面代码的结果是：上面四个对象(str,a,b,c)中的成员变量指针有相同的值，都为0x40404040.而这块内存块怎么知道有多少个CString引用它呢？同样，它也会记录一些信息。如被引用数，串长度，分配内存长度。    
这块引用内存块的结构定义如下：    
struct  CStringData    
{    
   long  nRefs;              //表示有多少个CString  引用它.  4    
   int  nDataLength;    //串实际长度.  4    
   int  nAllocLength;  //总共分配的内存长度（不计这头部的12字节）.  4    
};    
由于有了这些信息，CString就能正确地分配、管理、释放引用内存块。    
如果你想在调试程序的时候获得这些信息。可以在Watch窗口键入下列表达式：    
(CStringData\*)((CStringData\*)(this->m\_pchData)-1)或    
(CStringData\*)((CStringData\*)(str.m\_pchData)-1)//str为指CString实例    
   
正因为采用了这样的好机制，使得CString在大量拷贝时，不仅效率高，而且分配内存少。