



[General Programming](#) » [String handling](#) » [General](#), [Beginner](#)

The Complete Guide to C++ Strings, Part II - String Wrapper Classes

By [Michael Dunn](#), [Nishant Sivakumar](#)

A guide to the string wrapper classes provided by Visual C++ and class libraries

VC6Win2K, WinXP, Visual Studio, MFC, ATL, WTL, STL, Dev

Posted: **6 Oct 2002**

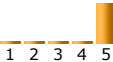
Updated: **12 Oct 2002**

Views: **534,785**

Bookmarked: **387 times**

 Prize winner in Competition "MFC/C++ Sep 2002"

198 votes for this article. 

Popularity: [11.12](#) Rating: **4.84** out of 5 

Introduction

Since C-style strings can be error-prone and difficult to manage, not to mention a target for hackers looking for buffer overrun bugs, there are lots of string wrapper classes. Unfortunately, it's not always clear which class should be used in some situations, nor how to convert from a C-style string to a wrapper class.

This article covers all the string types in the Win32 API, MFC, STL, WTL, and the Visual C++ runtime library. I will describe the usage of each class, how to construct objects, and how to convert to other classes. Nish has also contributed the section on managed strings and classes in Visual C++ 7.

In order to get the full benefit from this article, you *must* understand the different character types and encodings, as I covered in [Part I](#).

Rule #1 of string classes

Casts are bad, unless they are explicitly documented.

What prompted me to write these two articles was the frequent questions about how to convert string type X to type Z, where the poster was using a cast and didn't understand why the code didn't work. The various string types, especially `BSTR`, are not concisely documented in any one place, so I imagine some people were throwing in casts and hoping it would work.

A cast does **not** do any conversion to a string, **unless** the source string is a wrapper class with an explicitly documented conversion operator. A cast of a string literal does nothing to the string, so writing something like:

```
void SomeFunc ( LPCWSTR wdestr );

main()
{
    SomeFunc ( (LPCWSTR) "C:\\foo.txt" ); // WRONG!
}
```

will fail 100% of the time. It will compile, because the cast overrides the compiler's type-checking. But just because it compiles, doesn't mean the code is correct.

In the examples that follow, I will point out when casts are legal.

C-style strings and typedefs

As I covered in [Part I](#), Windows APIs are defined and documented in terms of `TCHAR`s, which can be MBCS or Unicode characters depending on whether you define the `_MBCS` or `_UNICODE` symbol when compiling. You should refer to [Part I](#) for a full description of `TCHAR`, but I will list the character typedefs here for convenience.

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*)

One additional character type is the [OLECHAR](#). This represents the character type used in Automation interfaces (such as the interfaces exposed by Word so you can manipulate documents). This type is normally defined as [wchar_t](#), however if you define the [OLE2ANSI](#) preprocessor symbol, [OLECHAR](#) will be defined as the [char](#) type. I know of no reason these days to define [OLE2ANSI](#) (it hasn't been used by Microsoft since the days of MFC 3), so from now on I will treat an [OLECHAR](#) as a Unicode character.

Here are the [OLECHAR](#)-related typedefs you will see:

Type	Meaning
OLECHAR	Unicode character (wchar_t)
LPOLESTR	string of OLECHAR (OLECHAR*)
LPCOLESTR	constant string of OLECHAR (const OLECHAR*)

There are also two macros used around string and character literals so that the same code can be used for both MBCS and 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 .

There are also variants on [_T](#) that you might encounter in documentation or sample code. There are *four* equivalent macros -- [TEXT](#), [_TEXT](#), [__TEXT](#), and [__T](#) -- that all do the same thing.

Strings in COM - BSTR and VARIANT

Many Automation and other COM interfaces use [BSTR](#) for strings, and [BSTRs](#) have a few pitfalls, so I will give [BSTR](#) its own section here.

[BSTR](#) is a hybrid between Pascal-style strings (where the length is stored explicitly along with the data) and C-style strings (where the string length must be calculated by looking for a terminating zero character). A [BSTR](#) is a Unicode string that has its length prepended, and is also terminated by a zero character. Here is an example of "Bob" as a [BSTR](#):

```
06 00 00 0042 006F 0062 0000 00
--length--  B   o   b   EOS
```

Notice how the length of the string is prepended to the string data. It is a [DWORD](#), and holds the number of *bytes* in the string, *not counting* the terminating zero. In this case, "Bob" contains 3 Unicode characters (not counting the terminating zero), for a total of 6 bytes. The length field is present so that when a [BSTR](#) is marshaled between processes or computers, the COM library knows how much data to transfer. (As a side note, a [BSTR](#) can hold any arbitrary block of data, not just characters, and can contain embedded zero characters. However, for the purposes of this article, I will not consider such cases.)

A [BSTR](#) variable in C++ is actually a pointer to the first character of the string. In fact, the type [BSTR](#) is defined this way:

```
typedef OLECHAR* BSTR;
```

This is very unfortunate, because in reality a **BSTR** is *not* the same as a Unicode string. That typedef defeats type-checking and allows you to freely mix **LPOLESTRs** and **BSTRs**. Passing a **BSTR** to a function expecting a **LPCOLESTR** (or **LPCWSTR**) is safe, however the reverse is not. Therefore, it's important to be aware of the exact type of string that a function expects, and pass the correct type of string.

To see why it is not safe to pass a **LPCWSTR** to a function expecting a **BSTR**, remember that the four bytes immediately before the string must store its length. There is no such length with a **LPCWSTR**. If the **BSTR** needs to be marshaled to another process (for example, an instance of Word that you are controlling), the COM library will look for that length and find garbage, or some other variable on your stack, or other random data. This will either cause the method to fail, or even crash if the perceived length is too long.

There are several APIs that operate on **BSTRs**, however the two most important ones are the functions that create and destroy a **BSTR**. They are **SysAllocString()** and **SysFreeString()**. **SysAllocString()** copies a Unicode string into a **BSTR**, while **SysFreeString()** frees the memory used by a **BSTR**.

```
BSTR bstr = NULL;

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

if ( NULL == bstr )
    // out of memory error

// Use bstr here...

SysFreeString ( bstr );
```

Naturally, the various **BSTR** wrapper classes take care of the memory management for you.

The other type used in Automation interfaces is **VARIANT**. This is used to send data between typeless languages like JScript and VBScript, as well as Visual Basic in some cases. A **VARIANT** can contain data of many different types, such as **long** and **IDispatch***. When a **VARIANT** contains a string, it is stored as a **BSTR**. I will have more to say about **VARIANTs** when I cover the **VARIANT** wrapper classes later.

String wrapper classes

Now that I've covered the various types of strings, I'll demonstrate the wrapper classes. For each one, I'll show how to construct an object and how to convert it to a C-style string pointer. The C-style pointer is often necessary for an API call, or to construct an object of a different string class. I will not cover other operators the classes provide, such as sorting or comparison.

Once again, do **not** blindly cast objects unless you understand exactly what the resulting code will do.

Classes provided by the CRT

_bstr_t

_bstr_t is a complete wrapper around a **BSTR**, and in fact it hides the underlying **BSTR**. It provides various constructors, as well as operators to access the underlying C-style string. However, there is no operator to access the **BSTR** itself, so a **_bstr_t** cannot be passed as an **[out]** parameter to COM methods. If you need a **BSTR*** to use as a parameter, it is easier to the ATL class **CComBSTR**.

A **_bstr_t** can be passed to a function that takes a **BSTR**, but only because of three coincidences. First, **_bstr_t** has a conversion function to **wchar_t***; second, **wchar_t*** and **BSTR** appear the same to the compiler because of the definition of **BSTR**; and third, the **wchar_t*** that a **_bstr_t** keeps internally points to a block of memory that follows the **BSTR** format. So even though there is no documented conversion to **BSTR**, it happens to work.

```

// 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 );

```

Note that `_bstr_t` also has conversion operators for `char*` and `wchar_t*`. This is a questionable design, because even though those are non-constant string pointers, you must not use those pointers to modify the buffer, because that could break the internal `BSTR` structure.

`_variant_t`

`_variant_t` is a complete wrapper around a `VARIANT`, and provides many constructors and conversion functions to operate on the multitude of types that a `VARIANT` can contain. I will only cover the string-related operations here.

```

// 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

```

Note that the `_variant_t` methods can throw exceptions if the type conversion cannot be made, so be prepared to catch `_com_error` exceptions.

Also note that there is no direct conversion from `_variant_t` to an MBCS string. You will need to make an interim `_bstr_t` variable, use another string class that provides the Unicode to MBCS conversion, or use an ATL conversion macro.

Unlike `_bstr_t`, a `_variant_t` can be passed directly as a parameter to a COM method. `_variant_t` derives from the `VARIANT` type, so passing a `_variant_t` in place of a `VARIANT` is allowed by C++ language rules.

STL classes

STL just has one string class, `basic_string`. A `basic_string` manages a zero-terminated array of characters. The character type is given in the `basic_string` template parameter. In general, a `basic_string` should be treated as an opaque object. You can get a read-only pointer to the internal buffer, but any write operations must use `basic_string` operators and methods.

There are two predefined specializations for `basic_string`: `string`, which contains `chars`, and `wstring`, which contains `wchar_ts`. There is no built-in `TCHAR` specialization, but you can use the one listed below.

```
// Specializations
typedef basic_string<TCHAR> 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
```

Unlike `_bstr_t`, a `basic_string` cannot directly convert between character sets. However, you can pass the pointer returned by `c_str()` to another class's constructor if the constructor accepts the character type, for example:

```
// 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 classes

CComBSTR

`CComBSTR` is ATL's `BSTR` wrapper, and is more useful in some situations than `_bstr_t`. Most notably, `CComBSTR` allows access to the underlying `BSTR`, which means you can pass a `CComBSTR` object to COM methods, and the `CComBSTR` object will automatically manage the `BSTR` memory for you. For example, say you wanted to call methods of this interface:

```
// Sample interface:
struct IStuff : public IUnknown
{
    // Boilerplate COM stuff omitted...
    STDMETHOD(SetText)(BSTR bsText);
    STDMETHOD(GetText)(BSTR* pbsText);
};
```

`CComBSTR` has an `operator BSTR` method, so it can be passed directly to `SetText()`. There is also an `operator &` that returns a `BSTR*`, so you can use the `&` operator on a `CComBSTR` object to pass it to a function that takes a `BSTR*`.

```
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` has similar constructors to `_bstr_t`, however there is no built-in converter to an MBCS string. For that, you can use an ATL conversion macro.

```

// 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 );

```

Note that in the last example, the `Detach()` method is used. After calling that method, the `CComBSTR` object no longer manages its `BSTR` or the associated memory. That's why the `SysFreeString()` call is necessary on `bstr4`.

As a footnote, the `operator &` override means you can't use `CComBSTR` directly in some STL collections, such as `list`. The collections require that the `&` operator return a pointer to the contained class, but applying `&` to a `CComBSTR` returns a `BSTR*`, not a `CComBSTR*`. However, there is an ATL class to overcome this, `CAdapt`. For example, to make a list of `CComBSTR`, declare it like this:

```
std::list< CAdapt<CComBSTR> > bstr_list;
```

`CAdapt` provides the operators required by the collection, but it is invisible to your code; you can use `bstr_list` just as if it were a list of `CComBSTR`.

CComVariant

`CComVariant` is a wrapper around a `VARIANT`. However, unlike `_variant_t`, the `VARIANT` is not hidden, and in fact you need to access the members of the `VARIANT` directly. `CComVariant` provides many constructors to operate on the multitude of types that a `VARIANT` can contain. I will only cover the string-related operations here.

```

// 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

```

Unlike `_variant_t`, there are no conversion operators to the various `VARIANT` types. As shown above, you must access the `VARIANT` members directly and ensure that the `VARIANT` holds data of the type you expect. You can call the `ChangeType()` method if you need to convert a `CComVariant`'s data to a `BSTR`.

```

CComVariant v4 = ... // Init v4 from somewhere
CComBSTR bs3;

    if ( SUCCEEDED( v4.ChangeType ( VT_BSTR ) ))
        bs3 = v4.bstrVal;

```

As with `_variant_t`, there is no direct conversion to an MBCS string. You will need to make an interim `_bstr_t` variable, use another string class that provides the Unicode to MBCS conversion, or use an ATL conversion macro.

ATL conversion macros

ATL's string conversion macros are a very convenient way to convert between character encodings, and are especially useful in function calls. They are named according to the scheme `[source type]2[new type]` or `[source type]2C[new type]`. Macros named with the second form convert to a constant pointer (thus the "C" in the name). The type abbreviations are:

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)

So, for example, `W2A()` converts a Unicode string to an MBCS string, and `T2CW()` converts a `TCHAR` string to a constant Unicode string.

To use the macros, first include the `atlconv.h` header file. You can do this even in non-ATL projects, since that header file has no dependencies on other parts of ATL, and doesn't require a `_Module` global variable. Then, when you use a conversion macro in a function, put the `USES_CONVERSION` macro at the beginning of the function. This defines some local variables used by the macros.

When the destination type is anything other than `BSTR`, the converted string is stored on the stack, so if you want to keep the string around for longer than the current function, you'll need to copy the string into another string class. When the destination type *is* `BSTR`, the memory is not automatically freed, so you must assign the return value to a `BSTR` variable or a `BSTR` wrapper class to avoid memory leaks.

Here are some examples showing various conversion macros:

```

// Functions taking various strings:
void Foo ( LPCWSTR wstr );
void Bar ( BSTR bstr );
// Functions returning strings:
void Baz ( BSTR* pbstr );

#include <atlconv.h>

main()
{
using std::string;
USES_CONVERSION;    // declare locals used by the ATL macros

// Example 1: Send an MBCS string to Foo()
LPCWSTR psz1 = "Bob";
string str1 = "Bob";

    Foo ( A2CW(psz1) );
    Foo ( A2CW(str1.c_str()) );

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

```

As you can see, the macros are very handy when passing parameters to a function if you have a string in one format and the function takes a different format.

MFC classes

CString

An MFC **CString** holds **TCHARs**, so the exact character type depends on the preprocessor symbols you have defined. In general, a **CString** is like an STL **string**, in that you should treat it as an opaque object and modify it only with **CString** methods. One nice advantage **CString** has over the STL **string** is that it has constructors that accept both MBCS and Unicode strings, and it has a converter to **LPCTSTR**, so you can pass a **CString** object directly to a function that accepts an **LPCTSTR**; there is no **c_str()** method you have to call.

```

// Constructing
CString s1 = "char string"; // construct from a LPCWSTR
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 );

```

You can also load a string from your string table. There is a `CString` constructor that will do it, along with `LoadString()`. The `Format()` method can optionally read a format string from the string table as well.

```
// 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, ... );
```

That first constructor looks odd, but that is actually the documented that way to load a string.

Note that the **only** legal cast you can apply to a `CString` is a cast to `LPCTSTR`. Casting to an `LPTSTR` (that is, a non-`const` pointer) is wrong. Getting in the habit of casting a `CString` to an `LPTSTR` will only hurt yourself, as when the code does break later on, you might not see why, because you used the same code elsewhere and it happened to work. The correct way to get a non-const pointer to the buffer is the `GetBuffer()` method.

As an example of the correct usage, consider the case of setting the text of an item in a list control:

```
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
```

The `pszText` member is an `LPTSTR`, a non-`const` pointer, therefore you call `GetBuffer()` on `str`. The parameter to `GetBuffer()` is the minimum length you want `CString` to allocate for the buffer. If for some reason you wanted a modifiable buffer large enough to hold 1K `TCHARs`, you would call `GetBuffer(1024)`. Passing 0 as the length just returns a pointer to the current contents of the string.

The crossed-out line above will compile, and it will even work, *in this case*. But that doesn't mean the code is correct. By using the non-`const` cast, you're breaking object-oriented encapsulation and assuming something about the internal implementation of `CString`. If you make a habit of casting like that, you will eventually run into a case where the code breaks, and you'll wonder why it isn't working, because you use the same code everywhere else and it (apparently) works.

You know how people are always complaining about how buggy software is these days? Bugs are caused by the programmers writing incorrect code. Do you really want to write code you *know* is wrong, and thus contribute to the perception that all software is buggy? Take the time to learn the correct way of using a `CString` and have your code work 100% of the time.

`CString` also has two functions that create a `BSTR` from the `CString` contents, converting to Unicode if necessary. They are `AllocSysString()` and `SetSysString()`. Aside from the `BSTR*` parameter that `SetSysString()` takes, they work identically.

```
// Converting to BSTR
CString s5 = "Bob!";
BSTR bs1 = NULL, bs2 = NULL;

bs1 = s5.AllocSysString();
s5.SetSysString ( &bs2 );

// ...
SysFreeString ( bs1 );
SysFreeString ( bs2 );
```

COleVariant

`COleVariant` is pretty similar to `CComVariant`. `COleVariant` derives from `VARIANT`, so it can be passed to a function that takes a `VARIANT`. However, unlike `CComVariant`, `COleVariant` only has an `LPCTSTR` constructor. There are not separate constructors for `LPCSTR` and `LPCWSTR`. In most cases this is not a problem, since your strings will likely be `LPCTSTR`s anyway, but it is a point to be aware of. `COleVariant` also has a constructor that accepts a `CString`.

```
// Constructing
CString s1 = _T("tchar string");
COleVariant v1 = _T("Bob"); // construct from an LPCTSTR
COleVariant v2 = s1; // copy from a CString
```

As with `CComVariant`, you must access the `VARIANT` members directly, using the `ChangeType()` method if necessary to convert the `VARIANT` to a string. However, `COleVariant::ChangeType()` throws an exception if it fails, instead of returning a failure `HRESULT` code.

```
// 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 classes

CString

WTL's `CString` behaves exactly like MFC's `CString`, so refer to the description of the MFC `CString` above.

CLR and VC 7 classes

`System::String` is the .NET class for handling strings. Internally, a `String` object holds an immutable sequence of characters. Any `String` method that supposedly manipulates the `String` object actually returns a new `String` object, because the original `String` is immutable. A peculiarity of `Strings` is that if you have more than one `String` containing the same series, of characters all of them actually refer the same object. The Managed Extensions to C++ have a new string literal prefix `S`, which is used to represent a managed string literal.

```
// Constructing
String* ms = S"This is a nice managed string";
```

You can construct a `String` object by passing an unmanaged string, but this is slightly less efficient than when you construct a `String` object by passing a managed string. This is because all instances of identical `S` prefixed strings represent the same object, but this is not true for unmanaged strings. The following code will make this clear:

```
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
```

The right way to compare strings that may not have been created using `S` prefixed strings is to use the `String::CompareTo()` method as shown below:

```
Console::WriteLine ( ms1->CompareTo(ms2) );
Console::WriteLine ( ms1->CompareTo(ms3) );
```

Both the above lines will print 0, which means the strings are equal.

Converting between a `String` and the MFC 7 `CString` is easy. `CString` has a converter to `LPCTSTR` and `String` has two constructors that take a `char*` and `wchar_t*`, therefore you can pass a `CString` straight to a `String` constructor.

```
CString s1 ( "hello world" );
String* s2 ( s1 ); // copy from a CString
```

Converting the other way works similarly:

```
String* s1 = S"Three cats";
CString s2 ( s1 );
```

This might puzzle you a bit, but it works because starting with VS.NET, `CString` has a constructor that accepts a `String` object:

```
CStringT ( System::String* pString );
```

For some speedy manipulations, you might sometimes want to access the underlying string:

```
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()` returns a `const __wchar_t*` to the underlying string which we need to pin down as otherwise the garbage collector might move the string in memory while we are manipulating its contents.

Using string classes with printf-style formatting functions

You must pay careful attention when using string wrapper classes with `printf()` or any function that works the way `printf()` does. This includes `sprintf()` and its variants, as well as the `TRACE` and `ATLTRACE` macros. Because there is no type-checking done on the additional parameters to the functions, you must be careful to only pass a C-style string pointer, not a complete string object.

So for example, to pass a string in a `_bstr_t` to `ATLTRACE()`, you *must* explicitly write the `(LPCSTR)` or `(LPCWSTR)` cast:

```
_bstr_t bs = L"Bob!";

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

If you forget the cast and pass the entire `_bstr_t` object, the trace message will display meaningless output, since what will be pushed on the stack is whatever internal data the `_bstr_t` variable keeps.

Summary of all the classes

The usual way of converting between two string classes is to take the source string, convert it to a C-style string pointer, and then pass the pointer to a constructor in the destination type. So here is a chart showing how to convert a string to a C-style pointer, and which classes can be constructed from C-style pointers.

Class	string type	convert to <code>char*</code> ?	convert to <code>const char*</code> ?	convert to <code>wchar_t*</code> ?	convert to <code>const wchar_t*</code> ?	convert to <code>BSTR</code> ?	construct from <code>char*</code> ?	construct from <code>wchar_t*</code> ?
<code>_bstr_t</code>	<code>BSTR</code>	yes, cast ¹	yes, cast	yes, cast ¹	yes, cast	yes ²	yes	yes
<code>_variant_t</code>	<code>BSTR</code>	no	no	no	cast to <code>bstr t</code> ³	cast to <code>bstr t</code> ³	yes	yes
<code>string</code>	<code>MBCS</code>	no	yes, <code>c_str()</code> method	no	no	no	yes	no
<code>wstring</code>	Unicode	no	no	no	yes, <code>c_str()</code> method	no	no	yes
<code>CComBSTR</code>	<code>BSTR</code>	no	no	no	yes, cast to <code>BSTR</code>	yes, cast	yes	yes
<code>CComVariant</code>	<code>BSTR</code>	no	no	no	yes ⁴	yes ⁴	yes	yes
<code>CString</code>	<code>TCHAR</code>	no ⁶	in MBCS builds, cast	no ⁶	in Unicode builds, cast	no ⁵	yes	yes
<code>ColeVariant</code>	<code>BSTR</code>	no	no	no	yes ⁴	yes ⁴	in MBCS builds	in Unicode builds

¹ Even though `_bstr_t` provides conversion operators to non-`const` pointers, modifying the underlying buffer may cause a GPF if you overrun the buffer, or a leak when the `BSTR` memory is freed.

² A `_bstr_t` holds a `BSTR` internally in a `wchar_t*` variable, so you can use the `const wchar_t*` converter to retrieve the `BSTR`. This is an implementation detail, so use this with caution, as it may change in the future.

³ This will throw an exception if the data cannot be converted to a `BSTR`.

⁴ Use `ChangeType()` then access the `bstrVal` member of the `VARIANT`. In MFC, this will throw an exception if the data cannot be converted.

⁵ There is no `BSTR` conversion function, however the `AllocSysString()` method returns a new `BSTR`.

⁶ You can temporarily get a non-const `TCHAR` pointer using the `GetBuffer()` method.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

About the Authors

Michael Dunn



Member

Michael lives in sunny Sunnyvale, California, and is still trying to break the habit (as a Buffy fan) of typing "Sunnydale." He started programming with an Apple //e in 4th grade, graduated from [UCLA](#) with a math degree in 1995, and immediately landed a job as a QA engineer at Symantec, working on the Norton AntiVirus team. He pretty much taught himself Windows and MFC programming, and in 1999 he designed and coded a new interface for Norton AntiVirus 2000.

Mike has been a a developer at [Napster](#) and at his own lil' startup, [Zabersoft](#), a development company he co-founded with offices in Los Angeles and Odense, Denmark. Mike is now a senior engineer at [VMware](#).

He also enjoys his hobbies of playing pinball, bike riding, photography, and the occasional 360 or MAME game (current favorite: *Space Invaders Extreme*). He would get his own snooker table too if they weren't so darn big! He is also sad that

he's forgotten the languages he's studied: French, Mandarin Chinese, and Japanese.

Mike was a [VC MVP](#) from 2005 to 2009.

Occupation: Software Developer (Senior)

Company: VMware

Location:  United States

Nishant Sivakumar



Member

Nish is a real nice guy living in Atlanta, who has been coding since 1990, when he was 13 years old. Originally from sunny Trivandrum in India, he recently moved to Atlanta from Toronto and is a little sad that he won't be able to play in snow anymore.

Nish has been a Microsoft Visual C++ MVP since October, 2002 - awfully nice of Microsoft, he thinks. He maintains an MVP tips and tricks web site - www.voidnish.com where you can find a consolidated list of his articles, writings and ideas on VC++, MFC, .NET and C++/CLI. Oh, and you might want to check out his blog on C++/CLI, MFC, .NET and a lot of other stuff - blog.voidnish.com


Nish loves reading Science Fiction, P G Wodehouse and Agatha Christie, and also fancies himself to be a decent writer of sorts. He has authored a romantic comedy [Summer Love and Some more Cricket](#) as well as a programming book - [Extending MFC applications with the .NET Framework](#).

Nish's latest book [C++/CLI in Action](#) published by Manning Publications is now available for purchase. You can read more about the book on his blog.

Despite his wife's attempts to get him into cooking, his best effort so far has been a badly done omelette. Some day, he hopes to be a good cook, and to cook a tasty dinner for his wife.

Location:  United States

Discussions and Feedback

 **174 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/string/cppstringguide2.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
 Last Updated: 12 Oct 2002
 Editor: [Nishant Sivakumar](#)

Copyright 2002 by Michael Dunn, Nishant Sivakumar
 Everything else Copyright © [CodeProject](#), 1999-2009
 Web15 | [Advertise on the Code Project](#)