Contents

Text and Strings in Visual C++

2/3/2021 • 2 minutes to read • Edit Online

An important aspect of developing applications for international markets is the adequate representation of local character sets. The ASCII character set defines characters in the range 0x00 to 0x7F. There are other character sets, primarily European, that define the characters within the range 0x00 to 0x7F identically to the ASCII character set and also define an extended character set from 0x80 to 0xFF. Thus, an 8-bit, single-byte-character set (SBCS) is sufficient to represent the ASCII character set, as well as the character sets for many European languages. However, some non-European character sets, such as Japanese Kanji, include many more characters than a single-byte coding scheme can represent, and therefore require multibyte-character set (MBCS) encoding.

In This Section

Unicode and MBCS
Discusses Visual C++ support for Unicode and MBCS programming.

Support for Unicode
Describes Unicode, a specification for supporting all character sets, including character sets that cannot be represented in a single byte.

Support for Multibyte Character Sets (MBCS)
Discusses MBCS, an alternative to Unicode for supporting character sets, like Japanese and Chinese, that cannot be represented in a single byte.

Generic-Text Mappings in tchar.h
Provides Microsoft-specific generic-text mappings for many data types, routines, and other objects.

How to: Convert Between Various String Types
Demonstrates how to convert various Visual C++ string types into other strings.

Related Sections

Internationalization
Discusses international support in the C run-time library.

International Samples
Provides links to samples demonstrating internationalization in Visual C++.

Language and Country/Region Strings
Provides the language and country/region strings in the C run-time library.

Unicode and MBCS

2/3/2021 • 2 minutes to read • Edit Online

The Microsoft Foundation Classes (MFC) library, the C run-time library for Visual C++, and the Visual C++ development environment are enabled to assist your international programming. They provide:

- Support for the Unicode standard on Windows. Unicode is the current standard and should be used whenever possible.

  Unicode is a 16-bit character encoding, providing enough encodings for all languages. All ASCII characters are included in Unicode as widened characters.

- Support for a form of multibyte character set (MBCS) called double-byte character set (DBCS) on all platforms.

  DBCS characters are composed of 1 or 2 bytes. Some ranges of bytes are set aside for use as lead bytes. A lead byte specifies that it and the following trail byte comprise a single 2-byte-wide character. You must keep track of which bytes are lead bytes. In a particular multibyte-character set, the lead bytes fall within a certain range, as do the trail bytes. When these ranges overlap, it might be necessary to evaluate the context to determine whether a given byte is functioning as a lead byte or a trail byte.

- Support for tools that simplify MBCS programming of applications written for international markets.

  When run on an MBCS-enabled version of the Windows operating system, the Visual C++ development system — including the integrated source code editor, debugger, and command-line tools — is completely MBCS-enabled. For more information, see MBCS Support in Visual C++.

NOTE

In this documentation, MBCS is used to describe all non-Unicode support for multibyte characters. In Visual C++, MBCS always means DBCS. Character sets wider than 2 bytes are not supported.

By definition, the ASCII character set is a subset of all multibyte-character sets. In many multibyte character sets, each character in the range 0x00 - 0x7F is identical to the character that has the same value in the ASCII character set. For example, in both ASCII and MBCS character strings, the 1-byte NULL character ('\0') has value 0x00 and indicates the terminating null character.

See also

Text and Strings
International Enabling

International Enabling

2/3/2021 • 2 minutes to read • Edit Online

Most traditional C and C++ code makes assumptions about character and string manipulation that do not work well for international applications. While both MFC and the run-time library support Unicode or MBCS, there is still work for you to do. To guide you, this section explains the meaning of "international enabling" in Visual C++:

- Both Unicode and MBCS are enabled by means of portable data types in MFC function parameter lists and return types. These types are conditionally defined in the appropriate ways, depending on whether your build defines the symbol `_UNICODE` or the symbol `_MBCS` (which means DBCS). Different variants of the MFC libraries are automatically linked with your application, depending on which of these two symbols your build defines.
- Class library code uses portable run-time functions and other means to ensure correct Unicode or MBCS behavior.
- You still must handle certain kinds of internationalization tasks in your code:
  - Use the same portable run-time functions that make MFC portable under either environment.
  - Make literal strings and characters portable under either environment, using the `_T` macro. For more information, see Generic-Text Mappings in tchar.h.
  - Take precautions when parsing strings under MBCS. These precautions are not needed under Unicode. For more information, see MBCS Programming Tips.
  - Take care if you mix ANSI (8-bit) and Unicode (16-bit) characters in your application. It is possible to use ANSI characters in some parts of your program and Unicode characters in others, but you cannot mix them in the same string.
  - Do not hard-code strings in your application. Instead, make them STRINGTABLE resources by adding them to the application's .rc file. Your application can then be localized without requiring source code changes or recompilation. For more information about STRINGTABLE resources, see String Editor.

NOTE

European and MBCS character sets have some characters, such as accented letters, with character codes greater than 0x80. Because most code uses signed characters, these characters greater than 0x80 are sign-extended when converted to `int`. This is a problem for array indexing because the sign-extended characters, being negative, indexes outside the array. Languages that use MBCS, such as Japanese, are also unique. Because a character might consist of 1 or 2 bytes, you should always manipulate both bytes at the same time.

See also

Unicode and MBCS
Internationalization Strategies

Internationalization Strategies

2/3/2021 • 2 minutes to read • Edit Online

Depending on your target operating systems and markets, you have several internationalization strategies:

- Your application uses Unicode.

  You use Unicode-specific functionality and all characters are 16 bits wide (although you can use ANSI characters in some parts of your program for special purposes). The C run-time library provides functions, macros, and data types for Unicode-only programming. MFC is fully Unicode-enabled.

- Your application uses MBCS and can be run on any Win32 platform.

  You use MBCS-specific functionality. Strings can contain single-byte characters, double-byte characters, or both. The C run-time library provides functions, macros, and data types for MBCS-only programming. MFC is fully MBCS-enabled.

- The source code for your application is written for complete portability — by recompiling with the symbol `_UNICODE` or the symbol `_MBCS` defined, you can produce versions that use either. For more information, see Generic-Text Mappings in tchar.h.

  You use fully portable C run-time functions, macros, and data types. MFC's flexibility supports any of these strategies.

The remainder of these topics focus on writing completely portable code that you can build as Unicode or as MBCS.

See also

Unicode and MBCS
Locales and Code Pages

Locales and Code Pages

2/3/2021 • 2 minutes to read • Edit Online

A locale ID reflects the local conventions and language for a particular geographical region. A given language might be spoken in more than one country/region; for example, Portuguese is spoken in Brazil as well as in Portugal. Conversely, a country/region might have more than one official language. For example, Canada has two languages: English and French. Thus, Canada has two distinct locales: Canadian-English and Canadian-French. Some locale-dependent categories include the formatting of dates and the display format for monetary values.

The language determines the text and data formatting conventions, while the country/region determines the local conventions. Every language has a unique mapping, represented by code pages, which includes characters other than those in the alphabet (such as punctuation marks and numbers). A code page is a character set and is related to the language. As such, a locale is a unique combination of language, country/region, and code page. The locale and code page setting can be changed at run time by calling the setlocale function.

Different languages might use different code pages. For example, the ANSI code page 1252 is used for English and most European languages, and the ANSI code page 932 is used for Japanese Kanji. Virtually all code pages share the ASCII character set for the lowest 128 characters (0x00 to 0x7F).

Any single-byte code page can be represented in a table (with 256 entries) as a mapping of byte values to characters (including numbers and punctuation marks), or glyphs. Any multibyte code page can also be represented as a very large table (with 64K entries) of double-byte values to characters. In practice, however, it is usually represented as a table for the first 256 (single-byte) characters and as ranges for the double-byte values.

For more information about code pages, see Code Pages.

The C run-time library has two types of internal code pages: locale and multibyte. You can change the current code page during program execution (see the documentation for the setlocale and _setmbcp functions). Also, the run-time library might obtain and use the value of the operating system code page, which is constant for the duration of the program's execution.

When the locale code page changes, the behavior of the locale-dependent set of functions changes to that dictated by the chosen code page. By default, all locale-dependent functions begin execution with a locale code page unique to the "C" locale. You can change the internal locale code page (as well as other locale-specific properties) by calling the `setlocale` function. A call to `setlocale` (LC_ALL, "") sets the locale to that indicated by the operating system user locale.

Similarly, when the multibyte code page changes, the behavior of the multibyte functions changes to that dictated by the chosen code page. By default, all multibyte functions begin execution with a multibyte code page corresponding to the operating system's default code page. You can change the internal multibyte code page by calling the `_setmbcp` function.

The C run-time function `setlocale` sets, changes, or queries some or all of the current program's locale information. The _wsetlocale routine is a wide-character version of `setlocale`; the arguments and return values of `_wsetlocale` are wide-character strings.

See also

Unicode and MBCS
Benefits of Character Set Portability

Benefits of Character Set Portability

2/3/2021 • 2 minutes to read • Edit Online

You can benefit from using MFC and C run-time portability features even if you do not currently intend to internationalize your application:

- Coding portably makes your code base flexible. You can later move it easily to Unicode or MBCS.
- Using Unicode makes your applications for Windows more efficient. Because Windows uses Unicode, non-Unicode strings passed to and from the operating system must be translated, which incurs overhead.

See also

Unicode and MBCS
Support for Unicode

Support for Unicode

2/3/2021 • 2 minutes to read • Edit Online

Unicode is a specification for supporting all character sets, including ones that can't be represented in a single byte. If you're programming for an international market, we recommend you use either Unicode or a multibyte character set (MBCS). Or, code your program so you can build it for either by changing a switch.

A wide character is a 2-byte multilingual character code. Tens of thousands of characters, comprising almost all characters used in modern computing worldwide, including technical symbols and special publishing characters, can be represented according to the Unicode specification as a single wide character encoded by using UTF-16. Characters that cannot be represented in just one wide character can be represented in a Unicode pair by using the Unicode surrogate pair feature. Because almost every character in common use is represented in UTF-16 in a single 16-bit wide character, using wide characters simplifies programming with international character sets. Wide characters encoded using UTF-16LE (for little-endian) are the native character format for Windows.

A wide-character string is represented as a `wchar_t[]` array and is pointed to by a `wchar_t*` pointer. Any ASCII character can be represented as a wide character by prefixing the letter L to the character. For example, L'\0' is the terminating wide (16-bit) NULL character. Similarly, any ASCII string literal can be represented as a wide-character string literal by prefixing the letter L to the ASCII literal (L"Hello").

Generally, wide characters take more space in memory than multibyte characters but are faster to process. In addition, only one locale can be represented at a time in a multibyte encoding, whereas all character sets in the world are represented simultaneously by the Unicode representation.

The MFC framework is Unicode-enabled throughout, and MFC accomplishes Unicode enabling by using portable macros, as shown in the following table.

Portable Data Types in MFC

NON-PORTABLE DATA TYPE	REPLACED BY THIS MACRO
`char`, `wchar_t`	`_TCHAR`
`char*`, `LPSTR` (Win32 data type), `LPWSTR`	`LPTSTR`
`const char*`, `LPCSTR` (Win32 data type), `LPCWSTR`	`LPCTSTR`

Class `CString` uses `_TCHAR` as its base and provides constructors and operators for easy conversions. Most string operations for Unicode can be written by using the same logic used for handling the Windows ANSI character set, except that the basic unit of operation is a 16-bit character instead of an 8-bit byte. Unlike working with multibyte character sets, you do not have to (and should not) treat a Unicode character as if it were two distinct bytes. You do, however, have to deal with the possibility of a single character represented by a surrogate pair of wide characters. In general, do not write code that assumes the length of a string is the same as the number of characters, whether narrow or wide, that it contains.

What do you want to do?

- Use MFC Unicode and Multibyte Character Set (MBCS) Support
- Enable Unicode in my program

- Enable both Unicode and MBCS in my program
- Use Unicode to create an internationalized program
- Learn the benefits of Unicode
- Use wmain so I can pass wide-character arguments to my program
- See a summary of Unicode programming
- Learn about generic-text mappings for byte-width portability

See also

Text and Strings
Support for Using wmain

Support for Using wmain

2/3/2021 • 2 minutes to read • Edit Online

Visual C++ supports defining a **wmain function** and passing wide-character arguments to your Unicode application. You declare formal parameters to **wmain**, using a format similar to `main`. You can then pass wide-character arguments and, optionally, a wide-character environment pointer to the program. The `argv` and `envp` parameters to **wmain** are of type `wchar_t*`. For example:

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

NOTE

MFC Unicode applications use `wWinMain` as the entry point. In this case, `CWinApp::m_lpCmdLine` is a Unicode string. Be sure to set `wWinMainCRTStartup` with the /ENTRY linker option.

If your program uses a **main** function, the multibyte-character environment is created by the run-time library at program startup. A wide-character copy of the environment is created only when needed (for example, by a call to the `_wgetenv` or `_wputenv` functions). On the first call to `_wputenv`, or on the first call to `_wgetenv` if an MBCS environment already exists, a corresponding wide-character string environment is created. The environment is then pointed to by the `_wenviron` global variable, which is a wide-character version of the `_environ` global variable. At this point, two copies of the environment (MBCS and Unicode) exist simultaneously and are maintained by the run-time system throughout the life of the program.

Similarly, if your program uses a **wmain** function, a wide-character environment is created at program startup and is pointed to by the `_wenviron` global variable. An MBCS (ASCII) environment is created on the first call to `_putenv` or `getenv` and is pointed to by the `_environ` global variable.

See also

Support for Unicode
Unicode Programming Summary
WinMain Function

Unicode Programming Summary

2/3/2021 • 2 minutes to read • Edit Online

To take advantage of the MFC and C run-time support for Unicode, you need to:

- Define `_UNICODE`.

  Define the symbol `_UNICODE` before you build your program.

- Specify entry point.

  On the **Advanced** page of the **Linker** folder in the project's Property Pages dialog box, set the **Entry Point** symbol to `wWinMainCRTStartup`.

- Use portable run-time functions and types.

  Use the proper C run-time functions for Unicode string handling. You can use the `wcs` family of functions, but you might prefer the fully portable (internationally enabled) `_TCHAR` macros. These macros are all prefixed with `_tcs`; they substitute, one for one, for the `str` family of functions. These functions are described in detail in the Internationalization section of the *Run-Time Library Reference*. For more information, see Generic-Text Mappings in tchar.h.

  Use `_TCHAR` and the related portable data types described in Support for Unicode.

- Handle literal strings properly.

  The Visual C++ compiler interprets a literal string coded as:

  ```
  L"this is a literal string"
  ```

  to mean a string of Unicode characters. You can use the same prefix for literal characters. Use the `_T` macro to code literal strings generically, so they compile as Unicode strings under Unicode or as ANSI strings (including MBCS) without Unicode. For example, instead of:

  ```
  pWnd->SetWindowText( "Hello" );
  ```

  use:

  ```
  pWnd->SetWindowText( _T("Hello") );
  ```

  With `_UNICODE` defined, `_T` translates the literal string to the L-prefixed form; otherwise, `_T` translates the string without the L prefix.

  TIP
  The `_T` macro is identical to the `_TEXT` macro.

- Be careful passing string lengths to functions.

  Some functions want the number of characters in a string; others want the number of bytes. For example, if `_UNICODE` is defined, the following call to a `CArchive` object will not work (`str` is a `CString`):

```
archive.Write( str, str.GetLength( ) );    // invalid
```

In a Unicode application, the length gives you the number of characters but not the correct number of bytes, because each character is 2 bytes wide. Instead, you must use:

```
archive.Write( str, str.GetLength( ) * sizeof( _TCHAR ) );    // valid
```

which specifies the correct number of bytes to write.

However, MFC member functions that are character-oriented, rather than byte-oriented, work without this extra coding:

```
pDC->TextOut( str, str.GetLength( ) );
```

`CDC::TextOut` takes a number of characters, not a number of bytes.

- Use fopen_s, _wfopen_s to open Unicode files.

To summarize, MFC and the run-time library provide the following support for Unicode programming:

- Except for database class member functions, all MFC functions are Unicode-enabled, including `CString`. `CString` also provides Unicode/ANSI conversion functions.
- The run-time library supplies Unicode versions of all string-handling functions. (The run-time library also supplies portable versions suitable for Unicode or for MBCS. These are the `_tcs` macros.)
- tchar.h supplies portable data types and the `_T` macro for translating literal strings and characters. For more information, see Generic-Text Mappings in tchar.h.
- The run-time library provides a wide-character version of `main`. Use `wmain` to make your application Unicode-aware.

See also

Support for Unicode

Support for Multibyte Character Sets (MBCSs)

2/3/2021 • 2 minutes to read • Edit Online

Multibyte character sets (MBCSs) are an older approach to the need to support character sets, like Japanese and Chinese, that cannot be represented in a single byte. If you are doing new development, you should use Unicode for all text strings except perhaps system strings that are not seen by end users. MBCS is a legacy technology and is not recommended for new development.

The most common MBCS implementation is double-byte character sets (DBCSs). Visual C++ in general, and MFC in particular, is fully enabled for DBCS.

For samples, see the MFC source code files.

For platforms used in markets whose languages use large character sets, the best alternative to Unicode is MBCS. MFC supports MBCS by using internationalizable data types and C run-time functions. You should do the same in your code.

Under MBCS, characters are encoded in either 1 or 2 bytes. In 2-byte characters, the first, or lead byte, signals that both it and the following byte are to be interpreted as one character. The first byte comes from a range of codes reserved for use as lead bytes. Which ranges of bytes can be lead bytes depends on the code page in use. For example, Japanese code page 932 uses the range 0x81 through 0x9F as lead bytes, but Korean code page 949 uses a different range.

Consider all the following in your MBCS programming.

MBCS characters in the environment MBCS characters can appear in strings such as file and directory names.

Editing operations

Editing operations in MBCS applications should operate on characters, not bytes. The caret should not split a character, the **Right Arrow** key should move right one character, and so on. **Delete** should delete a character; **Undo** should reinsert it.

String handling

In an application that uses MBCS, string handling poses special problems. Characters of both widths are mixed in a single string; therefore, you must remember to check for lead bytes.

Run-time library support

The C run-time library and MFC support single-byte, MBCS, and Unicode programming. Single-byte strings are processed with the `str` family of run-time functions, MBCS strings are processed with corresponding `_mbs` functions, and Unicode strings are processed with corresponding `wcs` functions. MFC class member function implementations use portable run-time functions that map, under the right circumstances, to the normal `str` family of functions, the MBCS functions, or the Unicode functions, as described in "MBCS/Unicode portability."

MBCS/Unicode portability

Using the tchar.h header file, you can build single-byte, MBCS, and Unicode applications from the same sources. Tchar.h defines macros prefixed with *_tcs* , which map to `str`, `_mbs`, or `wcs` functions, as appropriate. To build MBCS, define the symbol `_MBCS`. To build Unicode, define the symbol `_UNICODE`. By default, `_UNICODE` is defined for MFC applications. For more information, see Generic-Text Mappings in tchar.h.

NOTE

Behavior is undefined if you define both `_UNICODE` and `_MBCS`.

The Mbctype.h and Mbstring.h header files define MBCS-specific functions and macros, which you might need in some cases. For example, `_ismbblead` tells you whether a specific byte in a string is a lead byte.

For international portability, code your program with Unicode or multibyte character sets (MBCSs).

What do you want to do?

- Enable MBCS in my program
- Enable both Unicode and MBCS in my program
- Use MBCS to create an internationalized program
- See a summary of MBCS programming
- Learn about generic-text mappings for byte-width portability

See also

Text and Strings
MBCS Support in Visual C++

MBCS Support in Visual C++

2/3/2021 • 2 minutes to read • Edit Online

When run on an MBCS-enabled version of Windows, the Visual C++ development system (including the integrated source code editor, debugger, and command line tools) is MBCS-enabled, with the exception of the memory window.

The memory window does not interpret bytes of data as MBCS characters, even though it can interpret them as ANSI or Unicode characters. ANSI characters are always 1 byte in size and Unicode characters are 2 bytes in size. With MBCS, characters can be 1 or 2 bytes in size and their interpretation depends on which code page is in use. Because of this, it is difficult for the memory window to reliably display MBCS characters. The memory window cannot know which byte is the start of a character. The developer can view the byte values in the memory window and look up the value in tables to determine the character representation. This is possible because the developer knows the starting address of a string based on the source code.

Visual C++ accepts double-byte characters wherever it is appropriate to do so. This includes path names and file names in dialog boxes and text entries in the Visual C++ resource editor (for example, static text in the dialog editor and static text entries in the icon editor). In addition, the preprocessor recognizes some double-byte directives — for example, file names in `#include` statements, and as arguments to the `code_seg` and `data_seg` pragmas. In the source code editor, double-byte characters in comments and string literals are accepted, although not in C/C++ language elements (such as variable names).

Support for the Input Method Editor (IME)

Applications written for East Asian markets that use MBCS (for example, Japan) normally support the Windows IME for entering both single- and double-byte characters. The Visual C++ development environment contains full support for the IME.

Japanese keyboards do not directly support Kanji characters. The IME converts a phonetic string, entered in one of the other Japanese alphabets (Romaji, Katakana, or Hiragana) into its possible Kanji representations. If there is ambiguity, you can select from several alternatives. When you have selected the intended Kanji character, the IME passes two `WM_CHAR` messages to the controlling application.

The IME, activated by the ALT+` key combination, appears as a set of buttons (an indicator) and a conversion window. The application positions the window at the text insertion point. The application must handle `WM_MOVE` and `WM_SIZE` messages by repositioning the conversion window to conform to the new location or size of the target window.

If you want users of your application to have the ability to enter Kanji characters, the application must handle Windows IME messages. For more information about IME programming, see Input Method Manager.

Visual C++ Debugger

The Visual C++ debugger provides the ability to set breakpoints on IME messages. In addition, the Memory window can display double-byte characters.

Command-Line Tools

The Visual C++ command-line tools, including the compiler, NMAKE, and the resource compiler (RC.EXE), are MBCS-enabled. You can use the resource compiler's /c option to change the default code page when compiling your application's resources.

To change the default locale at source code compile time, use #pragma setlocale.

Graphical Tools

The Visual C++ Windows-based tools, such as Spy++ and the resource editing tools, fully support IME strings.

See also

Support for Multibyte Character Sets (MBCSs)
MBCS Programming Tips

MBCS Programming Tips

2/3/2021 • 2 minutes to read • Edit Online

In new development, you should use Unicode character encoding for all strings that end users might possibly see. MBCS is a legacy technology that has been superseded by Unicode. This section supplies tips for developers who must maintain existing programs that use MBCS and where it is not practical to convert to Unicode. The advice applies to MFC applications and applications written without MFC. Topics include:

- General MBCS Programming Advice
- Incrementing and Decrementing Pointers
- Byte Indices
- Last Character in a String
- Character Assignment
- Character Comparison
- Buffer Overflow

See also

Support for Multibyte Character Sets (MBCSs)

General MBCS Programming Advice

2/3/2021 • 2 minutes to read • Edit Online

Use the following tips:

- For flexibility, use run-time macros such as `_tcschr` and `_tcscpy` when possible. For more information, see Generic-Text Mappings in tchar.h.
- Use the C run-time `_getmbcp` function to get information about the current code page.
- Do not reuse string resources. Depending on the target language, a given string might have a different meaning when translated. For example, "File" on the application's main menu might translate differently from the string "File" in a dialog box. If you need to use more than one string with the same name, use different string IDs for each.
- You might want to find out whether your application is running on an MBCS-enabled operating system. To do so, set a flag at program startup; do not rely on API calls.
- When designing dialog boxes, allow approximately 30% extra space at the end of static text controls for MBCS translation.
- Be careful when selecting fonts for your application, because some fonts are not available on all systems.
- When selecting the font for dialog boxes, use MS Shell Dlg instead of MS Sans Serif or Helvetica. MS Shell Dlg is replaced with the correct font by the system before creating the dialog box. Using MS Shell Dlg ensures that any changes in the operating system to deal with this font will automatically be available. (MFC replaces MS Shell Dlg with the DEFAULT_GUI_FONT or the System font on Windows 95, Windows 98, and Windows NT 4 because those systems do not handle MS Shell Dlg correctly.)
- When designing your application, decide which strings can be localized. If in doubt, assume that any given string will be localized. As such, do not mix strings that can be localized with those that cannot.

See also

MBCS Programming Tips
Incrementing and Decrementing Pointers

Incrementing and Decrementing Pointers

2/3/2021 • 2 minutes to read • Edit Online

Use the following tips:

- Point to lead bytes, not trail bytes. It is usually unsafe to have a pointer to a trail byte. It is usually safer to scan a string forward rather than in reverse.
- There are pointer increment/decrement functions and macros available that move over a whole character:

  ```
  sz1++;
  ```

  becomes:

  ```
  sz1 = _mbsinc( sz1 );
  ```

  The `_mbsinc` and `_mbsdec` functions correctly increment and decrement in `character` units, regardless of the character size.
- For decrements, you need a pointer to the head of the string, as in the following:

  ```
  sz2--;
  ```

  becomes:

  ```
  sz2 = _mbsdec( sz2Head, sz2 );
  ```

  Alternatively, your head pointer could be to a valid character in the string, such that:

  ```
  sz2Head < sz2
  ```

  You must have a pointer to a known valid lead byte.
- You might want to maintain a pointer to the previous character for faster calls to `_mbsdec`.

See also

MBCS Programming Tips
Byte Indices

Byte Indices

2/3/2021 • 2 minutes to read • Edit Online

Use the following tips:

- Working with a bytewise index into a string presents problems similar to those posed by pointer manipulation. Consider this example, which scans a string for a backslash character:

  ```
  while ( rgch[ i ] != '\\' )
      i++;
  ```

  This might index a trail byte, not a lead byte, and thus it might not point to a `character`.

- Use the _mbclen function to solve the preceding problem:

  ```
  while ( rgch[ i ] != '\\' )
      i += _mbclen ( rgch + i );
  ```

  This correctly indexes to a lead byte, hence to a `character`. The `_mbclen` function determines the size of a character (1 or 2 bytes).

See also

MBCS Programming Tips
Last Character in a String

Last Character in a String

2/3/2021 • 2 minutes to read • Edit Online

Use the following tips:

- Trail byte ranges overlap the ASCII character set in many cases. You can safely use bytewise scans for any control characters (less than 32).

- Consider the following line of code, which might be checking to see if the last character in a string is a backslash character:

  ```
  if ( sz[ strlen( sz ) - 1 ] == '\\' )    // Is last character a '\'?
      // . . .
  ```

  Because `strlen` is not MBCS-aware, it returns the number of bytes, not the number of characters, in a multibyte string. Also, note that in some code pages (932, for example), '\' (0x5c) is a valid trail byte (`sz` is a C string).

  One possible solution is to rewrite the code this way:

  ```
  char *pLast;
  pLast = _mbsrchr( sz, '\\' );    // find last occurrence of '\' in sz
  if ( pLast && ( *_mbsinc( pLast ) == '\0' ) )
      // . . .
  ```

  This code uses the MBCS functions `_mbsrchr` and `_mbsinc`. Because these functions are MBCS-aware, they can distinguish between a '\' character and a trail byte '\'. The code performs some action if the last character in the string is a null ('\0').

See also

MBCS Programming Tips
Character Assignment

Character Assignment

2/3/2021 • 2 minutes to read • Edit Online

Consider the following example, in which the `while` loop scans a string, copying all characters except 'X' into another string:

```
while( *sz2 )
{
    if( *sz2 != 'X' )
        *sz1++ = *sz2++;
    else
        sz2++;
}
```

The code copies the byte at `sz2` to the location pointed to by `sz1`, then increments `sz1` to receive the next byte. But if the next character in `sz2` is a double-byte character, the assignment to `sz1` copies only the first byte. The following code uses a portable function to copy the character safely and another to increment `sz1` and `sz2` correctly:

```
while( *sz2 )
{
    if( *sz2 != 'X' )
    {
        _mbscpy_s( sz1, 1, sz2 );
        sz1 = _mbsinc( sz1 );
        sz2 = _mbsinc( sz2 );
    }
    else
        sz2 = _mbsinc( sz2 );
}
```

See also

MBCS Programming Tips
Character Comparison

Character Comparison

2/3/2021 • 2 minutes to read • Edit Online

Use the following tips:

- Comparing a known lead byte with an ASCII character works correctly:

  ```
  if( *sz1 == 'A' )
  ```

- Comparing two unknown characters requires the use of one of the macros defined in Mbstring.h:

  ```
  if( !_mbccmp( sz1, sz2) )
  ```

  This ensures that both bytes of a double-byte character are compared for equality.

See also

MBCS Programming Tips
Buffer Overflow

Buffer Overflow

2/3/2021 • 2 minutes to read • Edit Online

Varying character sizes can cause problems when you put characters into a buffer. Consider the following code, which copies characters from a string, `sz`, into a buffer, `rgch`:

```
cb = 0;
while( cb < sizeof( rgch ) )
    rgch[ cb++ ] = *sz++;
```

The question is: Was the last byte copied a lead byte? The following does not solve the problem because it can potentially overflow the buffer:

```
cb = 0;
while( cb < sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

The `_mbccpy` call attempts to do the correct thing — copy the full character, whether it is 1 or 2 bytes. But it does not take into account that the last character copied might not fit the buffer if the character is 2 bytes wide. The correct solution is:

```
cb = 0;
while( (cb + _mbclen( sz )) <= sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

This code tests for possible buffer overflow in the loop test, using `_mbclen` to test the size of the current character pointed to by `sz`. By making a call to the `_mbsnbcpy` function, you can replace the code in the `while` loop with a single line of code. For example:

```
_mbsnbcpy( rgch, sz, sizeof( rgch ) );
```

See also

MBCS Programming Tips

Support for ANSI

2/3/2021 • 2 minutes to read • Edit Online

Most MFC classes and methods support the ANSI character set, although the MFC framework as a whole is gradually evolving toward supporting only the Unicode character set. Because of the ongoing enhancements in Windows Vista and Windows Common Controls version 6.1, support for several ANSI classes and methods is deprecated. For more information, see Deprecated ANSI APIs and Support for Unicode.

See also

Support for Unicode
Deprecated ANSI APIs
Shell And Common Controls Versions

Generic-Text Mappings in tchar.h

2/3/2021 • 3 minutes to read • Edit Online

To simplify the transporting of code for international use, the Microsoft run-time library provides Microsoft-specific generic-text mappings for many data types, routines, and other objects. You can use these mappings, which are defined in tchar.h, to write generic code that can be compiled for single-byte, multibyte, or Unicode character sets, depending on a manifest constant that you define by using a `#define` statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

By using the tchar.h, you can build single-byte, Multibyte Character Set (MBCS), and Unicode applications from the same sources. tchar.h defines macros (which have the prefix `_tcs`) that, with the correct preprocessor definitions, map to `str`, `_mbs`, or `wcs` functions, as appropriate. To build MBCS, define the symbol `_MBCS`. To build Unicode, define the symbol `_UNICODE`. To build a single-byte application, define neither (the default). By default, `_UNICODE` is defined for MFC applications.

The `_TCHAR` data type is defined conditionally in tchar.h. If the symbol `_UNICODE` is defined for your build, `_TCHAR` is defined as `wchar_t`; otherwise, for single-byte and MBCS builds, it is defined as `char`. (`wchar_t`, the basic Unicode wide-character data type, is the 16-bit counterpart to an 8-bit `signed char`.) For international applications, use the `_tcs` family of functions, which operate in `_TCHAR` units, not bytes. For example, `_tcsncpy` copies `n` `_TCHARs`, not `n` bytes.

Because some Single Byte Character Set (SBCS) string-handling functions take (signed) `char*` parameters, a type mismatch compiler warning results when `_MBCS` is defined. There are three ways to avoid this warning:

1. Use the type-safe inline function thunks in tchar.h. This is the default behavior.
2. Use the direct macros in tchar.h by defining `_MB_MAP_DIRECT` on the command line. If you do this, you must manually match types. This is the fastest method, but is not type-safe.
3. Use the type-safe statically linked library function thunks in tchar.h. To do so, define the constant `_NO_INLINING` on the command line. This is the slowest method, but the most type-safe.

Preprocessor Directives for Generic-Text Mappings

# DEFINE	COMPILED VERSION	EXAMPLE
`_UNICODE`	Unicode (wide-character)	`_tcsrev` maps to `_wcsrev`
`_MBCS`	Multibyte-character	`_tcsrev` maps to `_mbsrev`
None (the default has neither `_UNICODE` nor `_MBCS` defined)	SBCS (ASCII)	`_tcsrev` maps to `strrev`

For example, the generic-text function `_tcsrev`, which is defined in tchar.h, maps to `_mbsrev` if you defined `_MBCS` in your program, or to `_wcsrev` if you defined `_UNICODE`. Otherwise, `_tcsrev` maps to `strrev`. Other data type mappings are provided in tchar.h for programming convenience, but `_TCHAR` is the most useful.

Generic-Text Data Type Mappings

GENERIC-TEXT DATA TYPE NAME	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_TCHAR`	`char`	`char`	`wchar_t`
`_TINT`	`int`	`unsigned int`	`wint_t`
`_TSCHAR`	`signed char`	`signed char`	`wchar_t`
`_TUCHAR`	`unsigned char`	`unsigned char`	`wchar_t`
`_TXCHAR`	`char`	`unsigned char`	`wchar_t`
`_T` or `_TEXT`	No effect (removed by preprocessor)	No effect (removed by preprocessor)	`L` (converts the following character or string to its Unicode counterpart)

For a list of generic-text mappings of routines, variables, and other objects, see Generic-Text Mappings in the Run-Time Library Reference.

NOTE

Do not use the `str` family of functions with Unicode strings, which are likely to contain embedded null bytes. Similarly, do not use the `wcs` family of functions with MBCS (or SBCS) strings.

The following code fragments illustrate the use of `_TCHAR` and `_tcsrev` for mapping to the MBCS, Unicode, and SBCS models.

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

If `_MBCS` has been defined, the preprocessor maps this fragment to this code:

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

If `_UNICODE` has been defined, the preprocessor maps this fragment to this code:

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

If neither `_MBCS` nor `_UNICODE` have been defined, the preprocessor maps the fragment to single-byte ASCII code, as follows:

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

Therefore, you can write, maintain, and compile a single-source code file to run with routines that are specific to any of the three kinds of character sets.

See also

Text and Strings

Using TCHAR.H Data Types with _MBCS Code

Using TCHAR.H Data Types with _MBCS Code

2/3/2021 • 2 minutes to read • Edit Online

When the manifest constant `_MBCS` is defined, a given generic-text routine maps to one of the following kinds of routines:

- An SBCS routine that handles multibyte bytes, characters, and strings appropriately. In this case, the string arguments are expected to be of type `char*`. For example, `_tprintf` maps to `printf`; the string arguments to `printf` are of type `char*`. If you use the `_TCHAR` generic-text data type for your string types, the formal and actual parameter types for `printf` match because `_TCHAR*` maps to `char*`.
- An MBCS-specific routine. In this case, the string arguments are expected to be of type `unsigned char*`. For example, `_tcsrev` maps to `_mbsrev`, which expects and returns a string of type `unsigned char*`. If you use the `_TCHAR` generic-text data type for your string types, there is a potential type conflict because `_TCHAR` maps to type `char`.

Following are three solutions for preventing this type conflict (and the C compiler warnings or C++ compiler errors that would result):

- Use the default behavior. tchar.h provides generic-text routine prototypes for routines in the run-time libraries, as in the following example.

  ```
  char * _tcsrev(char *);
  ```

  In the default case, the prototype for `_tcsrev` maps to `_mbsrev` through a thunk in Libc.lib. This changes the types of the `_mbsrev` incoming parameters and outgoing return value from `_TCHAR*` (that is, `char *`) to `unsigned char *`. This method ensures type matching when you are using `_TCHAR`, but it is relatively slow due to the function call overhead.

- Use function inlining by incorporating the following preprocessor statement in your code.

  ```
  #define _USE_INLINING
  ```

  This method causes an inline function thunk, provided in tchar.h, to map the generic-text routine directly to the appropriate MBCS routine. The following code excerpt from tchar.h provides an example of how this is done.

  ```
  __inline char *_tcsrev(char *_s1)
  {return (char *)_mbsrev((unsigned char *)_s1);}
  ```

  If you can use inlining, this is the best solution, because it guarantees type matching and has no additional time cost.

- Use direct mapping by incorporating the following preprocessor statement in your code.

  ```
  #define _MB_MAP_DIRECT
  ```

  This approach provides a fast alternative if you do not want to use the default behavior or cannot use inlining. It causes the generic-text routine to be mapped by a macro directly to the MBCS version of the routine, as in the following example from tchar.h.

```
#define _tcschr _mbschr
```

When you take this approach, you must be careful to ensure use of appropriate data types for string arguments and string return values. You can use type casting to ensure proper type matching or you can use the `_TXCHAR` generic-text data type. `_TXCHAR` maps to type `char` in SBCS code but maps to type `unsigned char` in MBCS code. For more information about generic-text macros, see Generic-Text Mappings in the *Run-Time Library Reference*.

See also

Generic-Text Mappings in tchar.h

How to: Convert Between Various String Types

2/3/2021 • 17 minutes to read • Edit Online

This topic demonstrates how to convert various Visual C++ string types into other strings. The strings types that are covered include `char *`, `wchar_t*`, _bstr_t, CComBSTR, CString, basic_string, and System.String. In all cases, a copy of the string is made when converted to the new type. Any changes made to the new string will not affect the original string, and vice versa.

Example: Convert from char *

Description

This example demonstrates how to convert from a `char *` to the other string types listed above. A `char *` string (also known as a C style string) uses a null character to indicate the end of the string. C style strings usually require one byte per character, but can also use two bytes. In the examples below, `char *` strings are sometimes referred to as multibyte character strings because of the string data that results from converting from Unicode strings. Single byte and multibyte character (`MBCS`) functions can operate on `char *` strings.

Code

```
// convert_from_char.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create and display a C style string, and then use it
    // to create different kinds of strings.
    char *orig = "Hello, World!";
    cout << orig << " (char *)" << endl;

    // newsize describes the length of the
    // wchar_t string called wcstring in terms of the number
    // of wide characters, not the number of bytes.
    size_t newsize = strlen(orig) + 1;

    // The following creates a buffer large enough to contain
    // the exact number of characters in the original string
    // in the new format. If you want to add more characters
    // to the end of the string, increase the value of newsize
    // to increase the size of the buffer.
    wchar_t * wcstring = new wchar_t[newsize];

    // Convert char* string to a wchar_t* string.
    size_t convertedChars = 0;
    mbstowcs_s(&convertedChars, wcstring, newsize, orig, _TRUNCATE);
    // Display the result and indicate the type of string that it is.
    wcout << wcstring << _T(" (wchar_t *)") << endl;
    // Convert the C style string to a _bstr_t string.
    _bstr_t bstrt(orig);
    // Append the type of string to the new string
    // and then display the result.
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert the C style string to a CComBSTR string.
    CComBSTR ccombstr(orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert the C style string to a CStringA and display it.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert the C style string to a CStringW and display it.
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a CStringW correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the C style string to a basic_string and display it.
    string basicstring(orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert the C style string to a System::String and display it.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Example: Convert from wchar_t *

Description

This example demonstrates how to convert from a `wchar_t *` to the other string types listed above. Several string types, including `wchar_t *`, implement wide character formats. To convert a string between a multibyte and a wide character format, you can use a single function call like `mbstowcs_s` or a constructor invocation for a class like `CStringA`.

Code

```
// convert_from_wchar_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
```

```
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create a string of wide characters, display it, and then
    // use this string to create other types of strings.
    wchar_t *orig = _T("Hello, World!");
    wcout << orig << _T(" (wchar_t *)") << endl;

    // Convert the wchar_t string to a char* string. Record
    // the length of the original string and add 1 to it to
    // account for the terminating null character.
    size_t origsize = wcslen(orig) + 1;
    size_t convertedChars = 0;

    // Use a multibyte string to append the type of string
    // to the new string before displaying the result.
    char strConcat[] = " (char *)";
    size_t strConcatsize = (strlen( strConcat ) + 1)*2;

    // Allocate two bytes in the multibyte output string for every wide
    // character in the input string (including a wide character
    // null). Because a multibyte character can be one or two bytes,
    // you should allot two bytes for each character. Having extra
    // space for the new string is not an error, but having
    // insufficient space is a potential security problem.
    const size_t newsize = origsize*2;
    // The new string will contain a converted copy of the original
    // string plus the type of string appended to it.
    char *nstring = new char[newsize+strConcatsize];

    // Put a copy of the converted string into nstring
    wcstombs_s(&convertedChars, nstring, newsize, orig, _TRUNCATE);
    // append the type of string to the new string.
    _mbscat_s((unsigned char*)nstring, newsize+strConcatsize, (unsigned char*)strConcat);
    // Display the result.
    cout << nstring << endl;

    // Convert a wchar_t to a _bstr_t string and display it.
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert the wchar_t string to a BSTR wide character string
    // by using the ATL CComBSTR wrapper class for BSTR strings.
    // Then display the result.

    CComBSTR ccombstr(orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // CW2A converts the string in ccombstr to a multibyte
        // string in printstr, used here for display output.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
        // The following line of code is an easier way to
        // display wide character strings:
        wcout << (LPCTSTR) ccombstr << endl;
    }

    // Convert a wide wchar_t string to a multibyte CStringA,
    // append the type of string to it, and display the result.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a wide character wchar_t string to a wide
    // character CStringW string and append the type of string to it
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a CStringW correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the wide character wchar_t string to a
    // basic_string, append the type of string to it, and
    // display the result.
    wstring basicstring(orig);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    // Convert a wide character wchar_t string to a
    // System::String string, append the type of string to it,
    // and display the result.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (wchar_t *)
Hello, World! (char *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Example: Convert from _bstr_t

Description

This example demonstrates how to convert from a `_bstr_t` to the other string types listed above. The `_bstr_t` object is a way to encapsulate wide character `BSTR` strings. A BSTR string has a length value and does not use a null character to terminate the string, but the string type you convert to may require a terminating null.

Code

```
// convert_from_bstr_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create a _bstr_t string, display the result, and indicate the
    // type of string that it is.
    _bstr_t orig("Hello, World!");
    wcout << orig << " (_bstr_t)" << endl;

    // Convert the wide character _bstr_t string to a C style
    // string. To be safe, allocate two bytes for each character
    // in the char* string, including the terminating null.
    const size_t newsize = (orig.length()+1)*2;
    char *nstring = new char[newsize];

    // Uses the _bstr_t operator (char *) to obtain a null
    // terminated string from the _bstr_t object for
    // nstring.
    strcpy_s(nstring, newsize, (char *)orig);
    strcat_s(nstring, newsize, " (char *)");
    cout << nstring << endl;

    // Prepare the type of string to append to the result.
    wchar_t strConcat[] = _T(" (wchar_t *)");
    size_t strConcatLen = wcslen(strConcat) + 1;

    // Convert a _bstr_t to a wchar_t* string.
    const size_t widesize = orig.length()+ strConcatLen;
    wchar_t *wcstring = new wchar_t[newsize];
    wcscpy_s(wcstring, widesize, (wchar_t *)orig);
    wcscat_s(wcstring, widesize, strConcat);
    wcout << wcstring << endl;

    // Convert a _bstr_t string to a CComBSTR string.
    CComBSTR ccombstr((char *)orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a _bstr_t to a CStringA string.
    CStringA cstringa(orig.GetBSTR());
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a _bstr_t to a CStringW string.
    CStringW cstring(orig.GetBSTR());
    cstring += " (CStringW)";
    // To display a cstring correctly, use wcout and
    // "cast" the cstring to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the _bstr_t to a basic_string.
    string basicstring((char *)orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert the _bstr_t to a System::String.
    String ^systemstring = gcnew String((char *)orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (_bstr_t)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Example: Convert from CComBSTR

Description

This example demonstrates how to convert from a `CComBSTR` to the other string types listed above. Like _bstr_t, a `CComBSTR` object is a way to encapsulate wide character BSTR strings. A BSTR string has a length value and does not use a null character to terminate the string, but the string type you convert to may require a terminating null.

Code

```
// convert_from_ccombstr.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

int main()
{
    // Create and initialize a BSTR string by using a CComBSTR object.
    CComBSTR orig("Hello, World!");
    // Convert the BSTR into a multibyte string, display the result,
    // and indicate the type of string that it is.
    CW2A printstr(orig);
    cout << printstr << " (CComBSTR)" << endl;

    // Convert a wide character CComBSTR string to a
    // regular multibyte char* string. Allocate enough space
    // in the new string for the largest possible result,
    // including space for a terminating null.
    const size_t newsize = (orig.Length()+1)*2;
    char *nstring = new char[newsize];

    // Create a string conversion object, copy the result to
    // the new char* string, and display the result.
    CW2A tmpstr1(orig);
    strcpy_s(nstring, newsize, tmpstr1);
    cout << nstring << " (char *)" << endl;

    // Prepare the type of string to append to the result.
    wchar_t strConcat[] = _T(" (wchar_t *)");
    size_t strConcatLen = wcslen(strConcat) + 1;

    // Convert a wide character CComBSTR string to a wchar_t*.
    // The code first determines the length of the converted string
    // plus the length of the appended type of string, then
    // prepares the final wchar_t string for display.
    const size_t widesize = orig.Length()+ strConcatLen;
    wchar_t *wcstring = new wchar_t[widesize];
    wcscpy_s(wcstring, widesize, orig);
    wcscat_s(wcstring, widesize, strConcat);

    // Display the result. Unlike CStringW, a wchar_t does not need
    // a cast to (LPCTSTR) with wcout.
    wcout << wcstring << endl;

    // Convert a wide character CComBSTR to a wide character _bstr_t,
    // append the type of string to it, and display the result.
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert a wide character CComBSTR to a multibyte CStringA,
    // append the type of string to it, and display the result.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a wide character CComBSTR to a wide character CStringW.
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a cstring correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert a wide character CComBSTR to a wide character
    // basic_string.
    wstring basicstring(orig);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    // Convert a wide character CComBSTR to a System::String.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (CComBSTR)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Example: Convert from CString

Description

This example demonstrates how to convert from a `CString` to the other string types listed above. `CString` is based on the TCHAR data type, which in turn depends on whether the symbol `_UNICODE` is defined. If `_UNICODE` is not defined, `TCHAR` is defined to be char and `CString` contains a multibyte character string; if `_UNICODE` is defined, `TCHAR` is defined to be `wchar_t` and `CString` contains a wide character string.

`CStringA` is the multibyte string always version of `CString`, `CStringW` is the wide character string only version. Neither `CStringA` nor `CStringW` use `_UNICODE` to determine how they should compile. `CStringA` and `CStringW` are used in this example to clarify minor differences in buffer size allocation and output handling.

Code

```
// convert_from_cstring.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Set up a multibyte CStringA string.
    CStringA origa("Hello, World!");
    cout << origa << " (CStringA)" << endl;

    // Set up a wide character CStringW string.
    CStringW origw("Hello, World!");
    wcout << (LPCTSTR)origw << _T(" (CStringW)") << endl;

    // Convert to a char* string from CStringA string
    // and display the result.
    const size_t newsizea = (origa.GetLength() + 1);
    char *nstringa = new char[newsizea];
    strcpy_s(nstringa, newsizea, origa);
    cout << nstringa << " (char *)" << endl;

    // Convert to a char* string from a wide character
    // CStringW string. To be safe, we allocate two bytes for each
    // character in the original string, including the terminating
    // null.
    const size_t newsizew = (origw.GetLength() + 1)*2;
    char *nstringw = new char[newsizew];
    size_t convertedCharsw = 0;
    wcstombs_s(&convertedCharsw, nstringw, newsizew, origw, _TRUNCATE );
    cout << nstringw << " (char *)" << endl;

    // Convert to a wchar_t* from CStringA
    size_t convertedCharsa = 0;
    wchar_t *wcstring = new wchar_t[newsizea];
    mbstowcs_s(&convertedCharsa, wcstring, newsizea, origa, _TRUNCATE);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert to a wide character wchar_t* string from
    // a wide character CStringW string.
    wchar_t *n2stringw = new wchar_t[newsizew];
    wcscpy_s( n2stringw, newsizew, origw );
    wcout << n2stringw << _T(" (wchar_t *)") << endl;

    // Convert to a wide character _bstr_t string from
    // a multibyte CStringA string.
    _bstr_t bstrt(origa);
    bstrt += _T(" (_bstr_t)");
    wcout << bstrt << endl;

    // Convert to a wide character _bstr_t string from
    // a wide character CStringW string.
    bstr_t bstrtw(origw);
    bstrtw += " (_bstr_t)";
    wcout << bstrtw << endl;
    // Convert to a wide character CComBSTR string from
    // a multibyte character CStringA string.
    CComBSTR ccombstr(origa);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Convert the wide character string to multibyte
        // for printing.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a wide character CComBSTR string from
    // a wide character CStringW string.
    CComBSTR ccombstrw(origw);

    // Append the type of string to it, and display the result.
    if (ccombstrw.Append(_T(" (CComBSTR)")) == S_OK)
    {
        CW2A printstrw(ccombstrw);
        wcout << printstrw << endl;
    }

    // Convert a multibyte character CStringA to a
    // multibyte version of a basic_string string.
    string basicstring(origa);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert a wide character CStringW to a
    // wide character version of a basic_string
    // string.
    wstring basicstringw(origw);
    basicstringw += _T(" (basic_string)");
    wcout << basicstringw << endl;

    // Convert a multibyte character CStringA to a
    // System::String.
    String ^systemstring = gcnew String(origa);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;

    // Convert a wide character CStringW to a
    // System::String.
    String ^systemstringw = gcnew String(origw);
    systemstringw += " (System::String)";
    Console::WriteLine("{0}", systemstringw);
    delete systemstringw;
}
```

```
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (char *)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CComBSTR)
Hello, World! (basic_string)
Hello, World! (System::String)
```

Example: Convert from basic_string

Description

This example demonstrates how to convert from a `basic_string` to the other string types listed above.

Code

```
// convert_from_basic_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Set up a basic_string string.
    string orig("Hello, World!");
    cout << orig << " (basic_string)" << endl;

    // Convert a wide character basic_string string to a multibyte char*
    // string. To be safe, we allocate two bytes for each character
    // in the original string, including the terminating null.
    const size_t newsize = (strlen(orig.c_str()) + 1)*2;
    char *nstring = new char[newsize];
    strcpy_s(nstring, newsize, orig.c_str());
    cout << nstring << " (char *)" << endl;

    // Convert a basic_string string to a wide character
    // wchar_t* string. You must first convert to a char*
    // for this to work.
    const size_t newsizew = strlen(orig.c_str()) + 1;
    size_t convertedChars = 0;
    wchar_t *wcstring = new wchar_t[newsizew];
    mbstowcs_s(&convertedChars, wcstring, newsizew, orig.c_str(), _TRUNCATE);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert a basic_string string to a wide character
    // _bstr_t string.
    _bstr_t bstrt(orig.c_str());
    bstrt += _T(" (_bstr_t)");
    wcout << bstrt << endl;

    // Convert a basic_string string to a wide character
    // CComBSTR string.
    CComBSTR ccombstr(orig.c_str());
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Make a multibyte version of the CComBSTR string
        // and display the result.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a basic_string string into a multibyte
    // CStringA string.
    CStringA cstring(orig.c_str());
    cstring += " (CStringA)";
    cout << cstring << endl;
    // Convert a basic_string string into a wide
    // character CStringW string.
    CStringW cstringw(orig.c_str());
    cstringw += _T(" (CStringW)");
    wcout << (LPCTSTR)cstringw << endl;

    // Convert a basic_string string to a System::String
    String ^systemstring = gcnew String(orig.c_str());
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (basic_string)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (System::String)
```

Example: Convert from System::String

Description

This example demonstrates how to convert from a wide character (Unicode) System::String to the other string types listed above.

Code

```
// convert_from_system_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

int main()
{
    // Set up a System::String and display the result.
    String ^orig = gcnew String("Hello, World!");
    Console::WriteLine("{0} (System::String)", orig);

    // Obtain a pointer to the System::String in order to
    // first lock memory into place, so that the
    // Garbage Collector (GC) cannot move that object
    // while we call native functions.
    pin_ptr<const wchar_t> wch = PtrToStringChars(orig);

    // Make a copy of the System::String as a multibyte
    // char* string. Allocate two bytes in the multibyte
    // output string for every wide character in the input
    // string, including space for a terminating null.
    size_t origsize = wcslen(wch) + 1;
    const size_t newsize = origsize*2;
    size_t convertedChars = 0;
    char *nstring = new char[newsize];
    wcstombs_s(&convertedChars, nstring, newsize, wch, _TRUNCATE);
    cout << nstring << " (char *)" << endl;

    // Convert a wide character System::String to a
    // wide character wchar_t* string.
    const size_t newsizew = origsize;
    wchar_t *wcstring = new wchar_t[newsizew];
    wcscpy_s(wcstring, newsizew, wch);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert a wide character System::String to a
    // wide character _bstr_t string.
    _bstr_t bstrt(wch);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert a wide character System::String
    // to a wide character CComBSTR string.
    CComBSTR ccombstr(wch);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Make a multibyte copy of the CComBSTR string
        // and display the result.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a wide character System::String to
    // a multibyte CStringA string.
    CStringA cstring(wch);
    cstring += " (CStringA)";
    cout << cstring << endl;

    // Convert a wide character System::String to
    // a wide character CStringW string.
    CStringW cstringw(wch);
    cstringw += " (CStringW)";
    wcout << (LPCTSTR)cstringw << endl;

    // Convert a wide character System::String to
    // a wide character basic_string.
    wstring basicstring(wch);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    delete orig;
}
```

```
Hello, World! (System::String)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
```

See also

ATL and MFC String Conversion Macros
CString Operations Relating to C-Style Strings

How to: Convert Standard String to System::String
How to: Convert System::String to Standard String
How to: Convert System::String to wchar_t* or char*
Programming with CComBSTR
mbstowcs_s, _mbstowcs_s_l
wcstombs_s, _wcstombs_s_l
strcpy_s, wcscpy_s, _mbscpy_s
strcat_s, wcscat_s, _mbscat_s
pin_ptr (C++/CLI)