Migrating Windows-Based Programs to Unicode

Creating a new program based on Unicode is fairly easy. Most of the logic used to handle Unicode characters is the same as for ASCII. Unicode has a few features that require special handling, but you can isolate these in your code. Converting an existing program that uses ANSI to one that uses Unicode or generic declarations is also straightforward. If you port your application to Win32 at the same time you port it to Unicode, you might find that PortTool, which comes with the Win32 SDK, is helpful. (See the Win32 SDK documentation for more information.) Following are 12 steps for changing an application so that you can use Unicode on Windows.

(1) Modify your code to use generic data types. Determine which variables declared as *char* or *char** are text, and not pointers to buffers or binary byte arrays. Change these types to TCHAR and TCHAR*, as defined in the Win32 file WINDOWS.H, or to _TCHAR as defined in the Visual C++ file TCHAR.H. Replace instances of LPSTR and LPCH with LPTSTR and LPTCH. Make sure to check all local variables and return types. Using generic data types is a good transition strategy because you can compile both ANSI and Unicode versions of your program without sacrificing the readability of the code. Don't use generic data types, however, for data that will always be Unicode or always ANSI. For example, one of the string parameters to *MultiByteToWideChar* and *WideCharToMultiByte* should always be in ANSI and the other should always be in Unicode.

(2) Modify your code to use generic function prototypes. For example, use the C run-time call *_tcslen* instead of *strlen,* and use the Win32 API *GetLocaleInfo* instead of *GetLocaleInfoA*. If you are also porting from 16 bits to 32 bits, most Win32 generic function prototypes conveniently have the same name as the corresponding Windows 3.1 API calls. (*TextOut* is one good example.) Besides, the Win32 API is documented using generic types. If you plan to use Visual C++ 2, become familiar with the available wide-character functions so that you'll know what kind of function calls you need to change. Always use generic data types when using generic function prototypes.

(3) Surround any character or string literal with the TEXT macro. The TEXT macro conditionally places an *L* in front of a character literal or a string literal. The C run-time equivalents are _T and _TEXT. Be careful with escape sequences. For example, the Win32 resource compiler interprets L\" as an escape sequence specifying a 16-bit Unicode double-quote character, not as the beginning of a Unicode string. Visual C++ 2 treats anything within L" " quotes as a multibyte string and translates it to Unicode byte by byte, based on the current locale, using *mbtowc*. One possible way to create a string with Unicode hex values is to create a regular string and then coerce it to a Unicode string (while paying attention to byte order).

```
char foo[4] = 0x40,0x40,0x40,0x41;
wchar_t *wfoo = (wchar_t *)foo;
```

(4) Create generic versions of your data structures. Type definitions for string or character fields in structures should resolve correctly based on the UNICODE compile-time flag. If you write your own string-handling and character-handling functions, or functions that take strings as parameters, create Unicode versions of them and define generic prototypes for them.

(5) Change your make process. When you want to build a Unicode version of your application, both the Win32 compile-time flag -DUNICODE and the C run-time compile-time flag -D_UNICODE must be defined.

(6) Adjust pointer arithmetic. Subtracting char* values yields an answer in terms of bytes; subtracting wchar_t* values yields an answer in terms of 16-bit chunks. When determining the number of bytes (for example, when allocating memory), multiply by *sizeof(TCHAR)*. When determining the number of characters from the number of

bytes, divide by *sizeof(TCHAR)*. You can also create macros for these two operations, if you prefer. C makes sure that the ++ and -- operators increment and decrement by the size of the data type.

(7) Check for any code that assumes a character is always 1 byte long. Code that assumes a character's value is always less than 256 (for example, code that uses a character value as an index into a table of size 256) must be changed. Remember that the ASCII subset of Unicode is fully compatible with 7-bit ASCII, but be smart about where you assume that characters will be limited to ASCII. Make sure your definition of NULL is 16 bits long.

(8) Add Unicode-specific code if necessary. In particular, add code to map data "at the boundary" to and from Unicode using the Win32 functions *WideCharToMultiByte* and *MultiByteToWideChar,* or using the C run-time functions *mbtowc, mbstowcs, wctomb,* and *wcstombs*. *Boundary* refers to systems such as Windows 95, to old files, or to output calls, all of which might expect or provide non□Unicode-encoded characters.

(9) Add code to support special Unicode characters. These include Unicode characters in the compatibility zone, characters in the private-use zone, combining characters, and characters with directionality. Other special characters include the private-use□zone non-character U+FFFF, which can be used as a sentinel, and the byte order marks U+FEFF and U+FFFE, which can serve as flags that indicate a file is stored in Unicode. The byte order marks are used to indicate whether a text stream is little-Endian or big-Endian□that is, whether the high-order byte is stored first or last. In plaintext, the line separator U+2028 marks an unconditional end of line. Inserting a paragraph separator, U+2029, between paragraphs makes it easier to lay out text at different line widths.

(10) Determine how using Unicode will affect file I/O. If your application will exist in both Unicode and non-Unicode variations, you'll need to consider whether you want them to share a file format. Standardizing on an 8-bit character set data file will take away some of the benefits of having a Unicode application. Having different file formats and adding a conversion layer so each version can read the other's files is another alternative. Even if you choose a Unicode file format, you might have to support reading in old non-Unicode files or saving files in non-Unicode format for backward compatibility. Also, make sure to use the correct *printf*-style format specifiers for Visual C++ 2, shown here:

Specifier	***printf Expects***	***wprintf Expects***
%s	SBCS or MBCS	Unicode
%S	Unicode	SBCS or MBCS
%hs	SBCS or MBCS	SBCS or MBCS
%ls	Unicode	Unicode

(11) Double-check the way in which you retrieve command line arguments. Use the function *GetCommandLine* rather than the *lpCmdLine* parameter (an ANSI string) that is passed to *WinMain*. *WinMain* cannot accept Unicode input because it is called before a window class is registered.

(12) Debug your port by enabling your compiler's type-checking. Do this (using W3 on Microsoft compilers) with and without the UNICODE flag defined. Some warnings that you might be able to ignore in the ANSI world will cause problems with Unicode. If your original code compiles cleanly with type-checking turned on, it will be easier to port. The warnings will help you make sure that you are not passing the wrong data type to code that expects wide-character data types. Use the Win32

NLSAPI or equivalent C run-time calls to get character typing and sorting information. Don't try to write your own logic □your application will end up carrying very large tables!

© 2017 Microsoft