WinAPI: Being Unicode Friendly

Disch (13742) Nov 28, 2009 at 7:20am

Section 0) Introduction

This article is to go over being Unicode Friendly in WinAPI.

I don't normally encourage programming in WinAPI, since you're typically better off with a crossplatform widgetry lib such as wxWidgets or QT or whatever. But a lot of people still like to use WinAPI directly... so I should at least point them in the right direction. Besides, a lot of the stuff here applies to wx as well (and possibly to QT, though I've never used QT so I can't say for sure).

I didn't put a lot of work into formatting or proofreading this article. So my apologies there. I still think it gets the idea across pretty well, even if my throughts are unorganized.

Unicode forever! Spread the love!

Section 1) The UNICODE macro

The UNICODE macro (and/or _UNICODE macro -- usually both) is scattered throughout all of WinAPI. It redefines some types and functions to use either char* strings (if it's not defined) or wchar_t* Unicode strings (if it is defined).

If you use MSVS, these macros are often automatically defined by the compiler before it begins compiling if you set your project settings to make the program a Unicode program. Otherwise you can do it yourself by #defining them before you include Windows.h:

```
#ifndef UNICODE
#define UNICODE
#endif

```

You don't need to #define either of them to use Unicode in your program. It just changes around some types to make it easier to use the Unicode parts of WinAPI.

Further in this article, "Unicode build" refers to UNICODE and _UNICODE being defined, whereas "ANSI build" refers to neither of them being defined.

Section 2) LPSTR, LPCSTR, LPTSTR, LPCTSTR, LPWTFISALLTHIS

Anybody who's looked at WinAPI has probably seen the above types... but what exactly are they?

An inexperienced C/C++ coder might think they're strings, like std::string. It can certainly look that way from the documentation and examples. And since WinAPI pages doesn't ever really seem to tell you exactly what they are, it's a logical conclusion.

However, this is not the case. All of the above are ***macros*** which #define different types.

Now you might look at "LPCTSTR" and see the "STR" in there, but the rest might look like random letter combinations that make no sense. Rest assured there's a method to the madness.

- The starting 'LP' stands for "Long Pointer". Without getting too much into what a Long Pointer is (or really what it used to be, it doesn't have as much meaning in modern computing), we'll just say that this is basically a pointer. This means that the LP is telling you that this type is not a string by itself, but is a POINTER to a string (or really, a C-style string).

- The 'C' means that the string is constant

- The 'W' means the string is wide (Unicode)

- The 'T' means the string is TCHARs (see section on TCHAR below)

So really, the #defines are the following:

```
#define  LPSTR          char*
#define  LPCSTR         const char*

#define  LPWSTR         wchar_t*
#define  LPWCSTR        const wchar_t*

#define  LPTSTR         TCHAR*
#define  LPCTSTR        const TCHAR*
```

Section 3) TCHAR, _T(), T(), TEXT()

TCHAR is #defined as either a char or a wchar_t depending on whether or not the UNICODE macro was defined.

By using TCHARs properly, you can create both ANSI and Unicode builds of your program. All you have to do is #define UNICODE if you want a Unicode build, or don't define it if you want an ANSI build.

This presents a bit of a problem, though. String literals in C++ can take 2 forms, either char or wchar_t:

```
const char*    a = "Foo";
const wchar_t* b = L"Bar";  // <-- note the L.  That makes it wide.
```

The compiler doesn't auto-detect... so things like this would throw compiler errors:

```
const char*    a = L"Foo"; // <-- error, can't point char* to a wide string
const wchar_t* b = "Bar";  // <-- error, can't point wchar_t* to a non-wide string
```

So what about this?:

```
const TCHAR*   c = "Foo";
```

Remember that TCHAR is char or wchar_t depending on Unicode. So the above code will work ***only if*** you are not building Unicode. If you are building Unicode you'll get an error.

Likewise, the following won't work ***unless*** you're building Unicode:

```
const TCHAR*   c = L"Foo";
```

To get around this problem... WinAPI provides some other macros, _T(), T(), and TEXT(), all of which do the same thing. In a Unicode build, they put the L before the string literal to make it wide, and in non-Unicode, they do nothing. Therefore they will always work hand in hand with TCHARs:

```
const TCHAR*   d = _T("foo");  // works in both Unicode and ANSI builds
```

Section 4) Function and Structure Name Aliases

A lot of Windows functions take strings as parameters. But because char and wchar_t strings are two distinctly different types, the same function can't be used for both of them.

Take for example, the WinAPI function "DeleteFile" which takes a single parameter. Let's say you want to delete "myfile.txt":

```
DeleteFile( _T("myfile.txt") );  // notice _T because DeleteFile takes a LPCTSTR
```

The trick here is that the function DeleteFile doesn't really exist! There are actually two different functions:

```
DeleteFileA( LPCSTR );  // ANSI version, taking a LPCSTR
DeleteFileW( LPCWSTR ); // Unicode version, taking LPCWSTR
```

DeleteFile is actually a ***macro*** defined as either DeleteFileA or DeleteFileW, depending on whether or not this is a Unicode build.

As such... for WinAPI functions that take a C style string... there are, in a sense, 3 different versions, each taking a different type of C string:

```
DeleteFile   <-  Takes a TCHAR string (LPCTSTR)
DeleteFileA  <-  Takes a char string (LPCSTR)
DeleteFileW  <-  Takes a wchar_t string (LPCWSTR)
```

This is true of virtually all WinAPI functions that take a C string as a param.

But it doesn't stop there! There are also some structs that have strings in them, as well. For instance, the OPENFILENAME structure contains various C strings for use with the open file dialog box. As you might expect, There are 3 versions of this struct as well:

```
OPENFILENAME  <-  has TCHAR strings
OPENFILENAMEA <-  has char strings
OPENFILENAMEW <-  has wchar_t strings
```

And again... note that OPENFILENAME doesn't ***really*** exist, but is just a #define of one of the other two depending on the build.

Last edited on Nov 28, 2009 at 7:27am

Disch (13742) Nov 28, 2009 at 7:21am

Section 5) Being Unicode friendly

So what does it take to be Unicode friendly in WinAPI?

For most programs... not very much. Just stick to the following and you'll be fine:

-) Use TCHAR for characters and C strings instead of char
-) Use std::basic_string<TCHAR> instead of std::string. You can even typedef your own kind of tstring:
`typedef std::basic_string<TCHAR> tstring;`
-) Don't use std::string, as this is a char string.
-) Put all string literals in `_T()` macros. UNLESS you are dealing with libs other than WinAPI. For example, standard lib functions like fstream ctors take char* strings -- so don't put those strings in _T() macros. Really, though, if you're using WinAPI, you shouldn't be using standard lib file I/O because the standard lib is not Unicode friendly.
-) Don't use standard lib C string functions like strcpy, strcat, sprintf, etc. These all work with char -- they don't work with wchar_t or TCHAR. Alternatively you can use 'tstring' member functions, and Windows specific TCHAR functions like _tcscpy, _tcscat, etc.
-) ***Never ever ever*** C style cast C strings from one type to another. C style casts mask very important compiler errors. Avoid C++ style casts also. Basically if you're getting type errors with your strings -- it's because you're doing something wrong. Don't try to cast around the problem.
-) Switch between ANSI builds and Unicode builds often to make sure that your program will compile in both. If that's too much of a hassle, make Unicode builds all the time and forget about ANSI builds.

For other programs where you do a lot of text manipulation, it gets a little trickier....

-) Be careful when reading or writing text to a file. Don't use TCHAR for this, since its size is variable. Use char if you're reading 8-bit characters from a file, and wchar_t if reading 16-bit characters.

-) Ideally if text is going in an output file, you should use a Unicode encoding, such as UTF-8 or UTF-16. However that is beyond the scope of this article (perhaps another day!)

-) If you need to use char or wchar_t directly (for instance the above situation), be very careful about how you move those strings to a TCHAR string. You'll typically have to copy the string over 1 character at a time or write your own string copy function to do that. I don't think WinAPI has any functions to help with such a case, and I know the standard lib doesn't.

For example:

```
// this function copies a char C string to a TCHAR C string:
void ustrcpy(TCHAR* dst, const char* src)
{
  while(*src)
  {
   *dst = *src;
   ++dst;
   ++src;
  }
  *dst = *src;
}

//---------
//  then, say you need to read a string from a file and put it in a text box with Se

char str[500] = {0};            // note I'm using char because I specifically want 8
ifstream myfile("myfile.txt");  // note no _T() macro because I'm dealing with std I
                                //  ideally you'd open the file with WinAPI's Create
                                //  that way because that is Unicode friendly.  Howe
                                //  to keep this example simple
myfile >> str;      // read the string

TCHAR buffer[500];  // need to copy to a TCHAR buffer in order to give it to SetWind
ustrcpy( buffer, str );

// give it to WinAPI
SetWindowText( hMyTextBox, buffer );
```

A better approach would be to make template functions for ustrcpy and similar so you can convert to/from all sorts of different types and sizes:

```
template <typename T, typename TT>
void ustrcpy( T* dst, const TT* src )
{
  //.. same as above
}
```

Alternatively... you can avoid the TCHAR version of the WinAPI function and use the ANSI version directly. This let's Windows take care of the conversion:

```
char str[500] = {0};
myfile >> str;

 // note here we specifically call SetWindowTextA, not SetWindowText.
 // this is because we're giving a char string and not a TCHAR string.
SetWindowTextA( hMyTextBox, str );
```

More to come? ???

Last edited on Nov 28, 2009 at 7:35am

Null (957) Nov 28, 2009 at 11:43am

I tried to compile this and i got an error: Illegal byte sequence
How can I fix this?

```
// compiled using mingw-g++
#define UNICODE
#define _UNICODE

```

```
#include <iostream>
#include <windows.h>
#include <tchar.h>
using namespace std;

wchar_t *str=_T("aaa ąčęėįšųūž");
COORD c;
int main()
{
DWORD d;
HANDLE hcon=GetStdHandle(STD_OUTPUT_HANDLE);
WriteConsoleOutputCharacterW(hcon,str,sizeof(str),c,0);


                cin.get();
        return 0;
}
```

Edit & run on cpp.sh

Disch (13742) Nov 28, 2009 at 5:52pm

Oho! This is a problem I didn't consider when writing the article:

```
wchar_t *str=_T("aaa ąčęėįšųūž");
```

This is problematic for a few reasons.

The error is probably being caused because your file isn't being saved as UTF-8 encoding. Try looking around in your settings for whatever text editor you're using to make sure.

But that might not the only problem.. once you get it to compile, it won't output what you expect! At least I don't think so. It might if gcc parses UTF-8 properly -- try it and see. I'm actually curious whether or not it works.

If it doesn't work, using Unicode in string literals like the above might be more trouble than it's worth. I can come up with a solution, but before I try to figure one out let's see if you have the problem I expect. Let me know!

Null (957) Nov 28, 2009 at 6:29pm

Source file saved as UTF-8: (list of errors)

```
main.cpp:1: error: stray '\239' in program
main.cpp:1: error: stray '\187' in program
main.cpp:1: error: stray '\191' in program
main.cpp:1: error: invalid token
main.cpp:1: error: `define' does not name a type
[few other errors here]
```

Source file saved in unicode format:

```
]
main.cpp:3:9: warning: null character(s) ignored
main.cpp:3:11: warning: null character(s) ignored
main.cpp:3:13: warning: null character(s) ignored
main.cpp:3:15: warning: null character(s) ignored
main.cpp:3:19: warning: null character(s) ignored
main.cpp:3:21: warning: null character(s) ignored
main.cpp:3:23: warning: null character(s) ignored
main.cpp:3:25: warning: null character(s) ignored
main.cpp:3:27: warning: null character(s) ignored
main.cpp:3:29: warning: null character(s) ignored
main.cpp:3:31: warning: null character(s) ignored
main.cpp:3:33: warning: null character(s) ignored
main.cpp:4:1: warning: null character(s) ignored
main.cpp:5:1: warning: null character(s) ignored
main.cpp:6:1: warning: null character(s) ignored
main.cpp:7:1: warning: null character(s) ignored
main.cpp:7:3: warning: null character(s) ignored
main.cpp:7:4: invalid preprocessing directive #i
[...]
```

Saving source file as ANSI destroys ąčęėįšųūž characters...

Disch (13742) Nov 28, 2009 at 7:03pm

Ah

Your text editor is dumb. It's putting a BOM at the start of a UTF-8 file which it shouldn't do. You'll have to switch to a better/more compliant editor or tweak its settings so it doesn't save the BOM.

EDIT:

On a side note... I tested this on GCC (on Linux -- can't test your whole program on Windows =( ):

```
const wchar_t *str= L"šū";
```

and the resulting string was as expected. So the good news is that once you straighten out the issue with your editor, that code should work.

Last edited on Nov 28, 2009 at 7:08pm

Null (957) Nov 28, 2009 at 7:08pm

I used Wordpad and notepad... and what is BOM?

Disch (13742) Nov 28, 2009 at 7:16p

Well then Wordpad and notepad are no good.

An IDE is better, IMO anyway. I'd recommend getting Code::Blocks. Or you could try TextPad (not an IDE, but has syntax highlighting for C++ and is pretty decent. I'm not sure how well it handles Unicode though)

BOM stands for "Byte Order Marker". It's a special unicode character (U+FFFE) used to identify the endianness of UTF-16 text files.

IE: a text file would start with 'FF FE' if it is UTF-16 BE, or 'FE FF' if it is UTF-16 LE.

Since endianness doesn't matter with UTF-8, you don't need a BOM (and in fact, the standard says you shouldn't have one). But apparently Notepad and Wordpad put it there anyway.. presumably to identify the file as Unicode rather than ASCII.

In UTF-8, U+FFFE is represented as 'EF BF BE', which are the 3 garbage characters giving you errors on line 1 of your program.

Null (957) Nov 28, 2009 at 7:35pm

Yes, you were right. I opened my source code with hex editor and first 3 bytes were EF BB BF.

Null (957) Nov 28, 2009 at 8:23pm

I thought it's easy. Looks like not:

```cpp
#define UNICODE
#define _UNICODE
#define _GLIBCXX_USE_WCHAR_T  // wcout doesn't work without this


#include <tchar.h>
#include <iostream>
#include <fstream>
#include <conio.h>
#include <windows.h>

using namespace std;


int main()
{

  wchar_t buf[512];




wcin>>buf;
wcout <<endl<<buf;
getch();
  return 0;
}
```

Edit & run on cpp.sh

When font is "Lucida console", then input is OK; I can enter ėjšž and other chars but wcout outputs wrong: eizs. What's wrong here

Disch (13742) Nov 28, 2009 at 8:35pm

wcout doesn't work like you'd expect. It's pretty much useless.

I was helping someone with a similar problem a long while back:

http://cplusplus.com/forum/windows/9797/

He was looking at outputting UTF-8 though, not really wide character strings.

Basically it boils down to this: The standard lib sucks at Unicode (at least for Windows -- cout/cin deal with UTF-8 just fine on my Linux box). If you want to be Unicode friendly, you have to use WinAPI.

Try the following:

```
int main()
{
  HANDLE hin = GetStdHandle(STD_INPUT_HANDLE);
  HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
  wchar_t buf[512];
  DWORD rd;

  ReadConsoleW( hin, buf, 512, &rd, 0 );
  WriteConsoleW( hout, buf, rd, &rd, 0 );

  getch();

  return 0;
}
```

EDIT:

It's also worth noting that I wrote this article with SDI/MDI window-style programs in mind, not really Console programs, since my interest in the console is zero.

Last edited on Nov 28, 2009 at 8:42pm

Topic archived. No new replies allowed.

x

Spotted an error? contact us

x