Why does misinterpreting UTF16-LE Unicode text as ANSI tend to show up as just one character?

devblogs.microsoft.com/oldnewthing/20180207-00

February 7, 2018

Raymond Chen

If you misinterpret ANSI text as Unicode, you usually get nonsense Chinese text. If you misinterpret Unicode text as ANSI, why do you usually just get the first character?

Okay, this one is a lot easier.

The Latin alphabet fits in the range U+0041 through U+007A. If you're using the UTF16-LE encoding (which is what Unicode means in the context of Windows), then the first byte will be the correct character, and the second byte will be zero, which will serve as the string terminator.

For example, `(char*)L"Abc"` will act like `"A"`.

I remember looking at the registry and finding a registry key directly under `HKEY_CURRENT_USER` called simple `S`. In other words, the program stored its settings under `HKEY_CURRENT_USER\S`.

This bugged me enough that I dove in to figure out how this happened.

The program in question had a Windows 95 version and a Windows NT version. They compiled both versions from the same code base by using the `TCHAR`-style functions, so that when compiled for Windows 95, it was an ANSI program, and when compiled for Windows NT, it was Unicode.

The program came with a helper DLL, which was also compiled as ANSI for Windows 95 and as Unicode for Windows NT. The name of the DLL was not inside an `#ifdef`, so even though the code was compiled twice, both versions of the DLL had the same name.

Furthermore, the `.def` file and the internal library's header file did not contain any `#ifdef`s either. So the Windows 95 version of `HELPER.DLL` had an exported function called `CreateRegistryKey` (say), which accepted an ANSI string. And the Windows NT version of `HELPER.DLL` also had an exported function called `CreateRegistryKey`, but which accepted a Windows NT string.

The problem was that their Windows NT product shipped with the Windows 95 version of the helper DLL!

Since the DLL name was the same, and the function names were the same, the operating system happily loaded the DLL and imported the function name successfully, even though it was the wrong function.

As a result, the Windows NT version passed a Unicode string to a function that interpreted it as an ANSI string, and the registry key name `Software` became misinterpreted as just `S`.

There are a few ways of avoiding the problem.

The obvious one is to abandon the Windows 95 version of the product. Because c'mon now.

Okay, but let's go back in time to a period when supporting Windows 95 was still a reasonable thing to do.

One option is to give the Windows 95 and Windows NT versions of the DLL different names, say, `HELPERA.DLL` and `HELPERW.DLL`. That way, when a program linked to `HELPERW.DLL` but you accidentally put `HELPERA.DLL` in the product directory, you would get a "DLL not found" error instead of running ahead with the wrong DLL.

Mind you, this solution would catch the problem only if it occurred at packaging. But if the problem was that the code linked together some object files compiled in ANSI mode and some object files compiled in Unicode mode, say because you used the wrong version of a static library, then the error would go undetected because both sets of object files will look for the function `CreateRegistryKey`, and if the module was linked with (say) `HELPERW.LIB`, then both sets of object files will link to `HELPERW.DLL`, even though half of them thought they were linking to `HELPERA.DLL`.

What they should have done was change the names of the exports. Export two functions `CreateRegistryKeyA` and `CreateRegistryKeyW`. Use an inline helper function or a macro in the header file so that ANSI clients are directed to `CreateRegistryKeyA` and Unicode clients are directed to `CreateRegistryKeyW`. The implementation of the helper DLL need only implement the versions of the functions corresponding to the desired character set. In other words, `HELPERA.DLL` implements `CreateRegistryKeyA` and `HELPERW.DLL` implements `CreateRegistryKeyW`. (If you use macros, then this happens automatically when you implement `CreateRegistryKey`.)

This design solves a few problems.

- If you package the wrong DLL, the file names will not match and you'll get an error at load time.

- If you have a mix of object files, you will get a linker error because `HELPERA.LIB` won't have entries for the Unicode versions, and vice versa.
- If you really needed to support the mixed version, you could link to both `HELPERA.LIB` And `HELPERW.LIB`. Each object file will pull the function it needs from the appropriate import library, and will bind to the corresponding DLL at runtime.
- In the future, you might decide to merge the helper libraries into a single helper library that supports both character sets. Giving the functions distinct names allows this to happen. (This is what most of Windows does. For example, `kernel32.dll` contains both ANSI and Unicode implementations of many functions, distinguished by function name.)

Moral of the story: If two functions are different, give them different names. (If you use mangled names, then the names will already be different due to different mangling.)

Related: What is `__wchar_t` (with the leading double underscores) and why am I getting errors about it?

Raymond Chen

Follow