The sad history of Unicode printf-style format specifiers in Visual C++

devblogs.microsoft.com/oldnewthing/20190830-00

August 30, 2019

Raymond Chen

Windows adopted Unicode before most other operating systems.[citation needed] As a result, Windows's solutions to many problems differ from solutions adopted by those who waited for the dust to settle.[1] The most notable example of this is that Windows used UCS-2 as the Unicode encoding. This was the encoding recommended by the Unicode Consortium because Unicode 1.0 supported only 65536 characters.[2] The Unicode Consortium changed their minds five years later, but by then it was far too late for Windows, which had already shipped Win32s, Windows NT 3.1, Windows NT 3.5, Windows NT 3.51, and Windows 95, all of which used UCS-2.[3]

But today we're going to talk about `printf`-style format strings.

Windows adopted Unicode before the C language did. This meant that Windows had to invent Unicode support in the C runtime. The result was functions like `wcscmp`, `wcschr`, and `wprintf`. As for `printf`-style format strings, here's what we ended up with:

- The `%s` format specifier represents a string in the same width as the format string.
- The `%S` format specifier represents a string in the opposite width as the format string.
- The `%hs` format specifier represents a narrow string regardless of the width of the format string.
- The `%ws` and `%ls` format specifiers represent a wide string regardless of the width of the format string.

The idea behind this pattern was so that you could write code like this:

```
TCHAR buffer[256];
GetSomeString(buffer, 256);
_tprintf(TEXT("The string is %s.\n"), buffer);
```

If the code is compiled as ANSI, the result is

```
char buffer[256];
GetSomeStringA(buffer, 256);
printf("The string is %s.\n", buffer);
```

And if the code is compiled as Unicode, the result is[4]

```
wchar_t buffer[256];
GetSomeStringW(buffer, 256);
wprintf(L"The string is %s.\n", buffer);
```

By following the convention that `%s` takes a string in the same width as the format string itself, this code runs properly when compiled either as ANSI or as Unicode. It also makes converting existing ANSI code to Unicode much simpler, since you can keep using `%s`, and it will morph to do what you need.

When Unicode support formally arrived in C99, the C standard committee chose a different model for `printf` format strings.

- The `%s` and `%hs` format specifiers represent an narrow string.
- The `%ls` format specifier represents a wide string.

This created a problem. There were six years and untold billions of lines of code in the Windows ecosystem that used the old model. What should the Visual C and C++ compiler do?

They chose to stick with the existing nonstandard model, so as not to break every Windows program on the planet.

If you want your code to work both on runtimes that use the Windows classic `printf` rules as well as those that use C standard `printf` rules, you can limit yourself to `%hs` for narrow strings and `%ls` for wide strings, and you'll get consistent results regardless of whether the format string was passed to `sprintf` or `wsprintf`.

```
#ifdef UNICODE
#define TSTRINGWIDTH TEXT("l")
#else
#define TSTRINGWIDTH TEXT("h")
#endif

TCHAR buffer[256];
GetSomeString(buffer, 256);
_tprintf(TEXT("The string is %") TSTRINGWIDTH TEXT("s\n"), buffer);

char buffer[256];
GetSomeStringA(buffer, 256);
printf("The string is %hs\n", buffer);

wchar_t buffer[256];
GetSomeStringW(buffer, 256);
wprintf("The string is %ls\n", buffer);
```

Encoding the `TSTRINGWIDTH` separately lets you do things like

```
_tprintf(TEXT("The string is %10") TSTRINGWIDTH TEXT("s\n"), buffer);
```

Since people like tables, here's a table.

Format		Windows classic	C standard	
`%s`	`printf`	`char*`	`char*`	⇐
`%s`	`wprintf`	`wchar_t*`	`char*`	
`%S`	`printf`	`wchar_t*`	N/A	
`%S`	`wprintf`	`char*`	N/A	
`%hs`	`printf`	`char*`	`char*`	⇐
`%hs`	`wprintf`	`char*`	`char*`	⇐
`%ls`	`printf`	`wchar_t*`	`wchar_t*`	⇐
`%ls`	`wprintf`	`wchar_t*`	`wchar_t*`	⇐
`%ws`	`printf`	`wchar_t*`	N/A	
`%ws`	`wprintf`	`wchar_t*`	N/A	

I highlighted the rows where the C standard agrees with the Windows classic format.[5] If you want your code to work the same under either format convention, you should stick to those rows.

[1] You'd think that adopting Unicode early would give Windows the first-mover advantage, but at least with respect to Unicode, it ended up being a first-mover disadvantage, because everybody else could sit back and wait for better solutions to emerge (such as UTF-8) before beginning their Unicode adoption efforts.

[2] I guess they thought that 65536 characters should be enough for anyone.

[3] This was later upgraded to UTF-16. Fortunately, UTF-16 is backward compatible with UCS-2 for the code points that are representable in both.

[4] Technically, the Unicode version was

```
unsigned short buffer[256];
GetSomeStringW(buffer, 256);
wprintf(L"The string is %s.\n", buffer);
```

because there was not yet a `wchar_t` as an independent type. Prior to the introduction of `wchar_t` to the standard, the `wchar_t` type was just a synonym for `unsigned short`. The changing fate of the `wchar_t` type has its own story.

[5] The Windows classic format came first, so the question is whether the C standard chose to align with the Windows classic format, rather than vice versa.

Raymond Chen

Follow