What('s) a character!

devblogs.microsoft.com/oldnewthing/20070105-00

January 5, 2007

Raymond Chen

Norman Diamond seems to have made a side career of harping on this topic on a fairly regular basis, although he never comes out and says that this is what he's complaining about. He just assumes everybody knows. (This usually leads to confusion, as you can see from the follow-ups.) Back in the ANSI days, terminology was simpler. Windows operated on `CHAR` s, which are one byte in size. Buffer sizes were documented as specified in bytes, even for textual information. For example, here's a snippet from the 16-bit documentation for the `GetWindowTextLength` function:

> The return value specifies the text length, in bytes, not including any null terminating character, if the function is successful. Otherwise, it is zero.

The use of the term *byte* throughout permitted the term *character* to be used for other purposes, and in 16-bit Windows, the term was repurposed to represent "one or bytes which together represent one (what I will call) linguistic character." For single-byte character sets, a linguistic character was the same as a byte, but for multi-byte character sets, a linguistic character could be one or two bytes. Documentation for functions that operated on linguistic characters said *characters*, and functions that operated on `CHAR` s, said *bytes*, and everybody knew what the story was. (Mind you, even in this nostalgic era, documentation would occasionally mess up and say *character* when they really meant *byte*, but the convention was adhered to with some degree of consistentcy.) With the introduction of Unicode, things got ugly. All documentation that previously used *byte* to describe the size of textual data had to be changed to read "the size of the buffer in bytes if calling the ANSI version of the function or in `WCHAR` s if calling the Unicode version of the function." A few years ago the Platform SDK team accepted my suggestion to adopt the less cumbersome "the size of the buffer in `TCHAR` s." Newer documentation from the core topics of the Platform SDK tends to use this alternate formulation. Unfortunately, most documentation writers (and 99% of software developers, who provide the raw materials for the documentation writers) aren't familiar with the definition of *character* that was set down back in 1983, and they tend to use the term to mean *storage character*, which is a term I invented just now to mean "a unit of storage sufficient to hold a single `TCHAR` ." (The Platform SDK uses what I consider to be the fantastically awkward term *normal character widths*.) For example, the `lstrlen` function returns the length of the string in storage characters, not linguistic

characters. And any function that accepts a sized output buffer obviously specifies the size in storage characters because the alternative is nonsense: How could you pass a buffer and say "Please fill this buffer with data. Its size is five linguistic characters"? You don't know what is going into the buffer, and a linguistic character is variable-sized, so how can you say how many linguistic characters will fit? Michael Kaplan enjoys making rather outrageous strings which result in equally outrageous sort keys. I remember one entry a while ago where he piled over a dozen accent marks over a single "a". That "a" plus the combining diacritics all equal one giant linguistic character. (There is a less extreme example here, wherein he uses an "e" plus two combining diacritics to form one linguistic character.) If you wanted your buffer to really be able to hold five of these extreme linguistic characters, you certainly would need it to be bigger than `WCHAR buffer[5]`.

As a result, my recommendation to you, dear reader, is to enter every page of documentation with a bias towards *storage character* whenever you see the word *character*. Only if the function operates on the textual data linguistically should you even consider the possibility that the author actually meant *linguistic character*. The only functions I can think of off-hand that operate on linguistic characters are `CharNext` and `CharPrev`, and even then they don't quite get it right, although they at least try.

Raymond Chen

Follow