A popular but wrong way to convert a string to uppercase or lowercase

devblogs.microsoft.com/oldnewthing/20241007-00

October 7, 2024

It seems that a popular way of converting a string to uppercase or lowercase is to do it letter by letter.

```
std::wstring name;

std::transform(name.begin(), name.end(), name.begin(),
    std::tolower);
```

This is wrong for many reasons.

First of all, `std::tolower` is not an *addressible function*. This means, among other things, that you are not allowed to take the function's address,[1] like we're doing here when we pass a pointer to the function to `std::transform`. So we'll have to use a lambda.

```
std::wstring name;

std::transform(name.begin(), name.end(), name.begin(),
    [](auto c) { return std::tolower(c); });
```

The next mistake is a copy-pasta: The code is using `std::tolower` to convert wide characters (`wchar_t`) even though `std::tolower` works only for narrow characters (even more restrictive than that: it works only for *unsigned* narrow characters `unsigned char`). There is no compile-time error because `std::tolower` accepts an `int`, and on most systems, `wchar_t` is implicitly promotable to `int`, so the compiler accepts the value without complaint even though over 99% of the potential values are out of range.

Even if we fix the code to use `std::towlower`:

```
std::wstring name;

std::transform(name.begin(), name.end(), name.begin(),
    [](auto c) { return std::towlower(c); });
```

it’s still wrong because it assumes that case mapping can be performed `char`-by-`char` or `wchar_t`-by-`wchar_t` in a context-free manner.

If the `wchar_t` encoding is UTF-16, then characters outside the basic multilingual plane (BMP) are represented by pairs of `wchar_t` values. For example, the Unicode character OLD HUNGARIAN CAPITAL LETTER A[2] (U+10C80) is represented by two UTF-16 code units: `D803` followed by `DC80`.

Passing these two code units to `towlower` one at a time prevents `towlower` from understanding how they interact with each other. If you call `towlower` with `DC80`, it recognizes that you passed only half of a character, but it doesn’t know what the other half is, so it has to just shrug its shoulders and say, “Um, `DC80`?” Too bad, because the lowercase version of OLD HUNGARIAN CAPITAL LETTER A (U+10C80) is OLD HUNGARIAN SMALL LETTER A (U+10CC0), so it should have returned `DCC0`. Of course `towlower` doesn’t have psychic powers, so you can’t really expect it to have known that the `DC80` was the partner of an unseen `D803`.

Another problem (which applies even if `wchar_t` is UTF-32) is that the uppercase and lowercase versions of a character might have different lengths. For example, LATIN SMALL LETTER SHARP S (“ß” U+00DF) uppercases to the two-character sequence “SS”:[3] Straße ⇒ STRASSE, and LATIN SMALL LIGATURE FL (“ﬂ” U+FB02) uppercases to the two-character sequence “FL”. In both examples, converting the string to uppercase causes the string to get longer. And in certain forms of the French language, capitalizing an accented character causes the accent to be dropped: à Paris ⇒ A PARIS. If the accented character à were encoded as LATIN SMALL LETTER A (U+0061) followed by COMBINING GRAVE ACCENT (U+0300), then converting to uppercase causes the string to get shorter.

Similar issues apply to the `std::string` version:

```
std::string name;

std::transform(name.begin(), name.end(), name.begin(),
    [](auto c) { return std::tolower(c); });
```

If the string potentially contains characters outside the 7-bit ASCII range, then this triggers undefined behavior when those characters are encountered. And for UTF-8 data, you have the same issues discussed before: Multibyte characters will not be converted properly, and it breaks for case mappings that alter string lengths.

Okay, so those are the problems. What’s the solution?

If you need to perform a case mapping on a string, you can use `LCMapStringEx` with `LCMAP_LOWERCASE` or `LCMAP_UPPERCASE`, possibly with other flags like `LCMAP_LINGUISTIC_CASING`. If you use the International Components for Unicode (ICU) library, you can use `u_strToUpper` and `u_strToLower`.

[1] The standard imposes this limitation because the implementation may need to add default function parameters, template default parameters, or overloads in order to accomplish the various requirements of the standard.

[2] I find it quaint that Unicode character names are ALL IN CAPITAL LETTERS, in case you need to put them in a Baudot telegram or something.

[3] Under the pre-1996 rules, the ß can capitalize under certain conditions to “SZ”: Maßen ⇒ MASZEN. And in 2017, the Council for German Orthography (*Rat für deutsche Rechtschreibung*) permitted LATIN CAPITAL LETTER SHARP S (“ẞ” U+1E9E) to be used as a capital form of ß.