Tip 73: Store Unicode in UTF-8

Ben Klemens

25 February 2012

level: citizen of the world
purpose: At the least, do no harm

Back when all the computing action was in the United States, ASCII (American Standard Code for Information Interchange) defined a numeric code for all of the usual letters and symbols printed on a standard U.S. QWERTY keyboard, which I will refer to as the naïve English character set. A C `char` is one byte = eight bits (binary digits) = 256 possible values. ASCII defined 128 characters, so it fit into a single `char` with even a bit to spare.

Unicode follows the same basic premise, assigning a single hexadecimal numeric value, typically between 0000 and FFFF, to every symbol used for human communication. By custom these *code points* are written with a form like U+0000. The work is much more ambitious and challenging, because it requires cataloging all the usual Western letters, tens of thousands of Chinese and Japanese characters, all the requisite glyphs for Klingon, Deseret, Ugaritic, and so on throughout the world and throughout human history.

Assigning a unique value to every symbol is already an amazing feat, and is largely a success: there is little if any ambiguity about which code point to use anywhere, in any language.

The next question is how it is to be encoded, and at this point things start to fall apart. The primary question is how many bytes to set as the unit of analysis. UTF-32 [UTF= UCS Transformation Format; UCS=Universal Character Set.] specifies 32 bits (4 bytes) as the basic unit, which means that every character can be encoded in a single unit, at the cost of a voluminous amount of empty padding, given that naïve English can be written with only 7 bits. UTF-16 uses two bytes as the basic unit, which handles most characters comfortably with a single unit but requires that some characters be written down using two. UTF-8 uses one byte as its unit, meaning still more code points written down via multi-unit amalgams.

I like to think about the UTF encodings as a sort of trivial encryption. For every code point, there is a single byte sequence in UTF-8, a single byte sequence in UTF-16, and a single byte sequence in UTF-32. Barring an exception discussed below, there is no reason to expect that the code point and the encrypted values are numerically the same, or even related in an obvious way, but I know that a properly programmed decoder can trivially translate among the UTF encodings and the correct Unicode code point.

So, as the author of a program that needs to communicate with humans who speak all kinds of languages, what are your responsibilities? You need to:

- work out what encoding the host system is using, so that you aren't fooled into using the wrong encoding to read inputs and can send back outputs that the host can correctly decode.
- successfully store text somewhere, unmangled.
- recognize that one character is not a fixed number of bytes, so any base-plus-offset code you write (given a Unicode string `us`, things like `us++`) may give you fragments of a code point.
- have on hand utilities to do any sort of comprehension of text: `toupper` and `tolower` work only for naïve English, so we will need replacements. [Tune in next time!]

Meeting these responsibilities will require picking the right internal encoding to prevent mangling, and having on hand a good library to help us when we need to decode.

The choice of internal coding is especially easy. UTF-8 was designed for you, the C programmer.

The UTF-8 unit is eight bits, which is what we C coders call a `char` (for the nonbelievers, C standard §5.2.4.2.1 defines it that way). *It is entirely valid to write a UTF-8 string to a* `char *` *variable, as with naïve English text.*

* The first 128 Unicode code points exactly match ASCII. For example, `A` = ASCII 41 = U+0041. Therefore, if your Unicode text happens to consist entirely of naïve English, then you can use the usual ASCII-oriented utilities on them, and if you send your naïve English text to the display claiming it is Unicode in UTF-8, all will work.

* U+0000 is a valid code point, which we C coders refer to as `'\0'`. (Because this is the ASCII zero as well, this rule is a special case of the last one). This is important because C strings need to have exactly one null character marking the end of the string, so UTF-8 strings are valid C `char *` strings. [Impish users may send you a string with a null in the middle, but that was a risk with ASCII as well, and users who did that kind of thing don't get to be surprised when their strings get truncated.]

Recall how the unit for UTF-16 and UTF-32 is several bytes long, and for naïve English there will be padding for most of the unit; that means that the first eight bits have very good odds of being entirely zero, which means that dumping UTF-16 or UTF-32 text to a `char *` variable will give you a string littered with null bytes. That is, a UTF-16 or UTF-32 string will only be a valid C `char *` string if you are exceedingly lucky.

So we C coders have been well taken care of: UTF-8 encoded text can be stored and copied with the `char *` string type we have been using all along. As long as we are just reading in UTF-8 strings, storing them without modification, and sending them back out to the system, we have no problems. We can even get away with certain basic operations: if you know your text is delimited by an ASCII delimiter, like one of `" \t\n\r:|;,"`, you can call `strtok_r` to split it into substrings, and be confident that you aren't breaking a code point in half.

OK, so we have an internal encoding. Next time: Glib saves the day again, with functions for dealing with UTF-8 and whatever language your users speak.