by Jeffreys Copeland and Haemer

MICHAEL AVETO

"hello, world"
– Everyone

"bonjour, monde"
– Tout le monde français

"konichiwa, sekai"
– Nihonjin

I18N, Part 1

Most of the software we've written is flawed: it assumes an anglophone environment in an ASCII codeset. While it might be OK for the program we wrote to index our music CDs, it doesn't help our pen pal in Japan organize his music collection–unless all of his CDs are by English-speaking artists. And it doesn't help our college roommate from Paris either, because his recordings will have titles with odd accent marks. In fact, our software probably won't even sort the titles correctly for our French friend, because the numeric sequence of the Latin-1 codeset is not the same as the lexical sequence. Indeed, if you look above at the linguistic variations of "hello, world," you'll see that the non-English ones require characters we don't even have in ASCII. What to do?

I18N is the accepted abbreviation for internationalization–"I," followed by 18 characters, followed by "N," which neatly solves the Anglo-American dispute over the use of "Z" versus "S." The interfaces we need when we internationalize software are dictated by a few standards, principally ANSI C, the POSIX family and The Open Group's System Headers and Interfaces Guide. Some time ago, in *S/W Expert*'s now defunct sibling *RS/Magazine*, we wrote a series of articles on this topic. Recently, Copeland has found himself up to his hips in I18N development as Softway Systems prepares to have its Interix system UNIX-branded, so we thought it was time for a quick refresher.

Software in 75 Languages

It should be obvious, even in places as homogenous as Boulder, CO, that English is not the only language in the world. The process of writing software to operate in different countries involves not just different languages, but different character sets and collation orders, as well as some extensions to things like regular expressions. How do we represent this information?

First, let's recognize that there are two roughly orthogonal sets of information: There are the text strings in the program that need to be translated ("file not found") and there is culture-specific information about our current environment (use comma instead of period for a radix point and use day-month-year as the date format). The text strings are stored in *message catalogs*, and the language and region in which we want to operate are represented by something called a *locale*. We'll come back to message catalogs later, but locales actually have six separate categories of data:

- `LC_CTYPE`, containing the data used by `ctype` functions.
- `LC_COLLATE`, containing data about the lexical order of the characters, as opposed to the numeric order of their character codes.
- `LC_NUMERIC`, providing data about how to format numeric quantities.
- `LC_MONETARY`, providing rules on formatting currency.
- `LC_TIME`, containing culture-specific details about the clock.
- `LC_MESSAGES`, not directly related

to message catalogs, but telling us how to recognize yes/no responses from the user.

In general, we declare what linguistic environment we're operating in with the LANG environment variable. Generally, LANG takes the form of the language followed by the country in two-letter abbreviations, often with the codeset appended. For example, fr_CA for Canadian French or en_US.Latin1 for American English.

To make our programs locale-aware, we call setlocale() early in the program, with the locale category and locale name. Almost always, the call will be setlocale(LC_ALL,"")–that is, for all locale categories, set the current environment to the value specified by LANG.

A Matter of Character

For European languages, getting enough characters in an 8-bit byte is fairly easy: 256 characters give us plenty of room for the likes of "n with tilde," "capital a with grave accent" and so on. This is even the case for languages like Russian and Ukrainian, which aren't based on the Latin alphabet. Many languages are handled by the ISO family of 8859 codeset standards. Most Western European languages use a character set called ISO8859-1, or Latin-1. Russian can be handled by 8859-5, which contains Cyrillic characters.

Unfortunately, there are some languages for which this doesn't work. Japanese, for example, has about 6,400 ideographs. Usually, these characters are represented externally to the program–on disk, for example–as multibyte sequences. Thus, "hello, world" in Japanese might be encoded (at two bytes per character) as the hexadecimal codes 82 B1 82 F1 82 C9 82 BF 82 F1 99 A2 8D 45.

But we're still stuck with the problem of characters not fitting in a char. Fortunately, ANSI C provides a data type just for this purpose, wchar_t, the wide character. A wchar_t is at least a 16-bit quantity. Solaris defines it with a typedef unsigned long wchar_t, while Interix uses unsigned short; UNIX successor Plan 9 substitutes a 16-bit Rune data type, which contains a Unicode character; your mileage may vary. When internationalizing programs, you'll find most of the changes you make center around manipulating wchar_ts. You'll also discover that wchar_ts will figure prominently in many of the bugs you find. You'll need to include <stlib.h> in your programs to use wide characters and related interfaces.

If the data are stored externally with the characters represented as multibyte strings, how do we convert them into wide characters to deal with them internally? Well, you may not have to. If you are only processing bytes and don't need to worry about the characters themselves–think cat–it's sufficient to be 8-bit clean. What's 8-bit clean? It means your code doesn't assume a 7-bit ASCII-only environment. An example of incorrect code we've tripped over in the past involves storing strings in an array of char, but assuming the characters are only seven bits, so that the program can store a single-status bit in the high-order bit of each char in the buffer. This code won't even work in the Latin-1 character set!

In the normal case, where we have a multibyte character to convert to a wide character, we'll avail ourselves of the mbtowc() interface. This useful function takes a pointer to a multibyte character string, a pointer to the destination wide character and the maximum number of bytes that comprise a character in the multibyte string; mbtowc returns the number of bytes actually converted, which allows you to increment the string pointer for the next pass. How many chars constitute a character? For all interesting implementations of the standards, it's a minimum of one byte and a maximum of MB_CUR_MAX. While MB_CUR_MAX may look like a defined constant value, it actually returns the size of the maximum character from the current locale.

On the other hand, if you have a string to convert, you can use the mbstowcs() interface, which takes pointers to the multibyte source string and the wchar_t target string, plus a count of the maximum number of bytes to convert.

In general, when you are operating on strings–think of a filter program–the steps you want to take for an I18N program are as follows:

- Read the string of multibyte characters.
- Convert it to a string of wchar_ts.
- Process the wchar_ts using an algorithm similar to the original char-based one ("similar" is important, more about that in a moment).
- Convert the wchar_t string back to multibyte characters.
- Output the multibyte string.

There are a handful of things we need to look out for when processing wide characters. To begin with, we want to compare against characters, not bytes, in our loops. For example, to step through a string, the classic loop

```
char *s, buf[128];
for( s = buf;  *s;  s++ )
```

needs to be transformed into

```
wchar_t *s, buf[128];
for( s = buf;  *s != L'\0';  s++ )
```

where L'\0' represents the wide-character null byte. Likewise, L'a' represents the wide-character version of lowercase a. By analogy to our normal single-byte methods, when doing input we need to read wide characters into wint_ts rather than wchar_ts, so we can recognize the wide character end-of-file marker (more about this when we discuss input and output).

A major source of bugs is the fact that sizeof(char) != sizeof(wchar_t). For example, even though memset (buf,0,BUFSIZ) worked on your character array, it won't work on your wchar_t array. Instead you need to say

```
memset(buf,0,BUFSIZ*sizeof(wchar_t))
```

An important consequence of this size difference is that problems may arise in using the same algorithm for single-byte characters and wide characters. For example, it means your bitmap for each character, short bitmaps[256], may suddenly become short bitmaps[65536] (assuming your wide character is an unsigned short), which may take far

more memory than you intended. Worse, a two-dimensional array of character data that previously took 64 KB of memory will now take 4 GB. Sometimes, the larger character size means we have to rethink our algorithms.

What else won't work quite the way we expect?

Strings are one of our most common data structures because so much of what we do is text processing. As you might expect, there are many helpful routines in the standard libraries for handling wide-character strings. The most important of these is comparison.

If all you need is to test the equality of two wide-character strings, you can substitute `wcscmp()` for `strcmp()`. This will also work if all you care about is the relative numeric values of a pair of strings. However, if you are trying to compare the lexical values of "ça" and "done," you suddenly have a more complicated problem.

In the normal course of events, we'd expect the string "ça" to be less than "done." But (assuming we're operating in Latin-1) the character code for ç is 0xE7, which is much higher than the code for d, 0x64. Fortunately, there's an interface to solve the problem. To compare two wide-character strings lexically, we use `wcscoll()`, which does a magic transformation on the strings based on their lexical values.

If we're going to be doing a lot of comparisons among the same set of strings–think `sort`–we don't want to perform the lexical transformation for every single comparison. We'd like to separate that step from `wcscoll()`, so that for the comparison step we could just use the relatively cheaper `wcscmp()`. For that, we want the `wscxfrm()` interface, which takes a wide-character string and returns, in another wide-character string, the lexically transformed value used by `wcscoll()`. In other words, `wcscoll()` is just two calls to `wcsxfrm()`, followed by a call to `wcscmp()`.

Finishing Up

This gets us about halfway through our discussion of I18N techniques. We've discussed locales and wide characters, and how changing from `char`s to `wchar_t`s can make a mess of your algorithms. We also began a discussion of wide-character strings. Next time, we'll finish up talking about strings and discuss input, output, time, dates and money. We'll also discuss message catalogs once over lightly.

Until then, happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats and being a thorn in the side of his local school board.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work`.