by Jeffreys Copeland and Haemer

MICHAEL AVETO

"Once more unto the breach, dear friends."
– William Shakespeare, *Henry V,* III:1

I18N, Part 2

Once upon a time–last month, to be exact–we were jarred into discussing internationalization (often abbreviated I18N) by some work Copeland is doing at Softway Systems in preparing its Interix system for UNIX branding. We were explaining the steps you must take to enable your anglophone software to handle characters other than ASCII and languages other than English. On the one hand, it's a cookbook problem because the patterns repeat themselves. On the other hand, we may need to consider our software a little more closely because algorithms appropriate for a character set with 128 elements just won't work for one with 6,400 ideographs. This is an application of the Hawker Observation: You need to seriously rethink the algorithm you're using every time you increase your data set by two orders of magnitude.

To review quickly: In most cases, you read data into your program with each character represented by one or more bytes. You convert the multibyte characters into the standard C `wchar_t` data type using the `mbtowc()` interface. Then you can process the `wchar_t`s using similar algorithms to your existing ones, but using different interfaces.

More Character Strings

When we left off last month, we had just finished talking about collating sequences. After that complicated problem, the remaining string-processing interfaces are fairly straightforward.

For example, in the I18N environment–as long as we are sure we have a single-byte character and have called `setlocale()` to set up the language specifics–we can still use the normal macros from `<ctype.h>`. In other words, `islower (ü)` returns true even though ü isn't in the ASCII range. But we also have wide-character versions of those same macros defined in `<wctype.h>`. If we have a wide-character `wc` containing that same u-umlaut, `iswlower(wc)` will be true. Even `towlower()` and `towupper()` do the expected thing.

That still leaves us with the interfaces for handling full strings rather than single characters. Again, things work in a reasonable fashion. There are wide-character equivalents for the major string functions defined in `<string.h>`. For example, `wcscpy(a,b)` copies the `wchar_t *b` into `a` and returns the pointer to `a`.

This means that if you have a code fragment for assembling a string such as

```
char *result, *whole, *fraction;
strcpy(result, whole);
strcat(result, ".");
strcat(result, fraction);
```

you can now use the analogous

```
wchar_t *result, *whole, *fraction;
wchar_t wradix[10];
char *radix;
wcscpy(result, whole);
radix =
  localeconv()->decimal_point;
```

```
mbstowcs(wradix, radix, 10);
wcscat(result, radix);
wcscat(result, fraction);
```

Notice that the code is not exactly the same. We can't assume that a foreign language uses the same marker for a decimal point as we do in English. The locale-specific radix character is available from the locale as a multibyte string in the `lconv` structure whose pointer is returned by `localeconv()`. Notice also that we're careful about converting the multibyte string to a wide-character string before we append it to our result. (If this was in the inner loop of our program, we'd prepare the wide-character version of the radix outside the loop.)

Input and Output

It's all well and good to have strings–even strings with European or Asian characters in them–but how can we get them in and out of our programs?

Again, by analogy, we have `%lc` and `%ls` specifiers to `printf` and `scanf`. This lets us print the string we assembled above with a line such as

```
printf("%d %ls\n", n++, result);
```

Notice that `%ls` (or the equivalent `%S`) converts the wide-character string to its multibyte form before writing it. In other words, the multibyte form is the normal external one, and the wide-character version is used for internal processing only. If you think about it, this makes perfect sense. The mapping from multibyte to wide-character is implementation-dependent and may not be portable between systems. For example, while Solaris uses a 32-bit wide-character representation, Interix uses 16-bits, so if Haemer writes a document in Japanese wide-characters on his Sun server, Copeland will be unable to read it from his Interix machine. However, we can easily exchange the document using the Shift-JIS codeset, which is one of the common Japanese multibyte representations.

But how do we get Japanese (or Chinese, or Korean) characters into the computer in the first place? This varies from system to system and isn't covered in any standard. However, in general, there is something called an input method editor, or IME. In rough outline, an IME provides a way to enter a word, which can be translated into an ideograph. In the case of Japanese, we generally have a keyboard with two Shift keys. One shifts from lowercase to uppercase, and the other shifts the keyboard into "kana" mode, which allows the keys to type Japanese phonetic characters, hiragana or katakana. When we enter a word in hiragana, we are then presented with the possible Japanese kanji for that word–remember that there is a many-to-one mapping in Japanese both for words into kanji (the name "Yoshihara" may have several different renditions in kanji, for example) and kanji into words (a given kanji may have several different readings). After we've chosen the correct kanji rendition, it is entered into the file. Again, there is no standard way to do this, so your mileage will certainly vary.

There is one last vital consideration for output. The word order in a given sentence varies from language to language. For example, "yellow flower" becomes "la flor amarilla" in Spanish. The differences between English and German are equally dramatic. To solve this problem, the standard `printf()` allows the order of the arguments to be handled in variable order in the format string. For example, if we have

```
printf(fmt, month, date, who);
```

we can use an `fmt` of `"%s %d is %s's birthday"` in English to produce "April 26 is James's birthday" and `"%3$s's geburtstag ist %2$d. %1$s"` in German to generate "James's geburtstag ist 26 April." Notice that qualifiers like `2$` allow us to reorder the parameters to account for word orders in different languages.

Message Catalogs

Allowing for different format strings is very nice, but how do we provide those in the program? We use the *message catalogs* mechanism, which we'll only cover in outline here.

Catalogs are a collection of text strings that your program loads at runtime, rather than having them stored in the executable itself. Generally, we find the catalogs by using the `NLSPATH` environment variable, which tells the `catopen()` interface where to look for the catalog based on current settings for environment variables like `LANG`. Our program then looks up individual strings with the `catgets()` interface. A typical use is something like

```
printf( catgets(cat,msg_set,msg_id,"%s %d"),
    mon, day );
```

In other words, we look up a particular message in catalog `cat`, and use `"%s %d"` if we can't find it.

In many existing programs, you'll find an amazing variety of syntactic sugar to hide the complicated `catgets()` call. The most common is the `_("message")` syntax used in many of GNU's utilities.

One last warning about text strings: You should remember that plurals vary from language to language, so the familiar English fragment

```
printf("%d error%s", n, (n==1)?"":"s");
```

won't work.

Translation of text strings is a complicated process. Whole organizations with the ostensible purpose of internationalization actually spend most of their time providing translation services.

Time

How to get time values sensibly printed is a real problem, given the different names for months and days of the week, different cultural requirements for formatting dates and different numbering schemes. These problems are all subsumed by the `strftime()` interface. It takes a buffer pointer, a size, a format and a pointer to a `tm` structure, as returned (for example) by `localtime()`, and generates a formatted string into the buffer, returning the length of the result.

The format specifiers for `strftime` are more numerous and every bit as complicated as those for `printf`. However, many of the specifiers will be familiar if you've used alternate formats from the `date` command. For example,

```
strftime(buf, SZ, "%A %d %B %Y", tm);
```

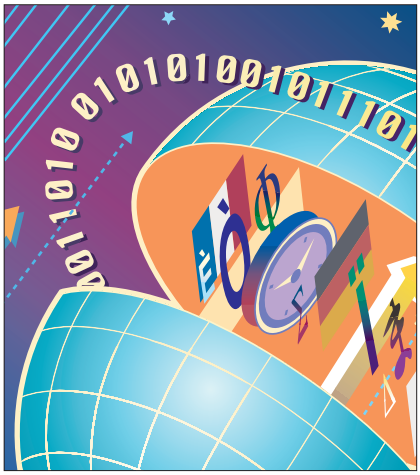
produces "Thursday 7 February 1985." The important point is that if I have `LANG` set to some locale other than POSIX, I can just as easily generate a string like "Donnerstag 7 Februari 1985."

One of the added complications for `strftime` is that to meet The Open Group's specifications it must also handle dates based on eras, the best-known example of which is the Japanese imperial date. In Japanese, the year we think of as 1985 is "Shouwa 60," or the 60th year of Hirohito's reign. To produce a date like this, we say

```
strftime(buf, SZ, "%A %d %B %EY", tm);
```

which in a Japanese locale results in "mokuyoubi 7 2gatsu 60 shouwanen," in the appropriate kanji characters. In a locale without information for era dating, the normal Gregorian year is supplied.

For the inverse problem–I have a string and I need the `tm` structure it represents–we have the `strptime()` interface. It takes as its arguments a buffer and a format specifier similar to those for `strftime`, and fills in the given `tm` with the information it is able to glean about the date from the string.



Even more amazing is `getdate()`. While it is not supplied in every system–it's a relatively recent addition to the standards–`getdate` allows us to check a variety of date format strings so that we need not exactly know the format of the date we're trying to parse a priori. The `getdate` interface performs this magic by referring to a file of possible date formats, and parsing the given buffer based on the first format it finds to match. Different applications can use different date format files based on the `DATEMSK` environment variable.

Numbers and Money

We've talked about the interfaces for character class, collation, strings, messages and time. We haven't yet talked about how we use the `LC_NUMERIC` and `LC_MONETARY` locale categories. Their data is relatively sparse and slightly overlapping.

The `LC_NUMERIC` category tells us what characters to use for the decimal point and thousands separator in the current locale. This allows us to change 1,789.456 in English to 1.789,456 in French. Our old friend `printf` already understands the decimal point, but to make it use the thousands separator, we need to use the `%'` modifier. The above number is rendered with a format like `%'.3f`.

Similarly, monetary quantities suffer from cultural variations. Many of them

are handled in the `strfmon()` interface. Like `strftime()`, `strfmon()` takes a buffer and size, a format and arguments, and returns the length of the character data placed in the buffer. The format specifier allows us to decide whether to use the local currency sign (such as a dollar sign) or the international currency name (such as USD), how many digits of pence, cents or pfennigs to include past the decimal point, whether to include thousands separators, what kind of fill characters to use for that check-like look and whether to use a minus sign or parentheses when our checkbook balance looks like the government's.

While numeric and monetary items use similar data, they use orthogonal locale categories because currency may have a different format than other numbers. Also, `strfmon()` doesn't use all the information provided in the locale.

To get the other data, like whether the minus sign precedes the currency symbol or follows it, we can use the data returned by the `localeconv()` interface, which we mentioned briefly above. It returns a pointer to an `lconv` structure that contains all of the data in the `LC_NUMERIC` and `LC_MONETARY` categories. Unless you are in a situation where you absolutely need the raw locale data, you're better off using the provided interfaces, which are (usually) more general and don't depend on the underlying data formats.

Finishing Up

We've spent two months quickly reviewing internationalization. I18N is a large can of worms, and we've just scraped off the top layer. Think of this as a checklist rather than a tutorial. For the complete story, nothing will substitute for reading the standards and manual pages. The locale chapter in The Open Group's *System Interface Definitions* is particularly useful (see `http://www.opengroup.org/publications` and click on "Common Access to the UNIX Documentation" for an online version).

As usual, we have no clue what we'll discuss next time because there's no telling what problems we'll find interesting in the meantime. But 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` *or alternately at* `ftp://ftp.expert.com/pub/Work`.