Tip 74: Use a Unicode library

Ben Klemens

27 February 2012

level: citizen of the world
purpose: think less about Unicode

In the last episode, I gave you some vocabulary about Unicode, and pointed out that a UTF-8 string can safely be stored in a C `char*` string.

Let us pause for a moment to be amazed at this achievement: all the characters of all the world's languages are all expressed using a single, unambiguous code set. I said that there's bickering about the encoding: 8 bits, 16 bits, 32 bits? Even there, we are approaching world consensus: as of this writing 68% of Web sites[1] use UTF-8. I mean, wow. Also, Mac and Linux boxes default to using UTF-8 for everything, so you can presume that an unmarked text file on a Mac or Linux box is in UTF-8.

Except a third of the world's web sites still aren't using UTF-8 at all, but are using a relatively archaic format, ISO/IEC 8859 (which has code pages, with names like Latin-1). And Windows, the free-thinking flipping-off-the-POSIX-man operating system, uses UTF-16.

So our first order of business is to convert from whatever the rest of the world dumped on us to UTF-8 so that we can use the data internally. That is, you'll need gatekeeper functions that encode incoming strings to UTF-8, and decode outgoing strings from UTF-8 to whatever the recipient wants on the other end, leaving you safe to do all internal work in one sensible encoding.

This is how Libxml[2] works: a well-formed XML document states its encoding at the header (and it has a set of rules for guessing if the encoding declaration is missing), so Libxml knows what translation to do. Libxml parses the document into an internal format, and then you query and edit that internal format. Barring errors, you are guaranteed that the internal format will be UTF-8, because Libxml doesn't want to deal with alternate encodings either.

If you have to do your own translations at the door, then you have `iconv`, which is POSIX-standard. This is going to be an unbelievably complicated function, given that there are a hundred encodings to choose from. The GNU provides a portable libiconv in case your computer doesn't have it on hand.

Glib provides a few wrappers to `iconv`, and the ones you're going to care about

[1] `http://w3techs.com/technologies/overview/character_encoding/all`
[2] `http://xmlsoft.org/encoding.html`

are `g_locale_to_utf8`[3] and `g_locale_from_utf8`. I invite you to RTFM on usage.

[International Business Machines provides a library named International Components for Unicode. I have trouble installing or recommending it.]

And while you're in the Glib, manual, you'll see a long section on Unicode manipulation tools[4]. You'll see that there are two types: those that act on UTF-8, and those that act on UTF-32 (which Glib stores via a `gunichar`).

Recall that 8 bytes is not nearly enough to express all characters in one unit, so a single character is between one and four units long. Thus, UTF-8 counts as a *multibyte encoding*, and therefore, the problems you'll have are getting the true length of the string (using a character-count or screen-width definition of *length*), getting the next full character, getting a substring, or getting a comparison for sorting purposes ("collating").

UTF-32 has enough to express any character with the same number of blocks, and so it is called a *wide character*. You'll often see reference to multibyte-to-wide conversions; this is the sort of thing they're talking about. [UTF-16 is wide enough to hold the basic language characters, so it gets treated as a wide encoding, which leads to huge problems when a multibyte character shows up.]

So once you have a single character in UTF-32 (Glib's `gunichar`), you'll have no problem doing character-content things with it, like get its type (alpha, numeric, &c), convert it to upper/lower case, et cetera.

A sidebar: if you read the C99 standard, you no doubt noticed that it includes a wide charater type, and all sorts of functions to go with. I'm not sure what they're really useful for. The width of a `wchar_t` isn't fixed by the standard, so it may mean 32-bit or 16-bit (or anything else). Compilers on Windows machines like to set it at 16-bit, to accommodate Microsoft's preference for UTF-16, but UTF-16 is still a multibyte encoding, so we need yet another type to guarantee a true wide (meaning fixed width) encoding. C11 fixes this by providing a `char16_t` and `char32_t`, but we don't have much code written around those types yet.

The sample code Here's a program to take in a file and break it into 'words,' by which I mean use `strtok_r` to break it at spaces and newlines, which are pretty universal. For each word, I use Glib to convert the first character from multibyte UTF-8 to wide character UTF-32, and then comment on whether that first character is a letter, a number, or a CJK-type wide symbol (CJK=Chinese Japanese Korean, btw).

The `file_to_string` function reads the whole input file to a string, then `localstring_to_utf8` converts it from the locale of your machine to UTF-8. The notable thing about my use of `strtok_r` is that it's just like any other. If I'm splitting at spaces and newlines, then I can guarantee you that I'm not splitting a multibyte character in half.

I output to HTML, because then I can specify UTF-8 and not worry about the encoding on the output side. If you have a UTF-16 host, open this in your browser.

[3] `http://developer.gnome.org/glib/2.31/glib-Character-Set-Conversion.html#g-locale-to-utf8`

[4] `http://developer.gnome.org/glib/2.31/glib-Unicode-Manipulation.html#gunichar`

```
/* Uses Glib, so don't forget LDADD=`pkg-config --libs glib-2.0` */

#include <glib.h>
#include <string.h> //strtok_r, strlen
#include <stdio.h>
#include <stdlib.h>
#include <locale.h> //setlocale

#define stopifnot(assertion, ...) if (!(assertion)){printf(__VA_ARGS__); exit(1)
#define Allocate(type, length) malloc(sizeof(type)*(length))
#define Reallocate(in, type, length) in = realloc(in, sizeof(type)*(length))

char *file_to_string(char *filename){
    GError *e=NULL;
    GIOChannel *f  = g_io_channel_new_file(filename, "r", &e);
    stopifnot(f, "failed to open file '%s'.\n", filename);
    char *filestring;
    stopifnot(g_io_channel_read_to_end(f, &filestring, NULL, &e) == G_IO_STATUS_
            "failed to read file.\n")
    return filestring;
}

//Frees instring for you---we can't use it for anything else.
char *localstring_to_utf8(char *instring){
    GError *e=NULL;
    setlocale(LC_ALL, ""); //get the OS's locale.
    char *out = g_locale_to_utf8(instring, -1, NULL, NULL, &e);
    free(instring); //done with the original
    stopifnot (g_utf8_validate(out, -1, NULL), "Trouble: I couldn't convert your
    return out;
}

//Get a null-terminated list of segments, broken by spaces or newlines.
char ** break_by_spaces(char *in){
    char *scratch, *result;
    char **out = Allocate(char*, 1);
    int len = 1;
    out[0] = strtok_r(in, " \n", &scratch);
    do {
        result = strtok_r(NULL, " \n", &scratch);
        Reallocate(out, char*, ++len);
        out[len-1] = result;
    } while (result);
    return out;
}
```

```
int main(int argc, char **argv){
    stopifnot(argc>1, "Please give a filename as an argument. I will print usefu

    char *ucs = localstring_to_utf8(file_to_string(argv[1]));
    FILE *out = fopen("uout.html", "w");
    fprintf(out, "<head><meta http-equiv=\"Content-Type\" content=\"text/html; c
    fprintf(out, "This document has %li characters.\n", g_utf8_strlen(ucs, -1));
    fprintf(out, "Its Unicode encoding required %i bytes.\n", strlen(ucs));
    fprintf(out, "Here it is, with each space-delimited element on a line (with
    for (char **spaced = break_by_spaces(ucs); *spaced; spaced++){
        fprintf(out, "%s", *spaced);
        gunichar c = g_utf8_get_char(*spaced);
        if (g_unichar_isalpha(c)) fprintf(out, " (a letter)");
        if (g_unichar_isdigit(c)) fprintf(out, " (a digit)");
        if (g_unichar_iswide(c)) fprintf(out, " (wide, probably CJK)");
        fprintf(out, "\n");
    }
    fclose(out);
}
```