Unicode in C and C++: What You Can Do About It Today

by Jeff Bezanson

If you write an email in Russian and send it to somebody *in Russia*, it is depressingly unlikely that he or she will be able to read it. If you write software, the burden of this sad state of affairs rests on your shoulders.

Given modern hardware resources, it is unacceptable that we can't yet routinely communicate text in different scripts or containing technical symbols. Fortunately, we are getting there.

After reading a lot on the subject and incorporating Unicode compatibility into some of my software, I decided to prepare this quick and highly pragmatic guide to digital text in the 21st century (for C programmers, of course). I don't mind adding my voice to the numerous articles that already exist on this subject, since the world needs as many programmers as possible to pick up these skills as soon as possible.

I. Encoding text

Given the variety of human languages on this planet, text is a complex subject. Many are scared away from dealing with world scripts, because they think of the numerous related software problems in the area instead of focusing on what they can actually do with their code to help.

The first thing to know is that *you do not have to worry about most problems with digital text*. The most difficult work is handled below the application layer, in OSes, UI libraries, and the C library. To give you an idea of what goes on though, here is a summary of software problems surrounding text:

- **Encoding**
  Mapping characters to numbers. *Many* such mappings exist; once you know the encoding of a piece of text, you know what character is meant by a particular number. Unicode is one such mapping, and a popular one since it incorporates more characters than any other at this time.
- **Display**
  Once you know what character is meant, you have to find a font that has the character and render it. This task is much complicated by the need to display both left-to-right and right-to-left text, the existence of combining characters that modify previous characters and have zero width, the fact that some languages require wider character cells than others, and context-sensitive letterforms.
- **Input**
  An **input method** is a way to map keystrokes (most likely several keystrokes on a typical keyboard) to characters. Input is also complicated by bidirectional text.
- **Internationalization (i18n)**
  This refers to the practice of translating a program into multiple languages, effectively by translating all of the program's strings.
- **Lexicography**
  Code that processes text as more than just binary data might searching, sorting, and modifying letter case (upper/lower) v need to perform such tasks, consider yourself lucky. If you dc UI toolkit or i18n library that already implements them.

If you are savvy with just the first issue (encoding), then OS-vendor should magically work with your program. Whether you want to or matter, and compared to proper handling of character encodings it having an unintelligible UI).

The encoding I'll talk about is called Unicode. Unicode officially encodes 1,114,112 characters, from 0x000000 to

Unicode characters. The most interesting one for C programmers is called UTF-8. UTF-8 is a "multi-byte" encoding scheme, meaning that it requires a variable number of bytes to represent a single Unicode value. Given a so-called "UTF-8 sequence", you can convert it to a Unicode value that refers to a character.

UTF-8 has the property that all existing 7-bit ASCII strings are still valid. UTF-8 only affects the meaning of bytes greater than 127, which it uses to represent higher Unicode characters. A character might require 1, 2, 3, or 4 bytes of storage depending on its value; more bytes are needed as values get larger. To store the full range of possible 32-bit characters, UTF-8 would require a whopping 6 bytes. But again, Unicode only defines characters up to 0x10FFFF, so this should never happen in practice.

UTF-8 is a specific scheme for mapping a sequence of 1-4 bytes to a number from 0x000000 to 0x10FFFF:

```
00000000 -- 0000007F:   0xxxxxxx
00000080 -- 000007FF:   110xxxxx 10xxxxxx
00000800 -- 0000FFFF:   1110xxxx 10xxxxxx 10xxxx
00010000 -- 001FFFFF:   11110xxx 10xxxxxx 10xxxx
```

The x's are bits to be extracted from the sequence and glued together to form the final number.

It is fair to say that UTF-8 is taking over the world. It is already used for filenames in Linux and is supported by all mainstream web browsers. This is not surprising considering its many nice properties:

1. It can represent all 1,114,112 Unicode characters.
2. Most C code that deals with strings on a byte-by-byte basis still works, since UTF-8 is fully compatible with 7-bit ASCII.
3. Characters usually require fewer than four bytes.
4. String sort order is preserved. In other words, sorting UTF-8 strings per-byte yields the same order as sorting them per-character by logical Unicode value.
5. A missing or corrupt byte in transmission can only affect a single character—you can always find the start of the sequence for the next character just by scanning a couple bytes.
6. There are no byte-order/endianness issues, since UTF-8 data is a byte stream.

The only price to pay for all this is that there is no longer a one-to-one correspondence between bytes and characters in a string. Finding the nth character of a string requires iterating over the string from the beginning.

See What is UTF-8? for more information about UTF-8.

Side note: Some consider UTF-8 to be discriminatory, since it allows English text to be stored efficiently at one byte per character while other world scripts require two bytes or more. This is a troublesome point, but it should not get in the way of Unicode adoption. First of all, UTF-8 was not really designed to preferentially encode English text. It was designed to preserve compatibility with the large body of existing code that scans for special characters such as line breaks, spaces, NUL terminators, and so on. Furthermore, the encoding used internally by a program has little impact on the user as long as it is able to represent their data without loss. UTF-8 is a great boon, especially for C programming. Think of it this way: if it allows you to internationalize an application that would have been difficult to convert otherwise, it is much less discriminatory than the alternative.

II. The C library

All recent implementations of the standard C library have lots of fu
Before reading up on them, it helps to know some vocabulary:

"Multibyte character" or **"multibyte string"** refers to text in o
encodings that exist throughout the world. A multibyte character d
store; the term is merely intended to be broad enough to encompas
fact only *one* such encoding; the actual encoding of user input is de
(selected as an option in a system dialog or stored as an environme
user will be in this encoding, and strings you pass to printf() are su
program can of course be in any encoding you want, but you might

"Wide character" or **"wide character string"** refers to text where each character is the same size (usually a 32-

uppercase (if applicable). strftime() can format a date and time string appropriately for the current locale, and strcoll() can do international sorting. These and other functions that depend on locale must be initialized at the beginning of your program using

```
#include <locale.h>

int main()
{
    char *locale;

    locale = setlocale(LC_ALL, "");
    ...
}
```

You don't have to do anything with the locale string returned by setlocale(), but you can use it to query your user's locale settings (more on this later).

The C library pretty much assumes you will be using multibyte strings throughout your program (since that's what you get as input). Since multibyte strings are opaque, a lot of functions beginning with "mb" are provided to deal with them. Personally, I don't like not knowing what encoding my strings use. One concrete problem with the multibyte thing is file I/O— a given file could be in any encoding, independent of locale. When you write a file or send data over a network, keeping the multibyte encoding might be a bad idea. (Even if all software uses only the proper locale-independent C library functions, and all platforms support all encodings internally, there is still no single standard for communicating the encoding of a piece of text; email messages and HTML tags do it in various ways.) You also might be able to do more efficient processing, or avoid rewriting code, if you knew the encoding your strings used.

Your encoding options

You are free to choose a string encoding for internal use in your program. The choice pretty much boils down to either UTF-8, wide (4-byte) characters, or multibyte. Each has its advantages and disadvantages:

UTF-8
- Pro: compatible with all existing strings and most existing code
- Pro: takes less space
- Pro: widely used as an interchange format (e.g. in XML)
- Con: more complex processing, O(n) string indexing

Wide characters
- Pro: easy to process
- Con: wastes space
- Pro/Con: although you can use the syntax

  ```
  L"Hello, world."
  ```

  to easily include wide-character strings in C programs, the size of wide characters is not consistent across platforms (some incorrectly use 2-byte wide characters)
- Con: should not be used for output, since spurious zero bytes and other low-ASCII characters with common meanings (such as '/' and '\n') will likely be sprinkled throughout the data.

Multibyte
- Pro: no conversions ever needed on input and output
- Pro: built-in C library support
- Pro: provides the widest possible internationalization, encodings and Unicode does not work well
- Con: strings are opaque
- Con: perpetuates incompatibilities. For example, there Russian sends data to another through your program, if his or her computer is configured for a different Rus converts to UTF-8, the text is effectively normalized so future) no matter what encoding it started in.

In this article I will advocate and give explicit instruction on using UTF-8 as an internal string encoding. Many

Below I'll outline concrete steps any C programmer could take to bring his or her code up to date with respect to text encoding. I'll also be presenting a simple C library that provides the routines you need to manipulate UTF-8.

Here's your to-do list:

1. **"char" no longer means character**
   I hereby recommend referring to character codes in C programs using a 32-bit unsigned integer type. Many platforms provide a "wchar_t" (wide character) type, but unfortunately it is to be avoided since some compilers allot it only 16 bits—not enough to represent Unicode. Wherever you need to pass around an individual character, change "char" to "unsigned int" or similar. The only remaining use for the "char" type is to mean "byte".

2. **Get UTF-8-clean**
   To take advantage of UTF-8, you'll have to treat bytes higher than 127 as perfectly ordinary characters. For example, say you have a routine that recognizes valid identifier names for a programming language. Your existing standard might be that identifiers begin with a letter:

   ```
   int valid_identifier_start(char ch)
   {
       return ((ch >= 'A' && ch <= 'Z') || (ch >= '
   }
   ```

   If you use UTF-8, you can extend this to allow letters from other languages as follows:

   ```
   int valid_identifier_start(char ch)
   {
       return ((ch >= 'A' && ch <= 'Z') || (ch >= '
               ((unsigned char)ch >= 0xC0));
   }
   ```

   A UTF-8 sequence can only start with values 0xC0 or greater, so that's what I used for checking the start of an identifier. Within an identifier, you would also want to allow characters >= 0x80, which is the range of UTF-8 continuation bytes.

   Most C string library routines still work with UTF-8, since they only scan for terminating NUL characters. A notable exception is strchr(), which in this context is more aptly named "strbyte()". Since you will be passing character codes around as 32-bit integers, you need to replace this with a routine such as my u8_strchr() that can scan UTF-8 for a given character. The traditional strchr() returns a pointer to the location of the found character, and u8_strchr() follows suit. However, you might want to know the index of the found character, and since u8_strchr() has to scan through the string anyway, it keeps a count and returns a character index as well.

   With the old strchr(), you could use pointer arithmetic to determine the character index. Now, any use of pointer arithmetic on strings is likely to be broken since characters are no longer bytes. You'll have to find and fix any code that assumes "(char*)b - (char*)a" is the number of characters between a and b (though it is still of course the number of *bytes* between a and b).

3. **Interface with your environment**
   Using UTF-8 as an internal encoding is now widespread amo
   your program runs in will not necessarily be nice enough to f

   The functions mbstowcs() and wcstombs() convert from and
   means multibyte string (i.e. the locale-specific string), and "w
   byte characters). Clearly, if you use wide characters internall
   a chance that the user's locale will be set to UTF-8 and you w
   advantage of that situation, you will have to specifically detec
   will have to convert from multibyte to wide to UTF-8.

Version 1.6 (1.5.x while in development) of the FOX toolkit uses UTF-8 internally, giving your program a nice

takes essentially the same time as iterating over bytes.

In your own code, you can use my u8_inc() and u8_dec() to move through strings. If you develop libraries or languages, be sure to expose some kind of inc() and dec() API so nobody has to move through a string by repeatedly requesting the nth character.

IV. Some UTF-8 routines

Various libraries are available for internationalization and converting between different text encodings. However, I couldn't find a straightforward set of C routines providing the minimal support needed for using UTF-8 as an internal encoding (although this functionality is often embedded in large UI toolkits and such). I decided to create a small library that could be used to bring UTF-8 to arbitrary C programs.

This library is quite incomplete; you might want to look at related FSF offerings and libutf8. libutf8 provides the multibyte and wide character C library routines mentioned above, in case your C library doesn't have them.

Since performance is sometimes a concern with UTF-8, I made my routines as fast and lightweight as possible. They perform minimal error checking— in particular, they do not bother to determine whether a sequence is valid UTF-8, which can actually be a security problem. I justify this decision by reiterating that the intention of the library is to manipulate an internal encoding; you can enforce that all strings you store in memory be valid UTF-8, enabling the library to make that assumption. Routines for validating and converting from/to UTF-8 are available free from Unicode, Inc.

Note that my routines do not need to support the many encodings of the world—the C library can handle that. If the current locale is not UTF-8, you call mbstowcs() on user input to convert any encoding (whatever it is) to a wide character string, then use my u8_toutf8() to convert it to the UTF-8 your program is comfortable with. Here's an example input routine wrapping readline():

```
char *get_utf8_input()
{
    char *line, *u8s;
    unsigned int *wcs;
    int len;

    line = readline("");
    if (locale_is_utf8) {
        return line;
    }
    else {
        len = mbstowcs(NULL, line, 0)+1;
        wcs = malloc(len * sizeof(int));
        mbstowcs(wcs, line, len);
        u8s = malloc(len * sizeof(int));
        u8_toutf8(u8s, len*sizeof(int), wcs, len
        free(line);
        free(wcs);
        return u8s;
    }
```

The first call to mbstowcs() uses the special parameter value NULL
multibyte string.

Anyway, on with the routines. They are divided into four groups:

Group 1: conversions

```
/* is c the start of a utf8 sequence? */
```

```
int u8_toucs(unsigned int *dest, int sz, char *s

/* convert UCS-4 to UTF-8
   srcsz = number of source characters, or -1 if
   sz = size of dest buffer in bytes
   returns # characters converted */
int u8_toutf8(char *dest, int sz, unsigned int *

/* single character to UTF-8 */
int u8_wc_toutf8(char *dest, wchar_t ch);
```

Note that the library uses "unsigned int" as its wide character type.
You can convert a known number of bytes, or a NUL-terminated string. The length of a UTF-8 string is often communicated as a byte count, since that's what really matters. Recall that you can usually treat a UTF-8 string like a normal C-string with N characters (where N is the number of bytes in the UTF-8 sequence), with the possibility that some characters are >127.

Group 2: moving through UTF-8 strings

```
/* character number to byte offset */
int u8_offset(char *str, int charnum);

/* byte offset to character number */
int u8_charnum(char *s, int offset);

/* return next character, updating a byte-index
unsigned int u8_nextchar(char *s, int *i);

/* move to next character */
void u8_inc(char *s, int *i);

/* move to previous character */
void u8_dec(char *s, int *i);
```

Group 3: Unicode escape sequences
In the absence of Unicode input methods, Unicode characters are often notated using special escape sequences beginning with \u or \U. \u expects up to four hexadecimal digits, and \U expects up to eight. With these routines your program can accept input and give output using such sequences if necessary.

```
/* assuming src points to the character after a
   escape sequence, storing the result in dest a
   input characters processed */
int u8_read_escape_sequence(char *src, unsigned

/* given a wide character, convert it to an ASCI
   buf, where buf is "sz" bytes. returns the num
int u8_escape_wchar(char *buf, int sz, unsigned

/* convert a string "src" containing escape sequ
int u8_unescape(char *buf, int sz, char *src);

/* convert UTF-8 "src" to ASCII with escape sequ
   if escape_quotes is nonzero, quote characters
   backslashes as well. */
```

```
int u8_escape(char *buf, int sz, char *src, int
```

Group 4: replacements for standard functions

```
/* return a pointer to the first occurrence of c
   found. character index of found character ret
char *u8_strchr(char *s, unsigned int ch, int *c

/* same as the above, but searches a buffer of a
   a NUL-terminated string. */
char *u8_memchr(char *s, unsigned int ch, size_t

/* count the number of characters in a UTF-8 str
int u8_strlen(char *s);

/* given the string returned by setlocale(), det
   locale speaks UTF-8 */
int u8_is_locale_utf8(char *locale);

/* these functions can print from UTF-8 strings.
   about locale; you can circumvent them if is_l
int u8_vprintf(char *fmt, va_list ap);
int u8_printf(char *fmt, ...);
```

utf8.c
utf8.h