Supporting Unicode in C/C++

Fran Litterio © 2018 – 2023

We review how Unicode strings are encoded into bytes. Then we show how C and C++ do not adequately support Unicode, which can cause malfunctions unless developers write Unicode-aware code.

Consider this string of characters:

I爱你ю́!

It contains five characters. The Unicode standard sometimes calls them *graphemes* or *grapheme clusters*. Unicode represents a character with one or more *code points*, which are integers with the prefix “U+”. A code point can also have optional text in parentheses, but it’s the integer that identifies the code point.

Let’s look at the code points that represent each character in the above string:

Character	Unicode Code Points
I	U+0049 (Latin capital letter I)
爱	U+611B (Han ideograph: love, like)
你	U+4F60 (Han ideograph: you, 2nd person)
Ю́	U+044E (Cyrillic small Yu ‘ю’) and U+0301 (Combining acute accent)
!	U+0021 (Exclamation mark)

Code points are written in hexadecimal. They range from U+0000 to U+10FFFF (a total of 1,114,112 code points). As of 2019, about 136,000 (12%) of them have been allocated by the Unicode Consortium, the organization that maintains and evolves the Unicode standard.

In the table above, note that the Cyrillic character ‘ю́’ is represented by two code points: U+044E (Cyrillic small Yu ‘ю’) followed by U+0301 (Combining acute accent). This is a *combined character*, which defines a new character to be the combination of other characters.

Some combined characters are represented by *three* code points, such as the Navajo vowel ‘ą́’, which is the combination of these code points:

- U+0061 (Latin small letter ‘a’)
- U+0328 (Combining ogonek)
- U+0301 (Combining acute accent)

A Unicode string is abstractly represented by a sequence of code points, but to store a string in memory, write it to persistent storage, or transmit it over a network, the code points must be transformed into bytes. This is where things get complicated.

Why can’t we simply store a Unicode string as an array of integer code point values? When Unicode was created in the late 1980s, that would have wasted valuable memory, because it requires 4 bytes to store each code point (recall that 0x10FFFF is the largest code point value). Instead, a memory efficient design was chosen.

Encoding Unicode Strings

An *encoding* algorithm converts a sequence of code points to a sequence of bytes. The corresponding *decoding* algorithm does the inverse. It's common to refer to an encoding algorithm simply as an *encoding*. Microsoft's documentation sometimes calls an encoding a *code page*.

There are many kinds of encodings: UTF-8, UTF-16LE/BE, UTF-32LE/BE, GB 18030, ANSI, ASCII, EBCDIC, etc. — but not all encodings support Unicode. Those that support Unicode generally have names that start with "UTF" (Unicode Transformation Format) followed by the number of bits in the binary elements generated by the encoding. The suffixes "LE" and "BE" identify the *endianness* of the bytes in the binary elements of the encoded string: either *little-endian* (LE) or *big-endian* (BE).

All Unicode encodings support the Universal Character Set, which includes almost every character from every human language — as well as some non-human languages.[1]

Let's look at three Unicode encodings of the above string with the bytes shown in hexadecimal and grouped under their associated character:

Encoding	**I**	**爱**	**你**	**Ю́**	**!**
UTF-8	49	E7 88 B1	E4 BD A0	F0 9F 90 A8	21
UTF-16LE	49 00	1B 61	60 4F	4E 04 01 03	21 00
UTF-32BE	00 00 00 49	00 00 61 1B	00 00 4F 60	00 00 04 4E 00 00 03 01	00 00 00 21

Depending on the encoding, the bytes corresponding to each character vary in number and value. The byte values are the output of an encoding-specific function of the code point(s) of each character. The details of these functions are beyond the scope of this document.

Code Units

Although code points are encoded into bytes, there is structure to the bytes. A code point is encoded into one or more *code units*. A code unit is one or more bytes.

Code units are the fundamental binary elements of an encoded string. Each encoding produces code units with a fixed size. For example:

- The UTF-8 code unit is *one byte*.
- The UTF-16LE code unit is *two bytes* in little-endian order.
- The UTF-32BE code unit is *four bytes* in big-endian order.[2]

Here's the above table with each code unit highlighted (note the unhighlighted gaps between code units):

Encoding	**I**	**爱**	**你**	**Ю́**	**!**
UTF-8	49	E7 88 B1	E4 BD A0	F0 9F 90 A8	21
UTF-16LE	49 00	1B 61	60 4F	4E 04 01 03	21 00
UTF-32BE	00 00 00 49	00 00 61 1B	00 00 4F 60	00 00 04 4E 00 00 03 01	00 00 00 21

UTF-8 encodes this string to 12 code units. UTF-16 and UTF-32 both encode it to 6 code units.

[1] The Unicode Roadmap plans to support Tengwar and Cirth, the written languages of the Elves and Dwarves from Tolkein's *The Lord of the Rings*. However, a proposal to support Klingon was rejected, because most Klingon is written in human alphabets.

[2] Every encoding that has a code unit wider than one byte has both a big-endian and little-endian variant.

Unicode's One-to-Many Mappings

It's important to note the one-to-many mappings here:

- Sometimes a *single* character is represented by *multiple* code points.
- Sometimes a *single* code point is encoded into *multiple* code units.
- In many encodings, a *single* code unit is *multiple* bytes.

Contrast this with pre-Unicode *single-byte encodings*, such as ASCII and ANSI (also called ISO Latin-1), in which each character encodes to exactly one byte, which is a one-to-one mapping.

ASCII uses only 7 bits in each byte, limiting it to 2^7 = 128 characters. ANSI uses all 8 bits, giving it 256 characters, but they are entirely English and western European characters (plus some symbols).

An example of the one-to-many mapping of characters to code units is the Cyrillic character 'Ю́', which is represented by two code points: U+044E and U+0301. Regardless of the encoding, 'Ю́' will always encode to at least two code units, because every code point encodes to at least one code unit.

Consider the emoji '😃', represented by code point U+1F603 (Smiling face with open mouth). The code point value U+1F603 is greater than $2^{16} – 1$ = 0xFFFF = 65,535, so encodings with one byte and two byte code units cannot encode this emoji to just one code unit. They must use multiple code units.

Let's look at three different encodings of this emoji with the code units highlighted:

Encoding	Bytes	# of Code Units	Code Unit Size
UTF-8	F0 9F 98 83	4	1 byte
UTF-16LE	3D D8 03 DE	2	2 bytes
UTF-32BE	00 01 F6 03	1	4 bytes

UTF-32BE encodes this emoji to one code unit, but it's four bytes wide. UTF-8 and UTF-16 have smaller code units, so they must encode it to multiple code units.

Suppose an application reads one of the above byte sequences from a file. If it doesn't know which encoding was used when the file was written, it cannot know which characters it has read. To mitigate this issue, some applications write a Byte Order Mark (BOM) at the start of a file to specify the encoding. Unfortunately, BOMs are not widely used. If software reads a BOM unexpectedly, it will misinterpret the data.[3]

UTF-32 is Not Widely Used

You may have noticed that UTF-32BE produces code units that have the same numeric value as the corresponding code points, as with the emoji '😃' (U+1F603) above. This happens because UTF-32 generates 32 bit code units, which are wide enough to hold any code point value from U+0000 to U+10FFFF.

Isn't this the design that was rejected back in the 1980s: storing each code point in a four byte integer? Yes. It's also the reason that UTF-32 is rarely used today. It's still considered too wasteful of memory.

Currently, the most widely used Unicode encodings are **UTF-8** (used on UNIX/Linux, Android, iOS, and the Web), **UTF-16LE** (used on Windows), and **GB 18030** (used in China).

[3] When no Byte Order Mark is present in a text file, some text editors, such as Notepad++ and Emacs, use heuristics to guess the encoding.

Obsolete Early Encodings: UCS-2, UCS-4, and UTF-7

In the 1980s, ISO/IEC and the Unicode Consortium developed UCS-2, which encoded each character to exactly one 16 bit code unit. UCS-2 supported only the most commonly used characters from Latin-derived languages (English, Spanish, French, Italian, etc.), as well as from Chinese, Japanese, and Korean.[4]

Of course, 16 bits per character was not sufficient to represent all characters, so in 1990 the IEEE introduced UCS-4, which generated 32 bit code units. This was resisted by the Unicode Consortium, because the companies in the consortium were heavily invested in UCS-2. Also, they considered UCS-4 wasteful of memory.

In 1996, UTF-16 was developed as a compromise. UCS-2 and UCS-4 are now obsolete. Microsoft dropped support for UCS-2 and UCS-4 in Windows 2000.

UTF-7 is an obsolete encoding created in 1994 for use with Multipurpose Internet Mail Extensions (MIME), a standard for encoding non-ASCII text and binary data in email messages. Despite its name, UTF-7 was not an official Unicode encoding, and it did not support the entire Universal Character Set.

Where Do Code Points Come From?

Code points are encoded into code units, but where do code points come from? Code points are generated when applications read text entered by a user. This requires cooperation between software at various levels:

- The OS kernel, which handles low-level hardware input (e.g., key presses, touch input, pen strokes, audio input, and other enabling technologies).
- System libraries that obtain text from the kernel (e.g., **user32.dll**, the C/C++ runtime libraries).
- Applications that call system libraries to obtain text (e.g., Firefox, Wordpad, Siri, etc.).

Importantly, *application software almost never sees code points*. By the time text reaches an application, it has already been encoded into code units by the OS kernel and/or system libraries.

When a hard-coded string literal appears in source code, as with **wprintf(L"Hello world\n")**, the string literal is encoded by the compiler, and its code units are embedded in the binary at compile time.

When is Text Decoded?

Text is typically decoded only when it is rendered on screen or formatted for printing. The decoding generally happens in system libraries not in applications, but applications that work with fonts and glyphs, such as document editors, Web browsers, and PDF viewers, may need to decode text.

Normalization

In some cases, Unicode represents the *same* character with *different* sequences of code points. For example, the code point sequence U+006E (Latin lowercase ‘n’) followed by U+0303 (Combining tilde ‘◌̃’) is equivalent to the single code point U+00F1 (Lowercase Spanish ‘ñ’). These are not two different characters. These are both ‘ñ’.

[4] This set of 2^{16} = 65,536 characters became the Unicode Basic Multilingual Plane (BMP), which defines the first 2^{16} code points.

Also, the Swedish character ‘Å’ is represented by *three* different code point sequences:

- U+00C5 (Latin capital A with ring above)
- U+212B (Angstrom)
- U+0041 (Latin capital letter A) and U+030A (Combining ring above)

Recall that code unit values are a function of the values of code points. So if a character can be represented by different *code points*, then the character’s *code units* can be different even when using the same encoding! Given this, how can applications reliably compare strings or search for substrings?

To address this issue, the Unicode standard defines *normalization*, an algorithm that transforms each character into a predictable and unique sequence of code points. When identical strings are normalized, and the resulting code points are encoded using the same encoding, the resulting bytes will be identical. So all we have to do is normalize our strings and use the same encoding, and we are all set. *Except …*

There are four different normalization algorithms (called normalization forms), and they produce incompatible code point sequences, so we must also normalize strings using the same normalization form.[5]

Application developers typically do not implement normalization. Normalization is implemented in system libraries that obtain text input. If applications use strings that were normalized on different operating systems, there is no guarantee the same normalization form was used for all of them.[6]

For more information about normalization, see Unicode equivalence at Wikipedia.

Case-insensitive String Comparison

Applications sometimes need to compare strings case-insensitively, which Unicode calls *case-folding*. The Unicode Character Database (UCD) defines which characters are case-insensitively equal for all languages.

Importantly, the rules defining which characters are case-insensitively equal vary between languages, even when those languages share some characters, such as English and Turkish.

For example, in English, a dotted lowercase ‘i’ (U+0069) uppercases to a dotless uppercase ‘I’ (U+0049). But in Turkish, a dotted lowercase ‘i’ (U+0069) uppercases to a *dotted* uppercase ‘İ’ (U+0130), and a *dotless* lowercase ‘ı’ (U+0131) uppercases to a dotless uppercase ‘I’ (U+0049). So in Turkish, ‘i’ is not case-insensitively equal to ‘I’, as it is in English.

In German, the lowercase ‘ß’ most commonly uppercases to the two character *digraph* “SS” and less commonly to uppercase ‘ẞ’, so the German word “Straße” (street) uppercases to “STRASSE”.[7]

In Greek, the uppercase ‘Σ’ (sigma) lowercases to ‘σ’, except at the end of word, where it lowercases to ‘ς’. For example, the uppercase Greek name “ὈΔΥΣΣΕΎΣ” (Odysseus) lowercases to “Ὀδυσσεύς”.

[5] The great thing about standards is that there are so many to choose from. :)

[6] This is the root cause of a bug in APFS (the Apple File System), that caused it to incorrectly display filenames from SMB shares.

[7] Yes, a string’s length can change when uppercased or lowercased.

When comparing Unicode strings case-insensitively, not only must the strings use the same normalization form and the same encoding, but for the comparison to be completely correct, the *language* of the strings must be the same.[8]

The topic of character case in Unicode is very complex. For more information, see these sections of the Unicode standard and supporting documents:

- Case mappings
- Caseless matching
- Character Properties, Case Mappings & Names FAQ
- The Unicode Character Database (UCD)

Sorting Text

The sort order of strings, which Unicode calls the collation order, depends on the language of the strings. Even languages which share most of the same characters, such as German and Swedish, do not sort those characters the same way. This complicates sorting strings that contain words from different languages.

Microsoft's Sorting and String Comparison page points out just a few of the complicating factors:

- In Swedish, some vowels with an accent sort after 'z' and 'Z', but in other European languages the same accented vowel sorts immediately after the non-accented vowel.
- In Hawaiian, all the vowels sort before all the non-vowels.
- Asian languages have several different sort orders depending on phonetics, radical order, number of pen strokes in an ideograph, etc.

The Unicode standard defines the Unicode collation algorithm. From the standard:

> This standard includes the Default Unicode Collation Element Table (DUCET), which is data specifying the default collation order for all Unicode characters, and the Common Locale Data Repository (CLDR) root collation element table that is based on the DUCET. This table is designed so that it can be tailored to meet the requirements of different languages and customizations.
>
> Briefly stated, the Unicode Collation Algorithm takes an input Unicode string and a Collation Element Table, containing mapping data for characters. It produces a sort key, which is an array of unsigned 16-bit integers. Two or more sort keys so produced can then be binary-compared to give the correct comparison between the strings for which they were generated.

The standard cautions that the DUCET must be tailored to obtain correct language-specific behavior:

> The Default Unicode Collation Element Table does not aim to provide precisely correct ordering for each language and script; tailoring is required for correct language handling in almost all cases. The goal is instead to have all the *other* characters, those that are not tailored, show up in a reasonable order.

Unicode string sorting is a nightmare to implement correctly. You must delegate this to APIs that do it properly. On Windows, you can use CompareStringEx to implement a sorting algorithm, such as Heapsort, Quicksort, etc.

For more information on sorting Unicode strings, see the Unicode Collation FAQ.

[8] Later, we'll see how applications can convey the language of the strings to the APIs that do case-folded string comparisons.

Using a Locale to Specify the Language of Strings

The previous sections show that string sorting APIs and case-folded comparison APIs need to know the language of the strings in order to work correctly, but a language cannot be expressed in a string's code units. Instead, applications should set the *locale* prior to performing these operations. The locale specifies the language to use for sorting and case-folded comparisons (among other things).[9]

This requires the application to know the language of the words in the strings, so it can set the locale to match — but *the vast majority of applications do not know this information*. They assume the default locale matches the language of all strings they will ever encounter.

This works most of the time, because the default locale typically matches the language used in the computer's geographic location, but when it malfunctions due to users entering text in a language that does not match the default locale, users are discouraged from using their own languages.

Unfortunately, there's no widely accepted technical solution to this issue. Graphical user interfaces typically do not allow a user to specify the language of the text they enter. There is no standard API that detects the language of a string, so software has no choice but to use the default locale.

There are no standard C or C++ runtime library APIs that do a locale-aware case-folded string comparison. On Windows, non-standard C runtime library API **_wcsicmp** compares two strings case-insensitively, but it only works *completely* correctly if standard C runtime library API **setlocale** is first called to set the locale to match the language of the words in both strings.

Alternatively, non-standard Windows C runtime API **_wcsicmp_l** performs a locale-aware case-folded string comparison without requiring a call to **setlocale**, because it takes a locale as a parameter.[10]

Do NOT perform a case-folded string comparison by uppercasing or lowercasing both strings and comparing them with a case-sensitive API, such as **wcscmp**. First, this is more work than using **_wcsicmp**, because the character uppercasing/lowercasing APIs, **towupper** and **towlower**, also require the locale to be set.

But worse, changing a string's case character-by-character does not always work correctly, because the rules depend on the words, not just the characters. For example, in German, "STRASSE" (street) lowercases to "Straße", but "SCHLOSS" (castle) lowercases to "Schloss".[11]

Ultimately, it's best to avoid uppercasing or lowercasing strings using OS or C/C++ library APIs, though some internationalization libraries support this.

[9] A locale also specifies typographic conventions for numbers, currency, and dates, and whether text flows left-to-right or right-to-left.

[10] UNIX/Linux systems provide the equivalent Posix-conforming APIs **wcscasecmp** and **wcscasecmp_l**.

[11] The initial uppercase characters in these lowercase words are not a mistake. All German nouns have an uppercase initial character, even when they do not appear at the start of a sentence, which is another reason NOT to change a string's case one character at a time: even if the locale is set correctly, APIs **towupper** and **towlower** cannot know if a character is part of a noun.

There's More than One Locale

The previous section describes "setting the locale", but actually there are many locales that coexist. Each locale affects a different set of APIs. On Windows, separate locales are maintained by the C runtime library, the C++ runtime library, the .NET runtime, and the Windows kernel, as follows:

- The C runtime library maintains a *global locale* for the current process, which can be changed with standard API **setlocale**.
    - This locale only affects APIs in the C runtime library.
    - On Windows, non-standard API **_configthreadlocale** switches between a single global locale or per-thread locales. These configurations are mutually exclusive.
    - In an application where some modules link to the *shared* C runtime library, those modules share a common C runtime global locale, but …
    - In an application where some modules link to the *static* C runtime library, each of those modules has its own C runtime global locale, because the static C runtime libraries are unaware of each other.
- The C++ runtime library maintains a global locale for the current process, which can be changed with standard method **std::locale::global**.
    - This locale only affects APIs in the C++ runtime library and, indirectly, in the C runtime library, because …
    - **std::locale::global** changes the global locale in *both* the C++ and C runtime libraries, unlike **setlocale** which only changes the C runtime library's global locale.
    - Standard C++ does not (yet) support per-thread locales.
    - The "shared vs. static" issue from the C runtime library also exists for the C++ runtime library.
- The .NET runtime maintains a locale for each managed thread, which can be changed by assigning static property **System.Globalization.CultureInfo.CurrentCulture**.
    - This locale only affects APIs in the .NET runtime.
    - The .NET runtime does not maintain a global locale for each process.
    - See .NET namespace **System.Globalization** and class **CultureInfo** for details.
- The Windows kernel maintains a locale for each thread in a process, which can be changed with system API **SetThreadLocale**.
    - This locale only affects APIs in the Windows kernel.
    - The Windows kernel does not maintain a global locale for each process.

Windows also maintains a default locale for the computer and each user account. These default locales determine the initial locales in new processes and threads. For details, see Microsoft's Locale and Culture Awareness page.

There is no API that sets all locales at the same time. Nor is there an API that sets the kernel locale at the same time as either the C, C++, or .NET locale.

Global locales are not thread-safe. Between the calls to **setlocale** and **_wcsicmp** in one thread, another thread might change the global locale. This is a race condition. If per-thread locales are not available (as in the C++ runtime library), synchronization logic must be used to prevent this race condition.

Using Unicode Strings in C/C++

In C/C++, a Unicode string is stored in an array, where each array element holds one code unit, as follows:

- A UTF-8 encoded string is stored in an array of **char**, because both a UTF-8 code unit and type **char** are one byte wide.
- On Windows, a UTF-16LE encoded string is stored in an array of **wchar_t**, because both a UTF-16 code unit and type **wchar_t** are two bytes wide.
- On UNIX/Linux systems, type **wchar_t** is four bytes wide, making it suitable for storing a UTF-32 code unit, but UTF-8 is far more widely used on UNIX/Linux.

This means a variable of type **char** or **wchar_t** contains a single code unit, which may be just a fraction of a character. On Windows, it's not possible to store every Unicode character in a **wchar_t**, because it's only 16 bits wide (2^{16} = 65,536), but there are more than 136,000 Unicode characters.

Suppose the 5-character string shown earlier (I爱你ю́!) is encoded using UTF-16LE and stored in this C/C++ array in a Windows application:

```
wchar_t str[100];  // Large enough to hold all the code units.
```

Each element of this array can contain one UTF-16LE code unit, which is two bytes wide. Here are the code unit values and characters stored in each element of this array (ignoring the null terminator):

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]
49 00	1B 61	60 4F	4E 04	01 03	21 00
I	爱	你	ю́		!

Individually, neither **str[3]** nor **str[4]** contain a character in this string, but together they contain the combined character 'ю́'. The same would be true if the 4th character were '😃', which also encodes to two UTF-16LE code units, even though it is not a combined character.

There are several problems here:

- The array contains six **wchar_t** objects (not counting the null terminator), but the string contains only *five* characters. The function call **wcslen(str)** returns **6**, because it counts code units not characters.
- The 5th character in the string ('!') is stored in **str[5]** instead of **str[4]** , which is not consistent with zero-based array indexing.
- Changing the 4th character to 'x' with **str[3] = L'x'** corrupts the string by overwriting the first half of the Cyrillic character 'ю́'. This creates a sequence of code units that do not decode into *any* character, because no language has an 'x' with an acute accent.

As well, the wide string functions in the C runtime library (**wcslen**, **wcsncpy**, etc.) operate on code units (i.e., **wchar_t** objects) not characters. For instance, this call to **wcsncpy** attempts to copy the first four characters from array **str** into array **dest**:

```
wcsncpy(dest, str, 4);
```

But it actually copies the first four code units, which contain just the first three-and-a-half characters. Only half of the 'ю́' character is copied! In fact, there is *no Unicode encoding* where copying the first four code units of a string is guaranteed to copy the first four characters.

The Windows *safe string* APIs have the same issues. This call to **StringCchLength** sets **length** to 6:

```
size_t length = 0;
StringCchLength(str, sizeof(str), &length);  // Sets length to 6!
```

UTF-8 Has Its Own Issues

UTF-8 encoded strings are stored in arrays of **char**, because the UTF-8 code unit is one byte. This means you cannot call the wide-character APIs (**wcslen**, **wcsncpy**, etc.) to manipulate UTF-8 strings, because those APIs only work with type **wchar_t**, so your code won't even compile.

If you pass UTF-8 strings to the ANSI string APIs (**strlen**, **strncpy**, etc.), your code will compile, but some APIs will function in unexpected ways, because the ANSI string APIs don't support Unicode.[12] For instance, **strcpy** works fine with UTF-8 strings, because it simply copies every code unit, but **strlen** counts bytes not Unicode characters. This is often acceptable, unless the application needs to count actual characters, such as requiring a password to contain a minimum number of characters.

UTF-8 has many of the same issues as UTF-16. For instance, UTF-8 encodes the two character string "你!" to the four byte sequence E4 BD A0 21. The first three bytes encode the '你', and the last byte encodes the '!'. If the array **char str[100]** contains that string, then:

- The call **strlen(str)** returns **4**, but the string contains just *two* characters.
- The call **strncpy(dest, str, 2)** copies the first two-thirds of the Han character '你' instead of the first two characters.
- **str[1]** does not access the '!' character. It evaluates to BD, the 2nd code unit of '你'.

The standard C runtime library does not have APIs that directly manipulate UTF-8 strings, however it has APIs that convert between UTF-8 and other encodings. These APIs require the C runtime locale to be set to a UTF-8 *locale*, like this:

```
setlocale(LC_ALL, "en_US.UTF-8");
```

Setting a locale is required because these APIs work on *multibyte strings*, a category of strings that includes more than just UTF-8 encoded strings, so there needs to be a way for the developer to specify how the strings are encoded. The C standard committee chose the current locale as the way to do this.

[12] UTF-8 was designed so that the ANSI string APIs will work correctly with UTF-8 strings *only if* the strings contain just the first 128 Unicode code points, which UTF-8 encodes to the same bytes as ASCII (i.e., the first 128 ANSI characters). So UTF-8 is compatible with ASCII and ANSI in this limited use case. No other Unicode encoding has this feature.

The Windows C runtime library provide non-standard APIs that directly manipulate multibyte and UTF-8 strings: **_mbslen**, **_mbstrlen**, **_mbscpy**, **_mbscmp**, etc., but they also require setting the C runtime locale to specify the encoding.

Later, we'll see that the standard C++ runtime library has APIs that directly support UTF-8 strings.

Multibyte Strings

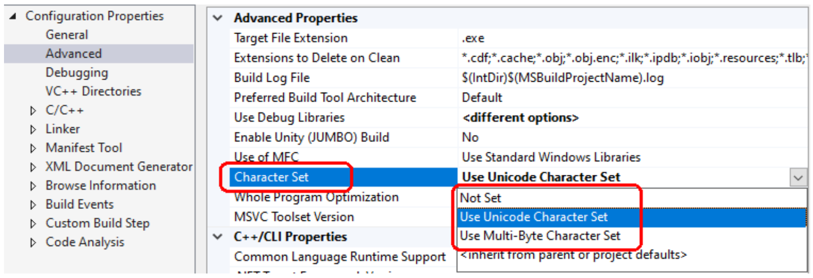
A *multibyte string* is encoded with one of a set of non-Unicode encodings (also called *code pages*), in which a single character might encode to multiple bytes. Multibyte strings were a step in the evolution from ANSI to Unicode, but there are few reasons to use multibyte strings in new code.

The C standard committee chose to implement UTF-8 support via the multibyte string APIs (see the previous section), even though UTF-8 is a Unicode encoding. But multibyte strings differ from Unicode strings in some important ways that make them difficult to use:

- No multibyte code page supports every character, so applications must programmatically switch the current code page to access various subsets of characters.
    - In contrast, *every* Unicode encoding supports the entire Universal Character Set.
    - A multibyte string cannot mix characters from different code pages (e.g., Greek and Cyrillic), because there's no way to specify *in the string* where to switch the code page.
    - As new characters are added to the Universal Character Set, multibyte encodings are not necessarily updated to support them.
- Unlike Unicode encodings, many multibyte encodings are *stateful*.
    - In a stateful encoding, the meaning of a given byte in a string depends on preceding bytes that are an arbitrary distance away.
    - This requires developers to maintain *per-string state* between calls to some multibyte string APIs, which complicates string indexing, searching, splitting, and concatenation.

Visual Studio Support for Unicode

Visual Studio C/C++ projects can be configured to use ANSI, multibyte, or Unicode APIs via the *Character Set* configuration parameter:[13]



Advanced Properties	
Target File Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*.iobj;*.resources;*.tlb;
Build Log File	$(IntDir)$(MSBuildProjectName).log
Preferred Build Tool Architecture	Default
Use Debug Libraries	**<different options>**
Enable Unity (JUMBO) Build	No
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Unicode Character Set**
Whole Program Optimization	Not Set
MSVC Toolset Version	Use Unicode Character Set
C++/CLI Properties	Use Multi-Byte Character Set
Common Language Runtime Support	<inherit from parent or project defaults>

This setting defaults to Unicode. This setting controls how the non-standard macros **_T**, **_TEXT**, and **_TCHAR** work, as well as which encodings are supported by the C runtime wrapper macros, such as **_tcslen**, **_tcscpy**, and the Windows system API wrapper macros. This allows different encodings to be used by the same source code simply by changing this project setting and re-building the application.

[13] Oddly, the ANSI setting is labeled "Not set".

For example, if the project is configured to use ANSI APIs, **_TCHAR** expands to **char**, **_T("hello")** expands to **"hello"**, **_tcslen** expands to **strlen**, and **CreateFile** expands to **CreateFileA**.

But if the project is configured to use Unicode APIs, **_TCHAR** expands to **wchar_t**, **_T("hello")** expands to **L"hello"**, **_tcslen** expands to **wcslen**, and **CreateFile** expands to **CreateFileW**.

Importantly, *you never need to use the above macros*, unless you plan to build two copies of your application: one supporting Unicode strings and one supporting ANSI strings. But nobody does this with production software, though it is common with test software. Instead, just leave the project configured for Unicode APIs, and use standard C/C++ Unicode syntax and APIs: **wchar_t**, **L"..."**, **wcslen**, **wcscpy**, etc.

One reason to use the above wrapper macros is when writing tests for pairs of APIs that support either ANSI or Unicode, such as the Windows APIs **CreateProcessA** and **CreateProcessW**. The test project can be configured at build-time to use either the ANSI APIs or the Unicode APIs with the same source code, which reduces code duplication.

The Null Character

C/C++ strings are terminated with a *null character*, which does not count towards the length of the string. The null character's code point is U+0000 (Null). Every encoding encodes the null character to a single code unit with all bits set to zero.

Let's revisit the table above that shows various encodings of the string "I爱你ю́!", highlighting the bytes that have all bits set to zero:

Encoding	**I**	**爱**	**你**	**ю́**	**!**
UTF-8	49	E7 88 B1	E4 BD A0	F0 9F 90 A8	21
UTF-16LE	49 00	1B 61	60 4F	4E 04 01 03	21 00
UTF-32BE	00 00 00 49	00 00 61 1B	00 00 4F 60	00 00 04 4E 00 00 03 01	00 00 00 21

In the UTF-16LE encoding above, there is a 00 byte in the middle of the string and also at the end, but neither one is a null character, because UTF-16 code units are two bytes wide, so the UTF-16 null character is a code unit with the value 00 00 (not shown in the above table). Similarly, in the UTF-32BE encoding, the null character is a code unit with the value 00 00 00 00.

UTF-8 code units are one byte wide, so UTF-8 encodes the null character to 00. UTF-8 is guaranteed never to generate a 00 byte in the middle of a string, because it would appear to be a null character, which would prematurely terminate the string.[14]

Note that strings can *appear* to contain an embedded null character when they actually do not. UTF-16LE encodes the 2-character string "£က" (British pound followed by Burmese letter Ka) to the byte sequence A3 00 00 10. The first two bytes encode the '£', and the last two bytes encode the 'က'.

It looks like there's a UTF-16 null character (00 00) in the middle of that string, but each 00 byte is part of a different code unit, so it's not a null character.

This means that applications must not search for the end of a UTF-16 string by scanning for two consecutive 00 bytes. Instead, it must scan for a code unit with all bytes equal to zero.

[14] This also contributes to UTF-8's partial compatibility with ANSI, because ANSI strings never have zero bytes in the middle.

Standard C++ Support for Unicode

Types **char** and **wchar_t** and the C runtime library APIs that work with them are defined by the C language standard, which means C++ supports them too. But C++ also provides the following standard classes that support ANSI and Unicode strings. Each class is an instance of template class **std::basic_string<T>**, where parameter **T** specifies the type of the code unit supported by that class.

C++ Class	Class Template Instance	Encoding
std::string	std::basic_string<char>	ANSI
std::wstring	std::basic_string<wchar_t>	UTF-16 or UTF-32 [15]
std::u8string	std::basic_string<char8_t>	UTF-8
std::u16string	std::basic_string<char16_t>	UTF-16
std::u32string	std::basic_string<char32_t>	UTF-32

All of these classes provide the same constructors, methods, and operators, because they are all instances of the same template class. Class **std::u32string** supports UTF-32 strings, which the Windows C runtime library does not support. Class **std::u16string** supports UTF-16 strings, which UNIX/Linux C runtime libraries typically do not support.

Unfortunately, these classes operate on code units not characters, just like the C string APIs. For example, method **std::basic_string<T>::length** counts code units, and **std::basic_string<T>::operator[ ]** indexes the string by code units.

But there is value to using these classes instead of pointers to **char**, **wchar_t**, **char8_t**, etc.:

- These classes handle memory allocation for the string without memory leaks.
    - When a local (i.e., stack-based) instance of one of these classes goes out of scope, the memory containing the string is automatically freed by the destructor.
    - When a new string value is assigned to an instance of one of these classes, the memory containing the old string is automatically freed.
- Instances of these classes can be referenced by the C++ smart pointer classes, **std::unique_ptr<T>** and **std::shared_ptr<T>**.
    - Example: `auto pMyString = std::make_shared<std::wstring>(L"Hello");`
    - As each **std::shared_ptr<T>** object is created, copied, and destroyed, it increments or decrements a reference count.
    - When a reference count becomes zero, the corresponding C++ object is automatically destroyed, which frees the memory containing the string. This is the C++ garbage collection mechanism.[16]

[15] Depending on the default encoding for the OS — UTF-16 on Windows, UTF-32 on UNIX/Linux.

[16] Unlike in C#/.NET, an object referenced by a C++ smart pointer is guaranteed to be garbage-collected at the moment its reference count becomes zero.

C++ provides five different string literal syntaxes, each of which implicitly converts to one of the above class types, as follows:

String Literal Syntax	Converts to Class	Encoding
`"Jerry"`	std::string	ANSI
`L"Bob"`	std::wstring	UTF-16 or UTF-32
`u8"Phil"`	std::u8string	UTF-8
`u"Bill"`	std::u16string	UTF-16
`U"Mickey"`	std::u32string	UTF-32

The same string literal prefixes can be used with character literals. For example: **L'a'**, **u8'b'**, etc.

Lastly, if your source code editor does not support entering a certain Unicode character in a string literal, both C and C++ support the character literal syntaxes **\uXXXX** and **\u00XXXXXX**, where XXXX and XXXXXX are exactly 4 or 6 hexadecimal digits (and the **\u** can optionally be written in uppercase). These are replaced by the code units for the Unicode character with the given code point value.

For example, the Euro symbol, '€', has code point U+20AC, so the character literal **u8'\u20AC'** becomes **u8'€'**.

These syntaxes have a fixed length so other text can follow without ambiguity. For instance, the string literal **u8"\u20AC10.50"** becomes **u8"€10.50"**, because it matches the first form above (**\uXXXX**).

C# Has the Same Issues, but .NET Helps

In C#, the managed types **String** and **Char** contain UTF-16LE code units — just like **wchar_t** in C/C++ on Windows. All of the above issues with C/C++ also apply to C#. For instance, C# property **String.Length** counts code units instead of characters — just like **strlen** and **wcslen** in C/C++. Also, C#'s string indexing operator, **[ ]**, has all the same issues as in C/C++.

However, .NET namespace **System.Globalization** contains classes that help. Class **StringInfo** provides method **GetTextElementEnumerator** that creates an enumerator that iterates over each character instead of each code unit. Also, class **System.Text.Encoding** provides methods to encode/decode strings in a variety of encodings. This requires extra programming, but so would using a Unicode-aware internationalization library in C/C++.

In C#, a locale-aware, case-insensitive string comparison can be done as follows:

```
var culture = new CultureInfo("ar-SA");  // Arabic/Saudi Arabia
if (String.Compare(str1, str2, culture, CompareOptions.IgnoreCase)
    ...
```

The locale used for the case-insensitive comparison is specified via the parameter of type **CultureInfo**, so the current thread's locale doesn't have to be changed. There are no standard C/C++ APIs that can do this, but the non-standard Windows API **_wcsicmp_l** can do this.

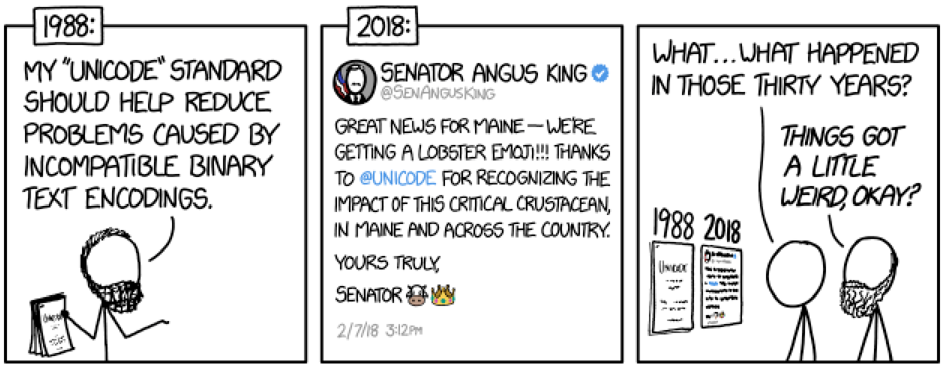
Conclusion

The right way to handle Unicode in C/C++ is to use an internationalization library, such as Boost.Locale or IBM's *International Components for Unicode* (ICU). These libraries provide APIs to normalize, encode, decode, sort, compare, convert, index, copy, search, and iterate over Unicode strings.

For more information:

- Visit the Unicode Consortium and its technical site.
- See Wikipedia's Unicode page.
- See the Unicode FAQ.
- Find code points for all characters in the Unicode character charts.
- Lookup code points of characters by entering them into the Unicode Character Inspector.

I'll give **xkcd** the final word …


1988:
MY "UNICODE" STANDARD SHOULD HELP REDUCE PROBLEMS CAUSED BY INCOMPATIBLE BINARY TEXT ENCODINGS.
2018:
SENATOR ANGUS KING
@SENANGUSKING
GREAT NEWS FOR MAINE—WE'RE GETTING A LOBSTER EMOJI!!! THANKS TO @UNICODE FOR RECOGNIZING THE IMPACT OF THIS CRITICAL CRUSTACEAN, IN MAINE AND ACROSS THE COUNTRY.
YOURS TRULY,
SENATOR 🐮👑
2/7/18 3:12PM
WHAT…WHAT HAPPENED IN THOSE THIRTY YEARS?
THINGS GOT A LITTLE WEIRD, OKAY?
1988 2018