UTF-8 Everywhere

Manifesto

1 Purpose of this document

This document contains special characters. Without proper rendering support, you may see question marks, boxes, or other symbols.

Our goal is to promote usage and support of the UTF-8 encoding and to convince that it should be the default choice of encoding for storing text strings in memory or on disk, for communication and all other uses. We believe that our approach improves performance, reduces complexity of software and helps prevent many Unicode-related bugs. We suggest that other encodings of Unicode (or text, in general) belong to rare edge-cases of optimization and should be avoided by mainstream users.

In particular, we believe that the very popular UTF-16 encoding (often mistakenly referred to as 'widechar' or simply 'Unicode' in the Windows world) has no place in library APIs except for specialized text processing libraries, e.g. ICU.

This document also recommends choosing UTF-8 for internal string representation in Windows applications, despite the fact that this standard is less popular there, both due to historical reasons and the lack of native UTF-8 support by the API. We believe that, even on this platform, the following arguments outweigh the lack of native support. Also, we recommend forgetting forever what 'ANSI codepages' are and what they were used for. It is in the user's bill of rights to mix any number of languages in any text string.

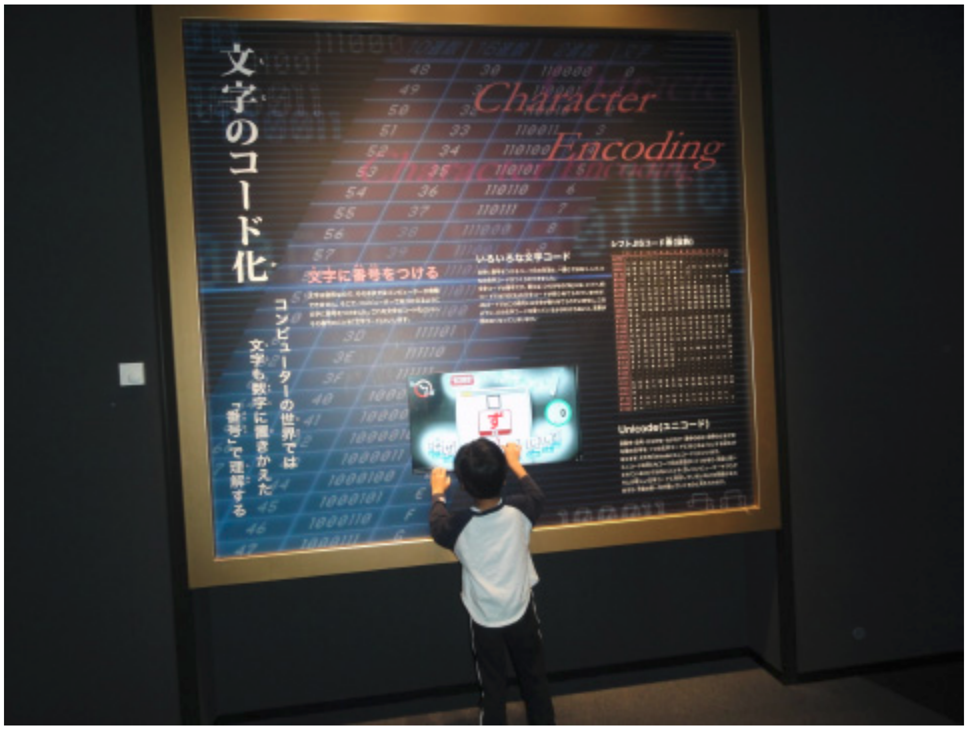
Across the industry, many localization-related bugs have been blamed on programmers' lack of knowledge in Unicode. We, however, believe that for an application that is not supposed to specialize in text, the infrastructure can and should make it possible for the program to be unaware of encoding issues. For instance, a file copy utility should not be written differently to support non-English file names. In this manifesto, we will also explain what a programmer should be doing if they do not want to dive into all complexities of Unicode and do not really care about what's inside the string.

Furthermore, we would like to suggest that counting or otherwise iterating over Unicode code points should not be seen as a particularly important task in text processing scenarios. Many developers mistakenly see code points as a kind of a successor to ASCII characters. This lead to software design decisions such as Python's string O(1) code point access. The truth, however, is that Unicode is inherently more complicated and there is no universal definition of such thing as *Unicode character*. We see no particular reason to favor Unicode code points over Unicode grapheme clusters, code units or perhaps even words in a language for that. On the other hand, seeing UTF-8 code units (bytes) as a basic unit of text seems particularly useful for many tasks, such as parsing commonly used textual data formats. This is due to a particular feature of this encoding. Graphemes, code units, code points and other relevant Unicode terms are explained in Section 5. Operations on encoded text strings are discussed in Section 7.

2 Background

In 1988, Joseph D. Becker published the first Unicode draft proposal. At the basis of his design was the naïve assumption that 16 bits per character would suffice. In 1991, the first version of the Unicode standard was published, with code points limited to 16 bits. In the following years many systems have added support for Unicode and switched to the UCS-2 encoding. It was especially attractive for new technologies, such as the Qt framework (1992), Windows NT 3.1 (1993) and Java (1995).

However, it was soon discovered that 16 bits per character will not do for Unicode. In 1996, the UTF-16 encoding was created so existing systems would be able to work with non-16-bit characters. This effectively nullified the rationale behind choosing 16-bit encoding in the first place, namely being a fixed-width encoding. Currently Unicode spans over 109449 characters, about 74500 of them being CJK ideographs.



Nagoya City Science Museum. Photo by Vadim Zlotnik.

Microsoft has often mistakenly used ‘Unicode’ and ‘widechar’ as synonyms for both ‘UCS-2’ and ‘UTF-16’. Furthermore, since UTF-8 cannot be set as the encoding for narrow string WinAPI, one must compile his code with `UNICODE` define. Windows C++ programmers are educated that Unicode must be done with ‘widechars’ (Or worse—the compiler setting-dependent TCHARs, which allow programmer to opt-out from supporting all Unicode code points). As a result of this mess, many Windows programmers are now quite confused about what is the right thing to do about text.

At the same time, in the Linux and the Web worlds, there is a silent agreement that UTF-8 is the best encoding to use for Unicode. Even though it provides shorter representation for English and therefore to computer languages (such as C++, HTML, XML, etc) over any other text, it is seldom less efficient than UTF-16 for commonly used character sets.

3 The facts

- In both UTF-8 and UTF-16 encodings, code points may take up to 4 bytes.
- UTF-8 is endianness independent. UTF-16 comes in two flavors: UTF-16LE and UTF-16BE (for the two different byte orders, respectively). Here we name them collectively as UTF-16.
- Widechar is 2 bytes in size on some platforms, 4 on others.
- UTF-8 and UTF-32 yield the same order when sorted lexicographically. UTF-16 does not.
- UTF-8 favors efficiency for English letters and other ASCII characters (one byte per character) while UTF-16 favors several Asian character sets (2 bytes instead of 3 in UTF-8). This is what made UTF-8 the favorite choice in the Web world, where English HTML/XML tags are intermixed with any-language text. Cyrillic, Hebrew and several other popular Unicode blocks are 2 bytes both in UTF-16 and UTF-8.
- UTF-16 is often misused as a fixed-width encoding, even by the Windows package programs themselves: in plain Windows edit control (until Vista), it takes two backspaces to delete a character which takes 4 bytes in UTF-16. On Windows 7, the console displays such characters as two invalid characters, regardless of the font being used.
- Many third-party libraries for Windows do not support Unicode: they accept narrow string parameters and pass them to the ANSI API. Sometimes, even for file names. In the general case, it is impossible to work around this, as a string may not be representable completely in any ANSI code page (if it contains characters from a mix of Unicode blocks). What is normally done by Windows programmers for file names is getting an 8.3 path to the file (if it already exists) and feeding it into such a library. It is not possible if the library is supposed to create a non-existing file. It is not possible if the path is very long and the 8.3 form is longer than `MAX_PATH`. It is not possible if short-name generation is disabled in OS settings.
- In C++, there is no way to return Unicode from `std::exception::what()` other than using UTF-8. There is no way to support Unicode for `localeconv` other than using UTF-8.
- UTF-16 remains popular today, even outside the Windows world. Qt, Java, C#, Python (prior to the

CPython v3.3 reference implementation, see below) and the ICU—they all use UTF-16 for internal string representation.

4 Opaque data argument

Let's go back to the file copy utility. In the UNIX world, narrow strings are considered UTF-8 by default almost everywhere. Because of that, the author of the file copy utility would not need to care about Unicode. Once tested on ASCII strings for file name arguments, it would work correctly for file names in any language, as arguments are treated as cookies. The code of the file copy utility would not need to change *at all* to support foreign languages. `fopen()` would accept Unicode seamlessly, and so would `argv`.

Now let's see how to do this on Microsoft Windows, a UTF-16 based architecture. To make a file copy utility that can accept file names in a mix of several different Unicode blocks (languages) here requires advanced trickery. First, the application must be compiled as Unicode-aware. In this case, it cannot have `main()` function with standard-C parameters. It will then accept UTF-16 encoded `argv`. To convert a Windows program written with narrow text in mind to support Unicode, one has to refactor deep and to take care of each and every string variable.

The standard library shipped with MSVC is poorly implemented with respect to Unicode support. It forwards narrow-string parameters directly to the OS ANSI API. There is no way to override this. Changing `std::locale` does not work. It's impossible to open a file with a Unicode name on MSVC using standard features of C++. The standard way to open a file is:

```
std::fstream fout("abc.txt");
```

The proper way to get around is by using Microsoft's own hack that accepts wide-string parameter, which is a non-standard extension.

On Windows, the `HKLM\SYSTEM\CurrentControlSet\Control\Nls\CodePage\ACP` registry key enables receiving non-ASCII characters, but only from a single ANSI codepage. An unimplemented value of 65001 would probably resolve the cookie issue, on Windows. If Microsoft implements support of this ACP value, this will help wider adoption of UTF-8 on Windows platform.

For Windows programmers and multi-platform library vendors, we further discuss our approach to handling text strings and refactoring programs for better Unicode support in the How to do text on Windows section.

5 Glyphs, graphemes and other Unicode species

Here is an excerpt of the definitions regarding characters, code points, code units and grapheme clusters according to the Unicode Standard with our comments. You are encouraged to refer to the relevant sections of the standard for a more detailed description.

Code point

Any numerical value in the Unicode codespace.[§3.4, D10] For instance: U+3243F.

Приве́т नमस्ते שָׁלוֹם

How many *characters* do you see?

Code unit

The minimal bit combination that can represent a unit of encoded text.[§3.9, D77] For example, UTF-8, UTF-16 and UTF-32 use 8-bit, 16-bit and 32-bit code units respectively. The above code point will be encoded as four code units ‘`f0 b2 90 bf`’ in UTF-8, two code units ‘`d889 dc3f`’ in UTF-16 and as a single code unit ‘`0003243f`’ in UTF-32. Note that these are just sequences of *groups of bits*; how they are stored on an octet-oriented media depends on the endianness of the particular encoding. When storing the above UTF-16 code units, they will be converted to ‘`d8 89 dc 3f`’ in UTF-16BE and to ‘`89 d8 3f dc`’ in UTF-16LE.

Abstract character

A unit of information used for the organization, control, or representation of textual data.[§3.4, D7] The standard further says in §3.1:

> For the Unicode Standard, [...] the repertoire is inherently open. Because Unicode is a universal encoding, any abstract character that could ever be encoded is a potential candidate to be encoded, regardless of whether the character is currently known.

The definition is indeed abstract. Whatever one can think of as a character—*is* an abstract character. For example, *tengwar letter ungwe* is an abstract character, although it is not yet representable in Unicode.

Encoded character
Coded character

A mapping between a code point and an abstract character.[§3.4, D11] For example, U+1F428 is a coded character which represents the abstract character 🐨 KOALA.

This mapping is neither total, nor injective, nor surjective:

- Surragates, noncharacters and unassigned code points do not correspond to abstract characters at all.
- Some abstract characters can be encoded by different code points; U+03A9 GREEK CAPITAL LETTER OMEGA and U+2126 OHM SIGN both correspond to the same abstract character ‘Ω’, and *must be treated identically*.
- Some abstract characters cannot be encoded by a single code point. These are represented by *sequences* of coded characters. For example, the only way to represent the abstract character ю́ *cyrillic small letter yu with acute* is by the sequence U+044E CYRILLIC SMALL LETTER YU followed by

U+0301 COMBINING ACUTE ACCENT.

Moreover, for some abstract characters, there exist representations using multiple code points, *in addition* to the single coded character form. The abstract character ǵ can be coded by the single code point U+01F5 LATIN SMALL LETTER G WITH ACUTE, or by the sequence <U+0067 LATIN SMALL LETTER G, U+0301 COMBINING ACUTE ACCENT>.

User-perceived character

Whatever the end user thinks of as a character. This notion is language dependent. For instance, 'ch' is two letters in English and Latin, but considered to be one letter in Czech and Slovak.

Grapheme cluster

A sequence of coded characters that 'should be kept together'.[§2.11] Grapheme clusters approximate the notion of user-perceived characters in a language independent way. They are used for, e.g., cursor movement and selection.

Glyph

A particular shape within a font. Fonts are collections of glyphs designed by a type designer. It's the text shaping and rendering engine responsibility to convert a sequence of code points into a sequence of glyphs within the specified font. The rules for this conversion might be complicated, locale dependent, and are beyond the scope of the Unicode standard.

'Character' may refer to any of the above. The Unicode Standard uses it as a synonym for *coded character*.[§3.4] When a programming language or a library documentation says 'character', it typically means a code unit. When an end user is asked about the number of characters in a string, he will count the user-perceived characters. A programmer might count characters as code units, code points, or grapheme clusters, according to the level of the programmer's Unicode expertise. For example, this is how Twitter counts characters. In our opinion, a string length function should not necessarily return one for the string '🐨' to be considered Unicode-compliant.

6 Asian texts: UTF-8 vs. UTF-16

So, most Unicode code points take the same number of bytes in UTF-8 and in UTF-16. This includes Russian, Hebrew, Greek and all non-BMP code points take 2 or 4 bytes in both encodings. Latin letters, together with punctuation marks and the rest of ASCII take more space in UTF-16, while some Asian characters take more in UTF-8. Couldn't Asian programmers, theoretically, object dumping UTF-16—which saves them 50% of the memory per character?

Here's reality. Saving half the memory is true only in artificially constructed examples containing only characters in the U+0800 to U+FFFF range. However, computer-to-computer text interfaces dominate over all other usages of text. This includes XML, HTTP, filesystem paths and configuration files—they all use almost exclusively ASCII characters, and in fact UTF-8 is very popular in respective Asian countries.

For a dedicated storage of Chinese books, UTF-16 may still be used as a fair optimization. As soon as the text is retrieved from such storage, it should be converted to the standard compatible with the rest of the world. In either case, if storage is at premium, a lossless compression will be used. In such cases, UTF-8 and UTF-16 will take roughly the same space. Furthermore, 'in the said languages, a glyph conveys more information than a [L]atin character so it is justified for it to take more space.' (Tronic, UTF-16 considered harmful).

Here are the results of a simple experiment. The space used by the HTML source of some web page (*Japan* article, retrieved from the Japanese Wikipedia on 2012–01–01) is shown in the first column. The second column shows the results for text with markup removed, that is 'select all, copy, paste into plain text file'.

	HTML Source (Δ UTF-8)	**Dense text (Δ UTF-8)**
UTF-8	767 KB (0%)	222 KB (0%)
UTF-16	1 186 KB (+55%)	176 KB (−21%)
UTF-8 zipped	179 KB (−77%)	83 KB (−63%)
UTF-16LE zipped	192 KB (−75%)	76 KB (−66%)
UTF-16BE zipped	194 KB (−75%)	77 KB (−65%)

As can be seen, UTF-16 takes about 50% more space than UTF-8 on real data, it only saves 20% for dense Asian text, and hardly competes with general purpose compression algorithms. The Chinese translation of this manifesto takes 58.8 KiB in UTF-16, and only 51.7 KiB in UTF-8.

7 Text operations on encoded strings

The popular text-based data formats (e.g. CSV, XML, HTML, JSON, RTF and source codes of computer programs) often contain ASCII characters as structure control elements and may contain both ASCII and non-ASCII text data strings. Working with a variable length encoding, where ASCII-inherited code points are shorter than other code points may seem like a difficult task, because encoded character boundaries within the string are not immediately known. This has driven software architects to opt for UCS-4 fixed-width encoding. (e.g. Python v3.3). In fact, this is both unnecessary and does not solve any real problem we know.

By design of this encoding, UTF-8 guarantees that an ASCII character value or a substring will never match a part of a multi-byte encoded character. The same is true for UTF-16. In both encodings, the code units of multi-part encoded code point will have MSB set to 1.

To find, say, '<' sign marking a beginning of an HTML tag, or an apostrophe (') in a UTF-8 encoded SQL statement to defend against an SQL injection, do as you would for an all-English plaintext ASCII string. The encoding guarantees this to work. Specifically, that every non-ASCII character is encoded in UTF-8 as a sequence of bytes, each of them having a value greater than 127. This leaves no place for collision for a naïve algorithm

—simple, fast and elegant, and no need to care about encoded character boundaries.

Also, you can search for a non-ASCII, UTF-8 encoded substring in a UTF-8 string as if it was a plain byte array—there is no need to mind code point boundaries. This is thanks to another design feature of UTF-8—a leading byte of an encoded code point can never hold value corresponding to one of trailing bytes of any other code point.

8 Further myths on counting characters

As we already noted, there is a popular idea that counting, splitting, indexing or otherwise iterating over code points in a Unicode string should be considered a frequent and important operation. In this section, we review this in further detail.

1. Counting characters can be done in constant time with UTF-16.

This is a common mistake by those who think that UTF-16 is a fixed-width encoding. It is not. In fact UTF-16 is a variable length encoding. Refer to this FAQ if you deny the existence of non-BMP characters.

2. Counting characters can be done in constant time with UTF-32.

This depends on the meaning of the misused word 'character'. It is true that we can count code units and code points in constant time in UTF-32. However, code points do not correspond to user-perceived characters. Even in the Unicode formalism some code points correspond to *coded character* and some to *non-characters*.

3. Counting coded characters or code points is important.

We think that the importance of code points is frequently overstated. This is due to common misunderstanding of the complexity of Unicode, which merely reflects the complexity of human languages. It is easy to tell how many characters are there in 'Abracadabra', but let's go back to the following string:

Приве́т नमस्ते שָׁלוֹם

It consists of 22 (!) code points, but only 16 grapheme clusters. It may be reduced to 20 code points if converted to NFC. Yet, the number of code points in it is irrelevant to almost any software engineering task, with perhaps the only exception of converting the string to UTF-32. For example:

- For cursor movement, text selection and alike, *grapheme clusters* shall be used.
- For limiting the length of a string in input fields, file formats, protocols, or databases, the length is measured in *code units* of some predetermined encoding. The reason is that any length limit is derived from the fixed amount of memory allocated for the string at a lower level, be it in memory, disk or in a particular data structure.

- The size of the string as it appears on the screen is unrelated to the number of code points in the string. One has to communicate with the rendering engine for this. Code points do not occupy one column even in monospace fonts and terminals. POSIX takes this into account.

4. In NFC each code point corresponds to one user-perceived character.

No, because the number of user-perceived characters that can be represented in Unicode is virtually infinite. Even in practice, most characters do not have a fully composed form. For example, the NFD string from the example above, which consists of three *real* words in three *real* languages, will consist of 20 code points in NFC. This is still far more than the 16 user-perceived characters it has.

5. The string `length()` operation must count user-perceived or coded characters. If not, it does not support Unicode properly.

Unicode support of libraries and programming languages is frequently judged by the value returned for the 'length of the string' operation. According to this evaluation of Unicode support, most popular languages, such as C#, Java, and even the ICU itself, would not support Unicode. For example, the length of the one character string '🐨' will be often reported to be 2 where UTF-16 is used as for the internal string representation and 4 for the languages that internally use UTF-8. The source of the misconception is that the specification of these languages use the word 'character' to mean a code unit, while the programmer expects it to be something else.

That said, the code unit count returned by those APIs is of the highest practical importance. When writing a UTF-8 string to a file, it is the length in bytes which is important. Counting any other type of 'characters' is, on the other hand, not very helpful.

9 Our conclusions

UTF-16 is the worst of both worlds, being both variable length and too wide. It exists only for historical reasons and creates a lot of confusion. We hope that its usage will further decline.

Portability, cross-platform interoperability and simplicity are more important than interoperability with existing platform APIs. So, the best approach is to use UTF-8 narrow strings everywhere and convert them back and forth when using platform APIs that don't support UTF-8 and accept wide strings (e.g. Windows API). Performance is seldom an issue of any relevance when dealing with string-accepting system APIs (e.g. UI code and file system APIs), and there is a great advantage to using the same encoding everywhere else in the application, so we see no sufficient reason to do otherwise.

Speaking of performance, machines often use strings to communicate (e.g. HTTP headers, XML, SOAP). Many see this as a mistake by itself, but regardless of that it is nearly always done in English and ASCII, giving UTF-8 further advantage there. Using different encodings for different kinds of strings significantly increases complexity and resulting bugs.

In particular, we believe that adding `wchar_t` to the C++ standard was a mistake, and so are the Unicode additions to C++11. What must be demanded from the implementations though, is that the *basic execution character set* would be capable of storing any Unicode data. Then, every `std::string` or `char*` parameter would be Unicode-compatible. 'If this accepts text, it should be Unicode compatible'—and with UTF-8, it is easy to achieve.

The standard facets have many design flaws. This includes `std::numpunct`, `std::moneypunct` and `std::ctype` not supporting variable-length encoded characters (non-ASCII UTF-8 and non-BMP UTF-16), or not having the information to perform the conversions. They must be fixed:

- `decimal_point()` and `thousands_sep()` shall return a string rather than a single code unit. This is how C locales do this through the `localeconv` function, albeit not customizable.
- `toupper()` and `tolower()` shall not be phrased in terms of code units, as it does not work in Unicode. For example, the Latin ﬄ ligature must be converted to FFL and the German ß to SS (there is a capital form ẞ, but the casing rules follow the traditional ones). In addition, some languages (e.g. Greek) have special final forms of some lower case letters, so case conversion routines must be aware of their position to perform the conversion correctly.

10 How to do text on Windows

This section is dedicated to developing multi-platform library development and to Windows programming. The problem with Windows platform is that it does not (yet) support Unicode-compatible narrow string system APIs. The only way to pass Unicode strings to Windows API is by converting to UTF-16 (also known as wide strings).

Note that our guidelines differ significantly from the Microsoft's original guide to Unicode conversion. Our approach based on performing the wide string conversion as close to API calls as possible, and never holding wide string data. In the previous sections we explained that this will typically result in better performance, stability, code simplicity and interoperability with other software.

- Do not use `wchar_t` or `std::wstring` in any place other than adjacent point to APIs accepting UTF-16.
- Do not use `_T("")` or `L""` literals in any place other than parameters to APIs accepting UTF-16.
- Do not use types, functions, or their derivatives that are sensitive to the `UNICODE` constant, such as `LPTSTR`, `CreateWindow()` or the `_T()` macro. Instead, use `LPWSTR`, `CreateWindowW()` and explicit `L""` literals.
- Yet, `UNICODE` and `_UNICODE` are always defined, to avoid passing narrow UTF-8 strings to ANSI WinAPI getting silently compiled. This can be done by VS project settings, under code name *Use Unicode character set*.

- `std::string` and `char*` variables are considered UTF-8, anywhere in the program.
- If you have the privilege of writing in C++, the `narrow()` / `widen()` conversion functions below can be handy for inline conversion syntax. Of course, any other UTF-8/UTF-16 coversion code would do.
- Only use Win32 functions that accept widechars (`LPWSTR`), never those which accept `LPTSTR` or `LPSTR` . Pass parameters this way:

  ```
  ::SetWindowTextW(widen(someStdString or "string litteral").c_str())
  ```

  The policy uses conversion functions described below. See also, a note on conversion performance.
- With MFC strings:

  ```
  CString someoneElse; // something that arrived from MFC.

  // Converted as soon as possible, before passing any further away from the API cal
  std::string s = str(boost::format("Hello %s\n") % narrow(someoneElse));
  AfxMessageBox(widen(s).c_str(), L"Error", MB_OK);
  ```

- For .NET developers: using the native UTF-16 based string class may be hard to avoid. Remember that this implementation detail leaks heavily through the interface of this class. For example, `string[index]` operation may return part of a character (as it would be with a UTF-8 byte array). When serializing strings into output files or communication devices, remember to specify `Encoding.UTF8` . Be ready to pay the performance penalities for conversion, e.g. in ASP.NET web applications, which typically generate UTF-8 HTML output.

Working with files, filenames and fstreams on Windows

- Always produce text output files in UTF-8.
- Using `fopen()` should anyway be avoided for RAII/OOD reasons. However, if necessary, use `_wfopen()` and WinAPI conventions as described above.
- Never pass `std::string` or `const char*` filename arguments to the `fstream` family. MSVC CRT does not support UTF-8 arguments, but it has a non-standard extension which should be used as follows:
- Convert `std::string` arguments to `std::wstring` with `widen` :

  ```
  std::ifstream ifs(widen("hello"), std::ios_base::binary);
  ```

  We will have to manually remove the conversion, when MSVC's attitude to `fstream` changes.
- This code is not multi-platform and may have to be changed manually in the future.
- Alternatively use a set of wrappers that hide the conversions.

Conversion functions

This guideline uses the conversion functions from the Boost.Nowide library (it is not yet a part of boost):

```
 std::string narrow(const wchar_t *s);
std::wstring widen(const char *s);
std::string narrow(const std::wstring &s);
std::wstring widen(const std::string &s);
```

The library also provides a set of wrappers for commonly used standard C and C++ library functions that deal with files, as well as means of reading and writing UTF-8 through iostreams.

These functions and wrappers are easy to implement using Windows' `MultiByteToWideChar` and `WideCharToMultiByte` functions. Any other (possibly faster) conversion routines can be used.

11 FAQ

1. **Q: Are you a linuxer? Is this a concealed religious fight against Windows?**

   A: No, I grew up on Windows, and I am primarily a Windows developer. I believe Microsoft made a wrong design choice in the text domain, because they did it earlier than others.

2. **Q: Are you an Anglophile? Do you secretly think English alphabet and culture are superior to any other?**

   A: No, and my country is non-ASCII speaking. I do not think that using a format which encodes ASCII characters in single byte is Anglo-centrism, or has anything to do with human interaction. Even though one can argue that source codes of programs, web pages and XML files, OS file names and other computer-to-computer text interfaces should never have existed, as long as they do exist, text is not always composed for human audiences.

3. **Q: Why do you guys care? I program in C# and/or Java and I don't need to care about encodings at all.**

   A: This is false. Both C# and Java offer a 16 bit `char` type, which is less than a Unicode character, congratulations. The .NET indexer `str[i]` works in units of the internal representation, hence a leaky abstraction once again. Substring methods will happily return an invalid string, cutting a non-BMP character in parts.

   Furthermore, you have to mind encodings when you are writing your text to files on disk, network communications, external devices, or any place for other program to read from. Please be kind to use

`System.Text.Encoding.UTF8` (.NET) in these cases, never `Encoding.ASCII`, UTF-16 or cellphone PDU, regardless of the assumptions about the contents.

Web frameworks like ASP.NET do suffer from the poor choice of internal string representation in the underlying framework: the expected string output (and input) of a web application is nearly always UTF-8, resulting in significant conversion overhead in high-throughput web applications and web services.

4. **Q: Isn't UTF-8 merely an attempt to be compatible with ASCII? Why keep this old fossil?**

   A: Regardless of whether UTF-8 was originally created as a compatibility hack, today it is a better and a more popular encoding of Unicode than any other.

5. **Q: UTF-16 characters that take more than two bytes are extremely rare in the real world. This practically makes UTF-16 a fixed-width encoding, giving it a whole bunch of advantages. Can't we just neglect these characters?**

   A: Are you serious about not supporting all of Unicode in your software design? And, if you are going to support it anyway, how does the fact that non-BMP characters are rare practically change anything, except for making software testing harder? What does matter, however, is that text manipulations are relatively rare in real applications—compared to just passing strings around as-is. This means the "almost fixed width" has little performance advantage (see Performance), while having shorter strings may be significant.

6. **Q: Why not just let any programmer use their favorite encoding internally, as long as they knows how to use it?**

   A: We have nothing against correct usage of any encoding. However, it becomes a problem when the same type, such as `std::string`, means different things in different contexts. While it is 'ANSI codepage' for some, for others, it means 'this code is broken and does not support non-English text'. In our programs, it means Unicode-aware UTF-8 string. This diversity is a source of many bugs and much misery. This additional complexity is something the world does not really need. The result is lots of Unicode-broken software, industry-wide. JoelOnSoftware suggests that having every programmer be aware of encodings is the solution to Unicode-broken software. We believe that with one mainstream encoding becoming the default for software APIs, one will be able to write a correct file copy program without being an expert in text and language issues.

7. **Q: My application is GUI-only. It does not do IP communications or file IO. Why should I convert strings back and forth all the time for Windows API calls, instead of simply using wide state variables?**

   This is a valid shortcut. Indeed, it may be a legitimate case for using wide strings. But, if you are planning to add some configuration or a log file in future, please consider converting the whole thing to narrow strings. That would be future-proof.

8. **Q: Why do you turn on the `UNICODE` define, if you do not intend to use Windows' `LPTSTR` / `TCHAR` /etc macros?**

   A: This is an additional safety mechanism against plugging a UTF-8 `char*` string into ANSI-expecting function variants of Windows API. We want it to generate a compiler error. It is the same kind of a hard-to-find bug as passing an `argv[]` string to `fopen()` on Windows: it assumes that the user will never pass non-current-codepage filenames. You will be unlikely to find this kind of a bug by manual testing, unless your testers are trained to supply Chinese file names occasionally, and yet it is a broken program logic. Thanks to the `UNICODE` define, you get a compiler error for that.

9. **Q: Isn't it quite naïve to think that Microsoft will stop using widechars one day?**

   A: Let's first see when they start supporting `CP_UTF8` as a valid codepage. This should not be very hard to do. Then, we see no reason why any Windows developer would continue using the widechar APIs. Also, adding support for `CP_UTF8` would 'unbreak' some of existing unicode-broken programs and libraries.

   Some say that adding `CP_UTF8` support would *break* existing applications that use the ANSI API, and that this was supposedly the reason why Microsoft had to resort to creating the wide string API. This is not true. Even some popular ANSI encodings are variable length (Shift JIS, for example), so no correct code would become broken. The reason Microsoft chose UCS-2 is purely historical. Back then UTF-8 hasn't yet existed, Unicode was believed to be 'just a wider ASCII', and it was considered important to use a fixed-width encoding.

10. **Q: What do you think about Byte Order Marks?**

    A: According to the Unicode Standard (v6.2, p.30): "Use of a BOM is neither required nor recommended for UTF-8".

    Byte order issues are yet another reason to avoid UTF-16. UTF-8 has no endianness issues, and the UTF-8 BOM exists only to manifest that this is a UTF-8 stream. If UTF-8 remains the only popular encoding (as it already is in the internet world), the BOM becomes redundant. In practice, most UTF-8 text files omit BOMs today.

    Using BOMs would require all existing code to be aware of them, even in simple scenarios as file concatenation. This is unacceptable.

11. **Q: What do you think about line endings?**

    A: Always use `\n (0x0a)` line endings, even on Windows. Files should be read and written in binary mode, which guarantees interoperability—a program will always give the same output on any system. Since the C and C++ standards use `\n` as in-memory line endings, this will cause all files to be written in the POSIX convention. It may cause trouble when the file is opened in Notepad on Windows; however, any decent text editor understands such line endings.

We also prefer SI units, the ISO-8601 date format, and floating point to the floating comma.

12. **Q: But what about performance of text processing algorithms, byte alignment, etc?**

    A: Is it really better with UTF-16? Maybe so. ICU uses UTF-16 for historical reasons, thus it is quite hard to measure. However, most of the times strings are treated as cookies, not sorted or reversed every second use. A denser encoding is then favorable for performance.

13. **Q: Is it really a fault of UTF-16 that people misuse it, assuming that it is 16 bits per character?**

    A: Not really. But yes, safety is an important feature of every design, and encodings are no exception.

14. **Q: If `std::string` means UTF-8, wouldn't that get confused with code that stores plain text in `std::string`s?**

    A: There is no such thing as plain text. There is no reason for storing codepage-ANSI or ASCII-only text in a class named 'string'.

15. **Q: Won't the conversions between UTF-8 and UTF-16 when passing strings to Windows slow down my application?**

    A: First, you will do *some* conversion either way. It's either when calling the system, or when interacting with the rest of the world, e.g. when sending a text string over TCP. Also, those of OS APIs which accept strings often perform tasks which are inherently slow, such as UI or file system operations. If your interaction with the system APIs dominate your application, here is a little experiment.

    One typical use of the OS APIs is to open files. This function executes in (184±3)μs on my machine:

```
void f(const wchar_t* name)
{
    HANDLE f = CreateFile(name, GENERIC_WRITE, FILE_SHARE_READ, 0, CREATE_ALWAYS,
    DWORD written;
    WriteFile(f, "Hello world!\n", 13, &written, 0);
    CloseHandle(f);
}
```

While this runs in (186±0.7)µs:

```
void f(const char* name)
{
    HANDLE f = CreateFile(widen(name).c_str(), GENERIC_WRITE, FILE_SHARE_READ, 0,
    DWORD written;
    WriteFile(f, "Hello world!\n", 13, &written, 0);
    CloseHandle(f);
}
```

(Run with `name="D:\\a\\test\\subdir\\subsubdir\\this is the sub dir\\a.txt"` in both cases. It was averaged over 5 runs. We used an optimized `widen` that relies on `std::string` contiguous storage guarantee given by C++11.)

This is just (1±2)% overhead. Also, `MultiByteToWideChar` is not the fastest UTF-8↔UTF-16 conversion function.

16. **Q: How do I write UTF-8 string literal in my C++ code?**

A: If you internationalize your software then all non-ASCII strings will be loaded from an external translation database, so it is not a problem.

If you still want to embed a special character you can do it as follows. In C++11 you can do it as:

```
u8"∃y ∀x ¬(x < y)"
```

With compilers that do not support 'u8' you can hard-code the UTF-8 code units as follows:

```
"\xE2\x88\x83y \xE2\x88\x80x \xC2\xAC(x \xE2\x89\xBA y)"
```

However the most straightforward way is to just write the string as-is and save the source file encoded in UTF-8:

```
"∃y ∀x ¬(x < y)"
```

Unfortunately, MSVC converts it to some ANSI codepage, corrupting the string. To work around this, save the file in UTF-8 *without BOM*. MSVC will assume that it is in the correct codepage and will not touch your strings. However, it renders it impossible to use Unicode identifiers and wide string literals (that you will not be using anyway).

17. **Q: I have a complex large char-based Windows application. What is the easiest way to make it Unicode-aware?**

Keep the chars. Define `UNICODE` and `_UNICODE` to get compiler errors where `narrow()` / `widen()`

should be used (this is done automatically by setting **Use Unicode Character Set** in Visual Studio project settings). Find all `fstream` and `fopen()` uses, and use wide overloads as described above. By now, you are almost done.

If you use 3rd-party libraries that do not support Unicode, e.g. forwarding file name strings as-is to `fopen()`, you will have to work around with tools such as `GetShortPathName()` as shown above.

18. **Q: What about Python? I heard they worked hard in v3.3 to support Unicode better.**

    A: Perhaps, they should have done less and the support would have been better. In the CPython v3.3 reference implementation, the internal string representation was changed. The UTF-16 was replaced by one of three possible encodings (ISO-8859-1, UCS-2 or UCS-4) depending on the actual string content. To add a single non-ASCII or non-BMP character, the entire string will often be implicitly converted to a different encoding. The internal encoding is transparent to the script. This design is meant to *optimize the performance of indexing operations* on Unicode code points. However, we argue that counting or indexing code points should not be important for the majority of uses—compared, for instance, to grapheme clusters. To our knowledge, Python currently provides no support of the latter.

    Therefore, we oppose representation-agnostic handling of strings, in favor of representation-transparent API with a UTF-8 internal representation. Indexing operations would be counting code units rather than the code points, as they in fact did before the change. This would also simplify the implementation and also improve performance, e.g. in scripts dealing with the Web, which is already dominated by UTF-8 encoded text, thus making the Python programming language more applicable in the server-side world. One may argue about the safety of string-cutting operations by script programmers, but then again, the same argument is valid for splitting grapheme clusters. Even though Unicode is now fully supported, we believe that Python, as a modern tool with less historical burden to carry, must do better job in text handling.

    Other than that, JPython and IronPython continue to rely on the less fortunate encoding used by their hosting platforms (Java and .NET, respectively) and care must be taken to handle the surrogate pairs correctly there.

19. **Q: But why `std::string`? Wouldn't it be a better, object-oriented approach to have a UTF-8 aware string class?**

    A: Not every piece of code dealing with strings is actually involved in processing and validation of text. A file copy program which receives Unicode file names and passes them to file IO routines, would do just fine with a simple byte buffer. If you design a library that accepts strings, the simple, standard and lightweight `std::string` would do just fine. On the contrary, it would be a mistake to reinvent a new string class and force everyone through your peculiar interface. Of course, if one needs more than just passing strings around, he should then use appropriate text processing tools. However, such tools are better to be independent of the storage class used, in the spirit of the container/algorithm separation in the STL. In fact, some consider `std::string` interface too bloated, as most of it was better to be moved out of the `std::string` class.

20. **Q: I already use this approach and I want to make our vision come true. What can I do?**

    A: Spread the word.

    Review your code and see what library is most painful to use in portable Unicode-aware code. Open a bug report to the authors. If you are a C or C++ library author, use `char*` and `std::string` with UTF-8 implied, and refuse to support ANSI code pages—since they are inherently Unicode-broken.

    If you are a Microsoft employee, push for implementing support of the `CP_UTF8` as one of narrow API code pages.

    Further ideas:

    - Create a repository for patches of commonly used 3rd-party libraries for UTF-8 support (e.g. PugiXML, LibTIFF, etc.) These are written in standard C and do not care about Windows.
    - Create a link-time patch for standard library functions (e.g. `fopen()`) on Windows, which would do the proper parameter conversion. This can be done for `main()` and the global environment variables.

12 About the authors

This manifesto was written by Pavel Radzivilovsky, Yakov Galka and Slava Novgorodov. It is a result of our experience and research of real-world Unicode issues and mistakes done by real-world programmers. Our goal here is to improve awareness of text issues and to inspire industry-wide changes to make Unicode-aware programming easier, ultimately improving the experience of users of those programs written by human engineers. Neither of us is involved in the Unicode consortium.

Special thanks to Glenn Linderman for providing information about Python, and to Markus Künne, Jelle Geerts, Lazy Rui and Jan Rüegg for reporting bugs and typos in this document.

Much of the text was inspired by discussions on StackOverflow initiated by Artyom Beilis, the author of Boost.Locale. Additional inspiration came from the development conventions at VisionMap and Michael Hartl's tauday.org.

13 External links

- The Unicode Consortium (The Unicode Standard, PDF)
- International Components for Unicode (ICU)
- Boost.Locale—high quality localization facilities in a C++ way.
- Should UTF-16 be considered harmful on StackOverflow, started by Artyom Beilis.
- How twitter counts characters

14 Feedback

You can leave comments/feedback on the Facebook UTF-8 Everywhere page. Your help and feedback are much appreciated.

Bitcoin donate to: 1UTF8gQmvChQ4MwUHT6XmydjUt9TsuDRn
The cash will be used for research and promotion.