Visual C++ Team Blog

Visual Studio

New Options for Managing Character Sets in the Microsoft C/C++ Compiler

★★★★★

February 22, 2016 by Jim Springfield // 32 Comments

The Microsoft C/C++ compiler has evolved along with DOS, 16-bit Windows, and 32/64-bit Windows. Its support for different characters sets, code pages, and Unicode has also changed during this time. This post will explain how our compiler has worked in the past and also cover some new switches provided by the C/C++ compiler in Visual Studio 2015 Update 2 CTP, specifically support for BOM-less UTF-8 files and controlling execution character sets. Please download this and try it out. For information on other compiler changes in Update 2, check out this post.

There are some great resources online that describe Unicode, DBCS, MBCS, code pages, and other things in great detail. I won't try to reproduce that here and will cover the basic concepts quickly. The Unicode Consortium site is a great place to learn more about Unicode.

There are two main aspects to understanding how our compiler deals with different character sets. The first is how it interprets bytes in a source file (source character set) and the second is what bytes it writes into the binary (execution character set). It is important to understand how the source code itself is encoded and stored on disk.

Explicit indication of Unicode encoding

There is a standard way to indicate Unicode files by using a BOM (byte-order mark). This BOM can indi UTF-32, UTF-16, and UTF-8, as well as whether it is big-endian or little-endian. These are indicated by t sequence of bytes that results from the encoding of the U+FEFF character into whatever encoding is be used. UTF-8 is encoded as a stream of bytes, so there isn't an actual "order" of the bytes that needs to be indicated, but the indicator for UTF-8 is still usually called a "BOM".

Implicit indication of encoding

In the early days of Windows (and DOS) before Unicode was supported, text files were stored with no indication of what encoding the file was using. It was up to the app as to how to interpret this. In DOS, any character outside of the ASCII range would be output using what was built in to the video card. In Windows,

this became known as the OEM (437) code page. This included some non-English characters as well as some line-drawing characters useful for drawing boxes around text.

Windows eventually added support for DBCS (double byte character sets) and MBCS (multi-byte character sets). There was still no standard way of indicating what the encoding of a text file was and the bytes would usually be interpreted using whatever the current code page of the system was set to. When 32bit Windows arrived, it had separate APIs for UTF-16 and another set for so-called "ANSI" APIs. These APIs took 8-bit characters that were interpreted using the current code page of the system.

Note: in Windows you cannot set the code page to a Unicode code page (either UTF-16 or UTF-8), so in many cases there is no easy way to make an older app understand a Unicode encoded file that does not have a BOM.

It is also common nowadays to encode files in UTF-8 without using a BOM. This is the default in most Linux environments. Although many Linux tools can handle a BOM, most tools won't generate one. Not having a BOM actually makes many things simpler such as concatenating files or appending to a file without having to worry about who is going to write the BOM.

How the Microsoft C/C++ compiler reads text from a file

At some point in the past, the Microsoft compiler was changed to use UTF-8 internally. So, as files are read from disk, they are converted into UTF-8 on the fly. If a file has a BOM, we use that and read the file using whatever encoding is specified and converting it to UTF-8. If the file does not have a BOM, we try to detect both little-endian and big-endian forms of UTF-16 encoding by looking at the first 8 bytes. If the file looks like UTF-16 we will treat it as if there was a UTF-16 BOM on the file.

If there is no BOM and it doesn't look like UTF-16, then we use the current code page (result of a call to GetACP) to convert the bytes on disk into UTF-8. This may or may not be correct depending on how the file was actually encoded and what characters it contains. If the file is actually encoded as UTF-8, this will never be correct as the system code page can't be set to CP_UTF8.

Execution Character Set

It is also important to understand the "execution character set". Based on the execution character set, t compiler will interpret strings differently. Let's look at a simple example to start.

```
const char ch = 'h';
const char u8ch = u8'h';
const wchar_t wch = L'h';
const char b[] = "h";
const char u8b[] = u8"h";
const wchar_t wb [] = L"h";
```

The code above will be interpreted as though you had typed this.

```
const char ch = 0x68;
const char u8ch = 0x68;
const wchar_t wch = 0x68;
const char b[] = {0x68, 0};
const char u8b[] = {0x68, 0};
const wchar_t wb [] = {0x68, 0};
```

This should make perfect sense and will be true regardless of the file encoding or current code page. Now, let's take a look at the following code.

```
const char ch = '兦';
const char u8ch = '兦';
const wchar_t wch = L'兦';
const char b[] = "兦";
const char u8b[] = u8"兦";
const wchar_t wbuffer[] = L"兦";
```

Note: I picked this character at random, but it appears to be the Han character meaning "disobedient", which seems appropriate for my purpose. It is the Unicode U+5C70 character.

We have several factors to consider in this. How is the file encoded that contains this code? And what is the current code page of the system we are compiling on? In UTF-16 the encoding is 0x5C70, in UTF-8 it is the sequence 0xE5, 0xB1, 0xB0. In the 936 code page, it is 0x8C, 0xDB. It is not representable in code page 1252 (Latin-1), which is what I'm currently running on. The 1252 code page is normally used on Windows in English and many other Western languages. Table 1 shows results for various file encodings when run on a system using code page 1252.

Table 1 – Example of results today when compiling code with various encodings.

File Encoding	UTF-8 w/ BOM	UTF-16LE w/ or w/o BOM	UTF-8 w/o BOM	DBCS (936)
Bytes in source file representing 兦	0xE5, 0xB1, 0xB0	0x70, 0x5C	0xE5, 0xB1, 0xB0	0x8C, 0xDB
Source conversion	UTF8 -> UTF8	UTF16-LE -> UTF-8	1252 -> UTF8	1252 -> UT[illegible]
Internal (UTF-8) representation	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xC3, 0xA5, 0xC2, 0xB1, 0xC2, 0xB0	0xC5, 0x92, 0x9B
Conversion to execution character set				
char ch = '兦'; UTF-8 -> CP1252	0x3F*	0x3F*	0xB0	0xDB
char u8ch = u8'兦'; UTF-8 -> UTF-8	error C2015	error C2015	error C2015	error C2015
wchar_t wch = L'兦'; UTF-8 -> UTF-16LE	0x5C70	0x5C70	0x00E5	0x0152

char b[] = "丱"; UTF-8 -> CP1252	0x3F, 0*	0x3F, 0*	0xE5, 0xB1, 0xB0, 0	0x8C, 0xDB, 0
char u8b[] = u8"丱"; UTF-8-> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xC3, 0xA5, 0xC2, 0xB1, 0xC2, 0xB0, 0	0xC5, 0x92, 0xC3, 0x9B, 0
wchar_t wb[] = L"丱"; UTF-8 -> UTF-16LE	0x5C70, 0	0x5C70, 0	0x00E5, 0x00B1, 0x00B0, 0	0x0152, 0x00DB, 0

The asterisk (*) indicates that warning C4566 was generated for these. In these cases the warning is "character represented by universal-character-name '\u5C70' cannot be represented in the current code page (1252)"
The error C2015 is "too many characters in constant"

These results probably doesn't make nearly as much sense as the simple case of the letter 'h', but I'll walk through what is going on in each case.

In columns one and two, we know what the encoding of the file is and so the conversion to the internal representation of UTF-8 is correctly 0xE5, 0xB1, 0xB0. The execution character set is Windows code page 1252, however, and when we try to convert the Unicode character U+5C70 to that code page, it fails and uses the default replacement character of 0x3F (which is the question mark). We emit warning C4566 but use the converted character of 0x3F. For the u8 character literal, we are already in UTF-8 form and don't need conversion, but we can't store three bytes in one byte and so emit error C2015. For wide literals, the "wide execution character set" is always UTF-16 and so the wide character and wide string are converted correctly. For the u8 string literal, we are already in UTF-8 form internally and no conversion is done.

In the third column (UTF-8 with no BOM), the on disk characters are 0xe5, 0xb1, and 0xb0. Each character is interpreted using the current code page of 1252 and converted to UTF-8, resulting in the internal sequence of three two-byte UTF-8 characters: (0xC3, 0xA5), (0xC2, 0xB1), and (0xC2, 0xB0). For the simple character assignment, the characters are converted back to codepage 1252, giving 0xE5, 0xB1, 0xB0. This results in a multicharacter literal and the results are the same as when the compiler encounters 'abcd'. The value of a multicharacter literal is implementation defined and in VC it is an int where each byte is from one character. When assigning to a char, you get conversion and just see the low byte. For u8 character literals we generate error C2015 when using more than one byte. Note: The compiler's treatment of multicharacter literals is very different for narrow chars and wide chars. For wide chars, we just take the first character of the multicharacter literal, which in this case is 0x00E5. In the narrow string literal, the sequence is converted back using the current code page and results in four bytes: 0xe5, 0xb1, 0xb0, 0. The u8 string literal uses the same character set as the internal representation and is 0xC3, 0xA5, 0xC2, 0xB1, 0xC2, 0xB0, 0.For a wide string literal, v UTF-16 as the execution character set which results in 0x00E5, 0x00B1, 0x00B2, 0.

Finally, in the fourth column we have the file saved using code page 936, where the character is stored on disk as 0x8C, 0xDB. We convert this using the current code page of 1252 and get two two-byte UTF-8 characters: (0xC5, 0x92), (0xC3, 0x9B). For the narrow char literal, the characters are converted back to 0x8C, 0xDB and the char gets the value of 0xDB. For the u8 char literal, the characters are not converted, but it is an error. For the wide char literal, the characters are converted to UTF-16 resulting in 0x0152, 0x00DB. The first value is used and 0x0152 is the value. For string literals, the similar conversions are done.

Changing the system code page

The results for the second and third columns will also be different if a different code page than 1252 is being used. From the descriptions above, you should be able to predict what will happen in those cases. Because of these differences, many developers will only build on systems that are set to code page 1252. For other code pages, you can get different results with no warnings or errors.

Compiler Directives

There are also two compiler directives that can impact this process. These are "#pragma setlocale" and "#pragma execution_character_set".

The setlocale pragma is documented somewhat here https://msdn.microsoft.com/en-us/library/3e22ty2t.aspx. This pragma attempts to allow a user to change the source character set for a file as it is being parsed. It appears to have been added to allow wide literals to be specified using non-Unicode files. However, there are bugs in this that effectively only allow it to be used with single-byte character sets. If you try to add a pragma set locale to the above example like this.

```
#pragma setlocale(".936")
const char buffer[] = "丌";
const wchar_t wbuffer[] = L"丌";
const char ch = '丌';
const wchar_t wch = L'丌';
```

The results are in Table 2, with the differences highlighted in Red. All it did was make more cases fail to convert and result in the 0x3F (?) character. The pragma doesn't actually change how the source file is read, instead it is used only when wide character or wide string literals are being used. When a wide literal is seen, the compiler converts individual internal UTF-8 characters back to 1252, trying to "undo" the conversion that happened when the file was read. It then converts them from the raw form to the codepage set by the "setlocale" pragma. However, in this particular case, the initial conversion to UTF-8 in column 3 and column 4 results in 3 or 2 UTF-8 characters respectively. For example, in column 4, the internal UTF-8 character of (0xC5, 0x92) is converted back to CP1252, resulting in the character 0x8C. The compiler then tries to convert that to CP936. However, 0x8C is just a lead byte, not a complete character, so the conversion fails yielding 0x3F, the default replacement character. The conversion of the second character also fails, resulting in another 0x3F. So, column three ends up with three 0x3F characters for the wide string literal and column 4 has two 0x3F characters in the literal.

For a Unicode file with a BOM, the result is the same as before, which makes sense as the encoding of t was strongly specified through the BOM.

Table 2 – Example of results today when compiling code with various encodings. Differences from Table 1 in red.

File Encoding	UTF-8 w/ BOM	UTF-16LE w/ or w/o BOM	UTF-8 w/o BOM	DBCS (936)
Bytes in source file representing 丌	0xE5, 0xB1, 0xB0	0x70, 0x5C	0xE5, 0xB1, 0xB0	0x8C, 0xDB
Source conversion	UTF8 -> UTF8	UTF16-LE -> UTF-8	1252 -> UTF8	1252 -> UTF-8

Internal (UTF-8) representation	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xC3, 0xA5, 0xC2, 0xB1, 0xC2, 0xB0	0xC5, 0x92, 0xC3, 0x9B
Conversion to execution character set				
char ch = '艹'; UTF-8 -> CP1252	0x3F*	0x3F*	0xB0	0xDB
char u8ch = u8'艹'; UTF-8 -> UTF-8	error C2015	error C2015	error C2015	error C2015
wchar_t wch = L'艹'; UTF-8 -> UTF-16LE	0x5C70	0x5C70	0x003F	0x003F
char b[] = "艹"; UTF-8 -> CP1252	0x3F, 0*	0x3F, 0*	0xE5, 0xB1, 0xB0, 0	0x8C, 0xDB, 0
char u8b[] = u8"艹"; UTF-8-> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xC3, 0xA5, 0xC2, 0xB1, 0xC2, 0xB0, 0	0xC5, 0x92, 0xC3, 0x9B, 0
wchar_t wb[] = L"艹"; UTF-8 -> UTF-16LE	0x5C70, 0	0x5C70, 0	0x003F, 0x003F, 0x003F, 0	0x003F, 0x003F, 0

The other pragma that affects all of this is #pragma execution_character_set. It takes a target execution character set, but only one value is supported and that is "utf-8". It was introduced to allow a user to specify a utf-8 execution character set and was implemented after VS2008 and VS2010 had shipped. This was done before the u8 literal prefix was supported and is really not needed any longer. At this point, we really encourage users to use the new prefixes instead of #pragma execution_character_set.

Summary of Current Issues

There are many problems with #pragma setlocale.

1. It can't be set to UTF-8, which is a major limitation.
2. It only affects string and character literals.
3. It doesn't actually work correctly with DBCS character sets.

The execution_character_set pragma lets you encode narrow strings as UTF-8, but it doesn't support ar other character set. Additionally, the only way to set this globally is to use /FI (force include) of a heade contains this pragma.

Trying to compile code that contains non ASCII strings in a cross platform way is very hard to get right.

New Options in VS2015 Update 2

In order to address these issues, there are several new compiler command-line options that allow you to specify the source character set and execution character set. The /source-charset: option can take either an IANA character set name or a Windows code page identifier (prefixed with a dot).

/source-charset:<iana-name>|.NNNN

If an IANA name is passed, that is mapped to a Windows code page using IMultiLanguage2::GetCharsetInfo. The code page is used to convert all BOM-less files that the compiler encounters to its internal UTF-8 format. If UTF-8 is specified as the source character set then no translation is performed at all since the compiler uses UTF-8 internally. If the specified name is unknown or some other error occurs retrieving information on the code page, then an error is emitted. One limitation is not being able to use UTF-7, UTF-16, or any DBCS character set that uses more than two bytes to encode a character. Also, a code page that isn't a superset of ASCII may be accepted by the compiler, but will likely cause many errors about unexpected characters.

The /source-charset option affects all files in the translation unit that are not automatically identified. (Remember that we automatically identify files with a BOM and also BOM-less UTF-16 files.) Therefore, it is not possible to have a UTF-8 encoded file and a DBCS encoded file in the same translation unit.

The /execution-charset:<iana-name>|.NNNN option uses the same lookup mechanism as /source-charset to get a code page. It controls how narrow character and string literals are generated.

There is also a /utf-8 option that is a synonym for setting "/source-charset:utf-8" and "/execution-charset:utf-8".

Note that if any of these new options are used it is now an error to use #pragma setlocale or #pragma execution-character-set. Between the new options and use of explicit u8 literals, it should no longer be necessary to use these old pragmas, especially given the bugs. However, the existing pragmas will continue to work as before if the new options are not used.

Finally, there is a new /validate-charset option, which gets turned on automatically with any of the above options. It is possible to turn this off with /validate-charset-, although that is not recommended. Previously, we would do some validation of some charsets when converting to internal UTF-8 form, however, we would do no checking of UTF-8 source files and just read them directly, which could cause subtle problems later. This switch enables validation of UTF-8 files as well regardless of whether there is a BOM or not.

Example Revisited

By correctly specifying the source-charset where needed, the results are now identical regardless of the encoding of the source file. Also, we can specify a specific execution character set that is independent of the source character set and results should be identical for a specific execution character set. In Table 3, yo see that we now get the exact same results regardless of the encoding of the source file. The data in gr indicates a change from the original example in Table 1.

Table 4 shows the results of using an execution character set of UTF-8 and Table 5 uses GB2312 as the execution character set.

Table 3 – Example using correct source-charset for each source file (current code page 1252). Green shows differences from Table 1.

File Encoding	UTF-8 w/ BOM	UTF-16LE w/ or w/o BOM	UTF-8 w/o BOM	DBCS (936)

Bytes in source file representing 丼	0xE5, 0xB1, 0xB0	0x70, 0x5C	0xE5, 0xB1, 0xB0	0x8C, 0xDB
Source conversion	UTF8 -> UTF8	UTF16-LE -> UTF-8	UTF8 -> UTF8	CP936 -> UTF-8
Internal (UTF-8) representation	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0
Conversion to execution character set				
char ch = '丼'; UTF-8 -> CP1252	0x3F*	0x3F*	0x3F*	0x3F*
char u8ch = u8'丼'; UTF-8 -> UTF-8	error C2015	error C2015	error C2015	error C2015
wchar_t wch = L'丼'; UTF-8 -> UTF-16LE	0x5C70	0x5C70	0x5C70	0x5C70
char b[] = "丼"; UTF-8 -> CP1252	0x3F, 0*	0x3F, 0*	0x3F, 0*	0x3F, 0*
char u8b[] = u8"丼"; UTF-8-> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0
wchar_t wb[] = L"丼"; UTF-8 -> UTF-16LE	0x5C70, 0	0x5C70, 0	0x5C70, 0	0x5C70, 0

Table 4 – Using an execution character set of utf-8 (code page 65001) correct /source-charset for file encoding

File Encoding	UTF-8 w/ BOM	UTF-16LE w/ or w/o BOM	UTF-8 w/o BOM	DBCS (936)
Bytes in source file representing 丼	0xE5, 0xB1, 0xB0	0x70, 0x5C	0xE5, 0xB1, 0xB0	0x8C, 0xDB
Source conversion	UTF8 -> UTF8	UTF16-LE -> UTF-8	UTF8 -> UTF8	CP936 -> U
Internal (UTF-8) representation	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1,
Conversion to execution character set				
char ch = '丼'; UTF-8 -> UTF-8	0xB0	0xB0	0xB0	0xB0
char u8ch = u8'丼'; UTF-8 -> UTF-8	error C2015	error C2015	error C2015	error C2015

wchar_t wch = L'丼'; UTF-8 -> UTF-16LE	0x5C70	0x5C70	0x5C70	0x5C70
char b[] = "丼"; UTF-8 -> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0
char u8b[] = u8"丼"; UTF-8-> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0
wchar_t wb[] = L"丼"; UTF-8 -> UTF-16LE	0x5C70, 0	0x5C70, 0	0x5C70, 0	0x5C70, 0

Table 5 – Using an execution character set of GB2312 (code page 936)

File Encoding	UTF-8 w/ BOM	UTF-16LE w/ or w/o BOM	UTF-8 w/o BOM	DBCS (936)
Bytes in source file representing 丼	0xE5, 0xB1, 0xB0	0x70, 0x5C	0xE5, 0xB1, 0xB0	0x8C, 0xDB
Source conversion	UTF8 -> UTF8	UTF16-LE -> UTF-8	UTF8 -> UTF8	CP936 -> UTF-8
Internal (UTF-8) representation	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0	0xE5, 0xB1, 0xB0
Conversion to execution character set				
char ch = '丼'; UTF-8 -> CP936	0xDB	0xDB	0xDB	0xDB
char u8ch = u8'丼'; UTF-8 -> UTF-8	error C2015	error C2015	error C2015	error C2015
wchar_t wch = L'丼'; UTF-8 -> UTF-16LE	0x5C70	0x5C70	0x5C70	0x5C70
char b[] = "丼"; UTF-8 -> CP936	0x8C, 0xDB, 0	0x8C, 0xDB, 0	0x8C, 0xDB, 0	0x8C, 0xDB,
char u8b[] = u8"丼"; UTF-8-> UTF-8	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0xB0, 0	0xE5, 0xB1, 0
wchar_t wb[] = L"丼"; UTF-8 -> UTF-16LE	0x5C70, 0	0x5C70, 0	0x5C70, 0	0x5C70, 0

Do's, Don'ts, and the Future

On Windows, save files as Unicode with a BOM when possible. This will avoid problems in many cases and most tools support reading files with a BOM.

In those cases where BOM-less UTF-8 files already exist or where changing to a BOM is a problem, use the /source-charset:utf-8 option to correctly read these files.

Don't use /source-charset with something other than utf-8 unless no other option exists. Saving files as Unicode (even BOM-less UTF8) is better than using a DBCS encoding.

Use of /execution-charset or /utf-8 can help when targeting code between Linux and Windows as Linux commonly uses BOM-less UTF-8 files and a UTF-8 execution character set.

Don't use #pragma execution_character_set. Instead, use u8 literals where needed.

Don't use #pragma setlocale. Instead, save the file as Unicode, use explicit byte sequences, or use universal character names rather than using multiple character sets in the same file.

Note: Many Windows and CRT APIs currently do not support UTF-8 encoded strings and neither the Windows code page nor CRT locale can be set to UTF-8. We are currently investigating how to improve our UTF-8 support at runtime. However, even with this limitation many applications on the Windows platform use UTF-8 encoding internally and convert to UTF-16 where necessary on Windows.

In a future major release of the compiler, we would like to change default handling of BOM-less files to assume UTF-8, but changing that in an update has the potential to cause too many silent breaking changes. Validation of UTF-8 files should catch almost all cases where that is an incorrect assumption, so my hope is that it will happen.

abc *1 year ago*

Excillent.

abc *1 year ago*

Note that in Chinese/Japanese/Korean versions of Windows Vista/7/8/8.1/10/10v1511, console codepage 65001 does not support CJK fonts. Either using ReadConsoleW/WriteConsoleW with internal buffers for UTF-8 streams (as Cygwin does) or a redistributable conhost supports CJK fonts in codepage 65001 is needed.

JoeWoodbury *1 year ago*

Just had a conversation about this today at work. This is great news.

How about making a UTF8 CRT? And (this one's a tall order) added Win32 APIs with a U8 extension, such as CreateFileU8?

Alex *1 year ago*

No need for new function names. If it was possible to set Windows code page to UTF-8, functions like CreateFile would be able to accept UTF-8 strings. After all, any ANSI string is a valid UTF-8 string too.

abc *1 year ago*

Afaik, Microsoft cannot just set Windows code page to UTF-8. Many ANSI programs uses IsDBCSLeadByte to test whether a character is two bytes or not. For encodings more than two bytes such as UTF-8, this does not work properly.

abc *1 year ago*

ANSI versions of Windows API should be used only for compatibility.
Microsoft offered a few UTF-8 versions of Windows API, like this: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682016.aspx , but not more.

Hendrik *1 y*

Actually, thats not quite true. Any ASCII string is valid UTF-8, but thats not true for ANSI.

abc *1 year ago*

I think either is a good idea. In Visual C++ .NET and Windows XP SDK, there is a UTF-16 extension for Windows 95/98/Me, which is buggy and not useful at all, called Microsoft Layer for

Unicode (MSLU) or UnicoWS. Visual Basic 6.0 and .NET Framework 2.0 also performs as a layer for Unicode on Windows 95/98/Me. As for UTF-8, only Windows APIs in Kernel32.dll is needed for implementation. Using UTF-8 in charater mode is more common since no parity check is needed, while using UTF-16 in GUIs is widely accepted because of the performance.

abc *1 year ago*

Sorry, alignment check, not parity check.

Samuel Bronson *1 year ago*

Yeah, anyone with sufficient skill should be able to implement Joe's "U8" UTF-8 variants as wrappers around the "W" functions.

My main worry is the sneaking suspicion that there might be a few "A" functions that are neither simply wrappers around the corresponding "W" functions, nor utility functions that don't really wrap anything, but in fact call into a shared internal implementation. Those could be tricky to wrap well.

Adeel *1 year ago*

Windows-wide UTF8 adaption would make perfect sense, since OSS community has proven UTF8 to be widely accepted standard.
As for .NET, this proposal is quite interesting: https://github.com/dotnet/coreclr/issues/1012.

Gregory *1 year ago*

Can you provide a compiler switch so that I can tell the compiler all my files are UTF-8 without BOM and skip the heuristics? That would prevent me from cluttering my file with compiler specific pragmas

Gregory *1 y*

Please forget my dumb comment. I'm sorry I skipped the section detailing /source-charset, /execution-charset and /utf-8 options 🙁

Trass3r *1 year ago*

One of these days...
fopen(u8"....", ...) etc.

Alexander Riccio *1 year ago*

fopen_s(&pFile, u8"....", ...)

Fixed wrong quotes & evil function for you.

Samuel Bronson *1 year ago*

And that will work on GNU/Linux, OS X, Android, FreeBSD, etc.?

(What will they come up with next, LoadLibrary_s?)

Alexander Riccio *1 year ago*

It'll work wherever libc supports Annex K, which unfortunately on Linux, the libc devs think that people should instead "learn to use the existing functions properly [sic]".

Annex K see also: https://community.rapid7.com/docs/DOC-2150

Darran Rowe *1 year ago*

The safe string functions were voted in as an annex in the C11 standard. This means they are not seen as an extension now, but as an optional component.

Alexander Riccio *1 year ago*

> We are currently investigating how to improve our UTF-8 support at runtime.

Good luck, you'll need it. The variable length nature of UTF-8 means (a) lots of ugly code, and (b) lots of opportunity for bugs.

Samuel Bronson *1 y*

Those problems are mostly irrelevant unless your code actually has reason to care about individual non-ASCII characters, no? In any case, UTF-16 has those problems too, and worse: they won't manifest nearly as often because most of the characters people use are in the BMP. But then someone comes along with a string of emoji, and ...

Alexander Riccio *1 year ago*

> But then someone comes along with a string of emoji, and ...

Don't worry, @FakeUnicode has that covered.

> Those problems are mostly irrelevant unless your code actually has reason to care about individual non-ASCII characters, no? In any case, UTF-16 has those problems too, and worse: they won't manifest nearly as often because most of the characters people use are in the BMP.

Yeah, but new code means new bugs, sadly.

Darran Rowe *1 year ago*

You are making it sound like the USA is the only country in the world that matters. Western European countries easily use Latin-1 Supplemental. If you look at languages spoken by the world, you will find that English isn't really up there at the top being the only language spoken, from sheer numbers, Chinese is easily more common than English.
If you then look at global area, you will easily find that the regions that French, Spanish and even Russian can also easily rival English.
So yes, there will be a lot of people who will have to care about that.

Tino Didriksen *1 year ago*

"On Windows, save files as Unicode with a BOM when possible. This will avoid problems in many cases and most tools support reading files with a BOM." – no! No no no…just no.

It's good that we can finally say /utf-8, but I really don't understand why it is needed. Simply always try to load all BOM-less files as UTF-8, and if it fails validation then fall back to heuristics. This is how all tools should work, so that we can finally get rid of the stupid UTF-8 BOM – only Microsoft is keeping back on this front, for no good reason.

Alex Moravcik *1 year ago*

It is mentioned in the last paragraph of the post: "In a future major release of the compiler, we would like to change default handling of BOM-less files to assume UTF-8, but changing that in an update has the potential to cause too many silent breaking changes."

Adrian *1 y*

This is great.

It would be nice if there were a way to tell the debugger visualizers to treat `std::string` and arrays of `char` as UTF-8 as well.

bames53 *1 year ago*

This is great news.

I notice an error in Table 4: the rows that show conversion from regular narrow character and string literals list the conversion being done as “UTF-8 -> CP1252” when it should be “UTF-8 -> UTF-8”.

Also I notice that the new conversion for < UTF-8>> produces a value of 0xB0. I understand why this is, and it is the same as GCC’s behavior. I think better behavior is to handle the whole UTF-8 encoding as a single _c-char_ in the C++ grammar so that this does not produce a multi-character character literal.

For reference Here’s clang’s behavior:
prog.cc:2:15: error: character too large for enclosing character literal type
char ch = ‘丼’;
^

clang 3.0 (old behavior): http://melpon.org/wandbox/permlink/zkOfu9K80OMiTYJS

clang (current behavior): http://melpon.org/wandbox/permlink/kn88gWQwWsB8P2N4

Jim Springfield *1 year ago*

Good catch. I updated Table 4 and Table 5 to show UTF-8 -> UTF-8 and UTF-8 -> CP936 respectively.

I will look at whether we can change the behavior. We already emit a similar error for a u8 character literal, so it is probably straightforward to do and given this is new it shouldn’t break existing code.

Christiano *8 months ago*

Hi, Jim,
do you are aware of ReadConsoleA bug with page code 65001?
To repeat the bug:
in cmd.exe, do this command:
chcp 65001
Run the program below:
/* ——- ReadConsoleA bug ————————*/
#include
#include
#include

int main()
{
int dwCount, x;
HANDLE hIn;

char szBuffer[1024];

```
hIn = GetStdHandle(STD_INPUT_HANDLE);
ReadConsoleA(hIn, szBuffer, 1024, &dwCount, NULL);

int j;
for(j=0;j<dwCount;j++)
printf("%02x . ",szBuffer[j]);

CloseHandle(hIn);
return 0;
}
/*————————————————*/
```

Enter "asd" input, the output will be: 61 . 73 . 64 . 0d . 0a .
Enter "á" input, the output will be: nothing.

Others developers also notice that bug:
https://www.google.com/?#q=readconsolea+65001+bug
I would like to ask: microsoft plans to fix this bug?

Thank you, I learned a lot reading the article.

Alex Moravcik *1 year ago*

It would be useful to have a Default Encoding combo box in the Options dialog.

abc *1 year ago*

I vote for this. I also need a Default Encoding option, in Visual Studio, VSCode, and even in Notepad. Wordpad have one in the Save As dialog.

BongoVR *1 year ago*

I would be interested if Microsoft also plans to enable the resource compiler to work well with UTF-8 encoded resource scripts. For using just a few special characters, we are currently forced to use UTF-16 causing problems with tools not able to digest that encoding well. It would really be a relief if could also use UTF-8 in RC files.

izogfif *11 months ago*

I'm currently trying to compile the Compact Language Detector on a machine with system locale set to "Chinese (Simplified, China)" in Microsoft Visual Studio Enterprise 2015 Version 14.0.25123.00 Update 2 and I'm having weird compiler errors. After reading your article I decided to use /utf-8 compiler command-line switch but during compilation got error. After that, I re-saved file with UTF-8 with Signature (BOM) and got different error (with /utf-8. Without /utf-8 compiler command-line option it's just a warning 4819). I believe it is a bug of the current compiler. Can you please check it and tell me what's wrong? I've posted detailed explanation on

connect.microsoft.com/VisualStudio/feedback/details/2633385/utf-8-breaks-compilation-of-c-files-saved-as-utf-8-with-bom
P.S.: I can add “u8” prefix for utf-8 string literals, but in case if there are utf-8 symbols in comments, this won’t work. I can re-save all my C++ files as UTF-16 with BOM, but they are kept up-to-date with the external sources automatically, so I can’t ask those who will compile my sources to go through each file of third-party library and re-save it as UTF-16.
