Misc lexing and string handling improvements

Document #: P2178R1
Date: 2020-07-14
Project: Programming Language C++
Audience: EWG, SG-16
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

This Omnibus paper proposes a series of small improvements to the lexing of C++ and the forming of string handling, notably to clarify the behavior of Unicode encoded sources files, and to reduce implementation divergence. This paper intends to supersede N3463 [1] and P1854R0 [3] and proposes a resolution to several core issues.

While some of these changes are unrelated, the intent of this paper's authors and SG-16 is to rewrite the description of lexing using more accurate terminology. A wording will, therefore, be provided incorporating all the desired design changes.

Updating both the design an the terminology lets us acknowledge and handle the subtilities of text which may not have been fully understood in a pre-Unicode world. The overarching goals are to reduce the divergence in implementations, the complexity of the mental model, and most importantly to make sure that the semantic of text elements is conserved through the different phases of compilation.

Revisions

R1

- Add a note about trailing whitespaces in raw string literals following a comment from the UK national body
- Add a FAQ
- Add some notes about BOM, ill-formed code units sequences, string concatenations.

Format and goals of this proposal

This proposal tries to establish a more or less exhaustive list of issues with lexing in C++. Some of these issues are too small to be worth the overhead cost of an individual paper, some may

require further exploration, many are intertwined. Providing wording for individual resolutions may lead to a hard-to-solve N-ways merge, as such we recommend first establishing the design then providing a wording that matches the design.

Is it worth it?

These issues have different severities or priorities. Few of them are critical on their own, however, it's a death-by-thousands cuts situation. It is the hope of the author than recensing them in a single paper will allow a holistic fix.

What about C?

Many of these changes could certainly benefit C. C++ seems to be in a better position to lead the charge on these changes, both because of the expertise present in SG-16 and a greater bandwidth.

Are these evolutionary?

It is unclear whether these issues are all evolutionary or merely wording defects (notably 8 and 11).

Proposals

All the proposed changes relate to phases 1-6 of translations. In particular, this proposal has no library impact.

Proposal 1: Mandating support for UTF-8 encoded source files in phase 1

The set of source file character sets is implementation-defined, which makes writing portable C++ code impossible. We proposed to mandate that C++ compilers must accept UTF-8 as an input format. Both to increase portability and to ensure that Unicode related features (ex P1949R3 [2] can be used widely. This would also allow us to better specify how Unicode encoded files are handled.

How the source file encoding is detected, and which other input formats are accepted would remain implementation-defined. Supporting UTF-8 would also not require mentioning files in the wording, the media providing the stream of UTF-8 data is not important.

Most C++ compilers (GCC, EDG, MSVC, Clang) support utf8 as one of their input format - Clang only supports utf8.

Unlike N3463 [1], we do not intend to mandate or say anything about BOMs (Byte Order Mark), following Unicode recommendations. BOMs should be ignored (but using BOMs as a means to detect UTF-8 files is a valid implementation strategy, which is notably used by MSVC). Indeed, we should give enough freedom to implementers to handle the cases where a BOM contradicts a compiler flag. Web browsers for example found a BOM to not be a reliable

mechanism. There are further issues with BOMs and toolings, such that they may be removed by IDEs or tooling. They are also widely used, and different companies have different policies.

In any case, mandating BOM recognition goes against Unicode recommendations[1]. But it is a widespread practice. and as such we recommend neither mandating nor precluding any behavior.

We do **not** propose to make UTF-8 source file a mandated default, nor the only supported format. Just that there must be some implementation-defined mechanism (such as a compiler flag) that would tell the compiler to read the file as UTF-8.

Proposal 2: What is a whitespace or a new-line?

We propose to specify that the following code point sequences are line-terminators (after phase 1):

```
LF:    Line Feed, U+000A
VT:    Vertical Tab, U+000B
FF:    Form Feed, U+000C
CR:    Carriage Return, U+000D
CR+LF: CR (U+000D) followed by LF (U+000A)
NEL:   Next Line, U+0085
LS:    Line Separator, U+2028
PS:    Paragraph Separator, U+2029
```

Line terminators and the following characters constitute whitespaces:

```
U+0009 HORIZONTAL TAB
U+0020 SPACE
U+200E LEFT-TO-RIGHT MARK
U+200F RIGHT-TO-LEFT MARK
```

These correspond to characters with the **Pattern_Whitespace** Unicode property. The line terminator subset is derived from **UAX14 - UNICODE LINE BREAKING ALGORITHM**.

We intend to fix CWG1655 [7] following this clarification.

Proposal 3: Preserve Normalization forms

We propose to specify that Unicode encodes files are not normalized in phase 1 or phase 5, as to preserve the integrity of string literals when both the source and the literal associated character set are the Unicode character set. Instead, the exact sequence of code points of these literals is preserved. In effect, this does not change the existing behavior of tested implementations in phase 1 (and phase 5 is already specified on a per-codepoint basis).

[1] Unicode 13 Standard, and confirmed in a mailing list discussion with members of the unicode consortium

Proposal 4: Making trailing whitespaces non-significant

There is a divergence of implementation in how compilers handle spaces between a backslash and the end of a line.

```
int main() {
    int i = 1
    // \
    + 42
    ;
    return i;
}
```

EDG(tested with icc front-end) GCC and Clang will trim the whitespaces after the backslash - and return `1` - MSVC will not and return `43`. Both strategies are valid as part of phase 1 "implementation-defined mapping". There is a clang issue about this #15509

To avoid this surprising implementation divergence we proposed that an implementation must trim all trailing whitespaces before handling \ slicing. This is reinforced by the fact that IDES and tools may discard such whitespaces. The Google-style guidelines forbid trailing whitespaces.

An additional or alternative approach is to deprecate \ that are not part of a preprocessor directive. We are not proposing this at this time.

Unlike other proposals in this paper, this maybe is a silent breaking change for code that is only compiled with MSVC. A quick analysis of the packages available in VCPKG didn't find any trailing whitespaces after backslashes.

We have not been able to measure the impact of this proposed change in the MSVC ecosystem. Other compilers, and all code supported by these compilers would be unaffected.

However, raw string literals should preserve the exact sequence of trailing whitespaces after line splicing reversal. The reversal of line splicing in raw strings might not be sufficiently specified, leading to some implementation divergence and open GCC issues #91412, #43606

Proposal 5: Restricting multi-characters literals to members of the Basic Latin Block

To better understand proposal 5 and 6, it is worth keeping in mind that *user perceived characters* such as 공 may be represented by several codepoints - in this case, ㄱ and ㅎ. é might be e and ◌́. As such what may look like a character literal composed of a single *c-char* may, in fact, be a 'multi-character' literal, without it being visually distinguishable in the source file. This is also the case for others scripts, including many Brahmic scripts and emojis[2]

`int i = 'é';` can be equivalent to either `int i = '\u00e9';` or `int i = 'e\u0301';` depending on source encoding and normalization. There is also a lot of divergence of implementations in how literals are interpreted.

[2] This is not easily demonstrated in LaTeX

	Clang	GCC UTF-8	MSVC UTF-8	GCC Latin-1	MSVC Latin-1
`'e\u0301';`	ill-formed	`int(0x65CC81)` + Warning	`int(0x65cc81)`	ill-formed	`int (0x653f)`
`'\u00e9';`	ill-formed	`int(0xC3A9)` + Warning	`int(0xC3A9)`	`0xFFFFFFFFFFFFFFE9`	`int(0x09)`

Note the presence of two code points in the first line.

We propose to limit multi-character literals to a sequence of code points from the Unicode Basic Latin Block (More or less equivalent to the ASCII character set) to limit the current confusion.

(We do not propose to deprecate multi-character literals).

With the proposed change:

```
'c'   // OK
'abc' // OK, multi-characters literal
'\u0080' // OK (if representable in the execution encoding)
'\u0080\u0080' // ill-formed
'é' // OK (if representable in the execution encoding) if one code point (NFC, U+00e9), otherwise
(e\u0301) ill-formed
```

Proposal 6: Making wide characters literals containing multiple or unrepresentable c-char ill-formed

The following constructs are well-formed but have widely different interpretations depending on implementations

```
wchar_t a = L'🧍';
wchar_t b = L'ab';
wchar_t c = L'é';
```

- the size of `wchar_t` being implementation defined, `L'🧍'` is correctly interpreted on Unix platforms where that size is 32 bits, but truncated by MSVC and other compatible windows compilers where `wchar_t` is 16 bits. MSVC first converts to UTF-16, and truncate to the first code unit which results in an invalid lone high surrogate `0xd83d`.
- `L'ab'` is equivalent to `L'a'` on msvc and `L'b'` on GCC and Clang. All implementations emit a warning under different warning levels
- `L'é'` can be either 1 or 2 c-char depending on the source normalization: `L'\u00e9'` behaves expectedly on all platforms, while `L'e\u0301'` will be `e` in MSVC and `U+0301` in GCC AND clang.

As such, we propose to make wide characters literals with multiple-chars or char which are not representable in the execution character set ill-formed.

Note that wide characters literals with multiple c-char, unlike multi-character-literals are represented by a single `wchar_t`. The other difference is that Unicode combining characters may be representable by a `wchar_t` whereas they cannot be represented by a `char`. The first Unicode combining characters appear in the *Combining Diacritical Marks* block, starting at `U+0300`.

Proposal 7: Making conversion of character and string literals to execution and wide execution encoding ill-formed for unrepresentable c-char

Implementations diverge on how they handle unrepresentable code points when conversion to execution encodings. GCC and Clang make the conversion ill-formed while MSVC usually replaces unrepresentable characters by a single question mark ?. Strings are text which carries intent and meaning; an implementation should not be able to alter that meaning.

We proposed to make such conversion ill-formed rather than implementation-defined.

After discussions in SG-16, we do not propose to allow implementations to consider multi-code points graphemes clusters when doing that conversion. For example considering `"e\u0301"`, `U+0301` does not have a representation in Latin-1, but the abstract character é does (U+00e9 maps to 0x00E9 in Latin 1).

However, it does not seems possible to guarantee that an implementation knows about all such mapping, which would lead to further implementation divergence and unnecessary burden on compilers. We, therefore, propose to be explicit about the conversion being done on each code point independently as is currently the case.

Proposal 8: Enforcing the formation of universal escape sequences in phase 2 and 4

EDG(icc), GCC, MSVC and Clang form universal character names from the following codes:

```
'\\
u00e9';
//---
#define CONCAT(x,y) x##y
CONCAT(\, U0001F431);
```

However, these behaviors are currently UB within the standard. We propose to standardize existing practice by making these behaviors well-defined.

Proposal 9: Reaffirming Unicode as the character set of the internal representation

The standard already specifies that characters outside of the basic source character set are converted to UCNs whose values are isomorphic to Unicode. We want to make it clear that characters that do not have representation in Unicode are ill-formed. This includes some characters in some Big5 encodings and exotic languages such as Klingon.

In particular, all characters in EBCDIC [3] , GB 18030 have a unique mapping in Unicode. The intent is to avoid the use of unassigned code points or the Private Use Area by the implementers, as well as to preserve semantic meaning in phase 1. The preservation of semantic meaning would also make **invalid UTF-8 sequences ill-formed in phase 1**, and other decoding errors

[3] For EBCDIC, the mapping of control characters is specified in Unicode Technical Report 16 - UTF-EBCDIC. This mapping is not semantic-preserving, to the extent control characters have semantics.

(as there exist no mapping for such invalid sequence). Octal/hexadecimal escape sequences can be used in string literal to form arbitrary binary data.

Notably, it is important to consider that the current specification limits the implementation-defined mapping to universal character names to valid (0-10FFF) code points, and any such valid code point can appear in the source before phase 1. It is therefore not possible for an implementation to uniquely map unrepresentable characters to a valid code point.

We only seek to mandate *representability* in Unicode. This proposal has no bearing on the actual internal representation strategy of already conforming implementations. Notably, mandating internal as-if Unicode representation doesn't preclude bytewise preservation of narrow and wide string literals when the execution encoding is identical to the source encoding, as long as there exists a Unicode representation, as this is otherwise non-observable: It is an important implementation strategy for encoding which are not roundtrip-able through Unicode such as Shift JIS to preserve the byte content of string literals when both source and execution encodings are identical. Such preservation is otherwise non-observable and doesn't need to be mandated, but it needs not to be precluded. Again, this constraint exists in practice as many implementations use Unicode internally and do not use the "implementation-defined mapping" leeway to nefarious ends.

Specifically:

- EBCDIC encodings would be converted to Unicode according to UTF-EBCDIC / IBM CDRA - as such, EBCDIC specific control characters are mapped to C1 control characters. C1 controls characters have no meaning on their own and are designed to be interpreted in an application-specific manner.
- GB 18030 maps to Unicode, but a handful of code points maps to the Private Use Area
- Big 5: Most abstract characters map to Unicode, with the rare exception of some spelling of some people or place names. Notably, Unicode prescribes a mapping for Windows implementation (code page 951/950)
- All other encodings have a complete, semantic preserving mapping to assigned Unicode code points.

That list does not intend to be prescriptive, but to show that the C++ standard doesn't need and shouldn't try to handle characters not representable of Unicode. Furthermore, the author hopes that the wording can convert *universal-character-names* to Unicode code points as soon as they are encountered or formed (which is exclusively a matter of wording, that would neither affect behavior nor prescribe an internal representation).

Following this clarification we hope to fix CWG1332 [6]

How to handle ill-formed code unit sequences in phase 1?

Ill-formed code unit sequences can appear if the assumed source encoding does not match the actual source encoding. This scenario should be ill-formed as it most certainly is a bug. However, if these sequences appear in comments, they may be safely ignored in some cases,

and we shouldn't preclude implementers to do so. They should just not appear in text elements (identifiers and literals).

Proposal 10: Make `L` in `_Pragma` ill-formed

`_Pragma(L"")` is equivalent to `_Pragma("")`.

We propose to remove the `_Pragma(L"")` syntax as both strings are interpreted as sequences of Unicode code points and never as a wide execution literal. C++ handling of text is confusing enough not to add meaningless characters. This would resolve CWG897 [5]. Note that there is a divergence of implementation between C++ and C where C discard all prefixes and C++ only discards L.

Out of the 90 millions lines of code of the 1300+ open source projects available on vcpkg, a single use of that feature was found within clang's lexer test suite, for a total of 2000 uses of `_Pragma`. Similarly, the only uses of `_Pragma (u8"")`, `_Pragma (u"")`, `_Pragma (U"")`, etc were found in Clang's test suite (both because these are valid C and because neither GCC nor Clang are comforming, only `L""` is described as valid by the C++ standard).

Proposal 11: Make character literals in preprocessor conditional behave like they do in C++ expression

```
#if 'A' == '\x65'
#endif
if ('A' == 0x65){}
```

Both conditions are not guaranteed to yield a similar result, as the value of character literals in preprocessor conditional is not required to be identical to that of character literals in expressions.

However, a survey of the 1300+ open sources projects available on vcpkg shows that the primary use case for these macros is exactly to detect the narrow literal encoding at compile time and all compilers available on compiler explorer treat these literals as if they were in the narrow literal encoding.

Notably, a few libraries use that pattern to detect EBCDIC or ASCII narrow literal encoding. Of the 50 usages of the pattern, all but one where in C libraries.

While we think there should be a better way to detect encodings in C++ [4], there is no reason to deprecate that feature.

Instead, we recommend adopting the standard practice and user expectation of converting these literals to the narrow literal encoding before evaluating them.

This also removes yet another theoretical encoding, which further simplifies the mental model.

Proposal 12: Improved wording for phase 6 string concatenation

String literal concatenation happen **after** each literal has been converted to its associated encoding in phase 5, and as such the standard is actively encouraging mojibake in the following scenarios:

```
L"" ""
u8"" ""
u"" ""
U"" ""
"" u""
"" L""
"" U""
"" u8""
```

While string conversion is definitively useful, the current behavior is not.

Indeed, each fragment may be in a different encoding after concatenation: UTF-8 data concatenated to Shift-JIS data, for example.

We propose 2 possible resolutions:

- The program is ill-formed unless each *string-literal* part of a concatenation has the same prefix
- The encoding of the first *string-literal* determines the encoding of the *string-literal* resulting from the concatenation and the program is ill-formed if any but the first *string-literal* literal has an encoding prefix:

  ```
  ""   ""   // OK, const char*
  u8"" ""   // OK, equivalent to u8"" u8"", const char8_t*
  u""  ""   // OK, equivalent to u"" u"", const char16_t*
  ""   ""   // OK, equivalent to U"" U"", const char32_t*
  L""  ""   // OK, equivalent to L"" L"", const wchar_t*
  ""  L""   // ill-formed
  L"" u""   // ill-formed
  ```

  In which case, each string-literal should be interpreted as having the same prefix in phase 5, so they are converted to the same encoding prior to concatenation.

A more conservative approach would be to only make ill-formed sequences with heterogenous prefixes such as `L"" u""`. Such conditionally supported sequences are unsupported by existing implementations Compiler explorer. This option is explored in a separate paper by Jeans Maurer.

There is implementation divergence in the handling of concatenation:

MSVC converts each string to its associated execution encoding, as specified by phase 6, while GCC and clang consider the following snippets equivalent (where `X` is any valid *encoding-prefix*).

```
X""  ""
"" X""
X"" X""
```

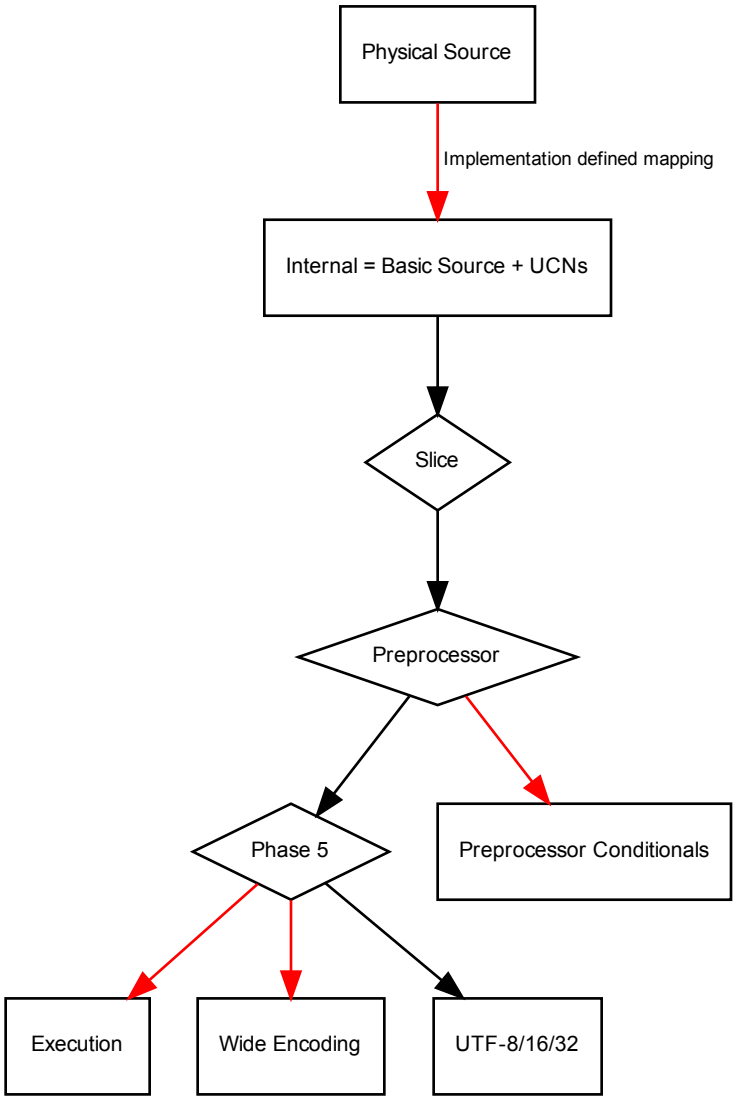
A better wording mechanism would be:

- A single encoding is determined from the encoding prefix of all strings
- Each string is converted to that encoding.
- Strings are concatenated
- A null character is added.

Annex: Schematization of text encodings handling during compilation

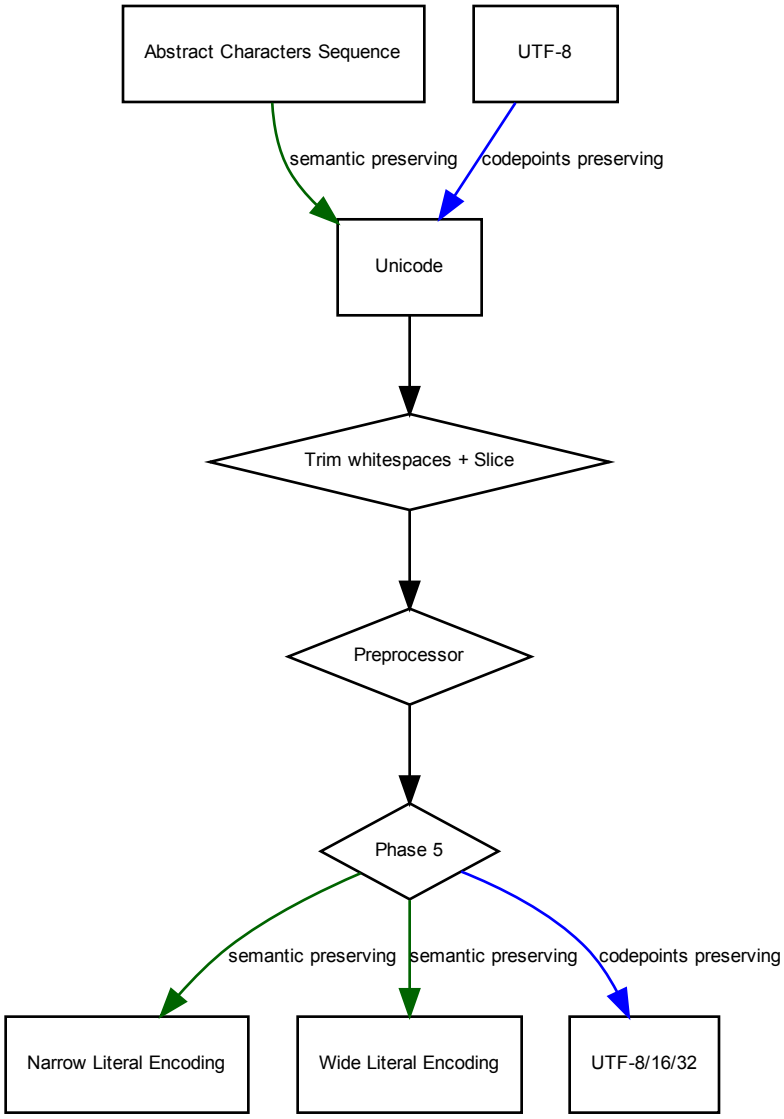
Current model

The graph below is a simplification of the different encodings that appear during compilation. Each rectangle represents a possibly different encoding. The red arrows represent operations that may alter the semantic of text elements.



Proposed model

In the proposed model, all conversions are either semantic preserving[4] or ill-formed. Preprocessor Conditionals use the narrow literal encoding (execution literal encodings). Outside of raw literals, *universal-character-names* are converted to codepoints as they are formed, the wording is specified in term of the Unicode character set.



[4]The value of multi-character literals remains implementation-defined

Acknowledgments

Thanks to the people who provided feedback on the proposed changes, notably Tom Honnermann, Hubert Tong, Aaron Ballman, Steve Downey, Roger Orr, Jeans Maurer.

Many thanks to Peter Brett for his support and his thorough feedback.

References

[1] Beman Dawes. N3463: Portable program source files. https://wg21.link/n3463, 11 2012.

[2] Steve Downey, Zach Laine, Tom Honermann, Peter Bindels, and Jens Maurer. P1949R3: C++ identifier syntax using unicode standard annex 31. https://wg21.link/p1949r3, 4 2020.

[3] Corentin Jabot. P1854R0: Conversion to execution encoding should not lead to loss of meaning. https://wg21.link/p1854r0, 10 2019.

[4] Corentin Jabot. P1885R2: Naming text encodings to demystify them. https://wg21.link/p1885r2, 3 2020.

[5] Daniel Krügler. CWG897: _pragma and extended string-literals. https://wg21.link/cwg897, 5 2009.

[6] Mike Miller. CWG1332: Handling of invalid universal-character-names. https://wg21.link/cwg1332, 6 2011.

[7] Mike Miller. CWG1655: Line endings in raw string literals. https://wg21.link/cwg1655, 4 2013.

[UAX-14] *UNICODE LINE BREAKING ALGORITHM*
https://www.unicode.org/reports/tr14/

[Unicode] Unicode 13
http://www.unicode.org/versions/Unicode13.0.0/

[CDRA] IBM
IBM Character Data Representation Architecture
https://www.ibm.com/downloads/cas/G01BQVRV

[UTF-EBCDIC] V.S. Umamaheswaran
UTF-EBCDIC - Unicode Technical Report #16
http://www.unicode.org/reports/tr16/tr16-8.html

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4861