

Support for UTF-8 as a portable source file encoding

Document #: P2295R6
Date: 2022-07-01
Programming Language C++
Audience: SG-16, EWG, SG-22
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Peter Brett <pbrett@cadence.com>

We should have some notion of a portable C++ source file, and without a known fixed encoding it's hard to argue that such a thing exists — Richard Smith

Credits

While many people, notably many people from SG-16 provided feedbacks on the design presented in this paper, the general direction was originally proposed in 2012 by Beman Dawes in [N3463](#) [1].

Thanks Mr. Dawes!

Abstract

We propose that UTF-8 source files should be supported by all C++ compilers.

Revisions

R6

- Remove a note stating the wording was against [P2314R2](#) [7] as this paper has been merged.
- Rewording using CWG consensus wording.

R5

- Change wording following SG-16 guidance

R4

- Improve wording following SG-16 guidance

R3

- Improve wording following SG-16 guidance

R2

- Add references for BOM guidelines
- Clarify that Clang will support a wider range of source encodings in the future
- Clarify that phase 5 codepoint preservation is handled by [P2314R1](#) [6].

R1

- Remove the section about whitespaces and associated wording

Motivation

Even this simple program cannot be written portably in C++ and may fail to compile:

```
int main() {}
```

— Beman Dawes

The set of source file character sets is implementation-defined, which makes writing portable C++ code impossible. We propose 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 UTF-8 as one of their input format - Clang only supports UTF-8 (but there is an ongoing effort to extend the set of supported source encodings).

Previous works

Original Proposal: [N3463](#) [1] (Beman Dawes, 2012). More recent discussions about Normalization preservation, whitespace, and UTF in [P2178R1](#) [3] (Points 1, 2, 3, and 8)

Design

Codepoint preservations

An important reason to mandate support for UTF-8 support is that it allows us to mandate codepoint preservation during lexing: the compiler should not try to normalize UTF encoded strings, remove characters, and so forth so that users can expect the content of strings to be preserved through translation. (This does not negate that neither the source encoding nor the encoding used internally by the compiler are unobservable by the program).

Invalid code units

Not all code units or code unit sequences represent valid code points. For example, in UTF-8, some sequences can be overlong (`0xC0 0xAF`).

In other encodings, some code unit sequences may encode an unassigned code point, which therefore cannot be represented in Unicode. For example `0x8 0x00` is an unallocated JIS X 0208:1997 sequence.

In both cases, we believe the program should be ill-formed. Both these scenarios seldom occur in practice except when the implementation (or the user) infers the wrong encoding. For example, in `windows-1251`, `0xC0` represents the cyrillic letter A (U+0410), which when interpreted as UTF-8 results in an invalid code unit sequence.

As such, the compiler is interpreting the source file incorrectly and the result of the compilation cannot be trusted, resulting in compilation failures or bad interpretation of string literals further down the line.

The issue can sometimes be innocuous: frequently, non-ASCII characters appear in comments of files that otherwise only contain ASCII, usually when the name of a maintainer appears in a source file. However, most mojibake (encoding miss-interpretation issues) cannot be detected. If the compiler detects invalid characters in comments, there may be undetectable mojibake in string literals, which would lead to runtime bugs.

Therefore, we think invalid code unit sequences in phase 1 should always be diagnosed. Our proposed wording makes invalid code unit sequences ill-formed for the UTF-8 encoding.

However, this does not necessarily constitute a breaking change (see the "Impact on implementations" section)

A note on unassigned Unicode codepoints

When mapping from UTF-8 to any representation of a Unicode code point sequence, it is neither necessary to know whether a codepoint is assigned or not. The only requirement on codepoints is that they are scalar values. As such, the proposed change is completely agnostic of the Unicode version. The only time the Unicode is relevant during lexing is for checking whether an identifier is a valid identifier.

BOM

Unlike [N3463 \[1\]](#), we do not intend to mandate or say anything about BOMs (Byte Order Mark), following Unicode recommendations [1]. 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 source refactoring tools. They are also widely used, and different companies have different policies.

In any case, mandating BOM recognition goes against Unicode recommendations¹. But it is a widespread practice, and as such we recommend neither mandating nor precluding any behavior.

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

For the purpose of translation, a leading BOM is therefore ignored (and doesn't affect the column count of `source_location::column()`).²

Non goals

Other existing and upcoming papers try to improve various aspects of lexing related to Unicode, and text encoding, some of them described in [P2178R1](#) [3] and [P2194R0](#) [5]. This paper only focuses on the handling of UTF-8 physical source files.

In particular, it doesn't try to provide a standard way to specify or detect encoding for the current translation unit or current source file. This proposal conserves the status quo: the mechanism by which encoding is inferred by the compiler for each source file remains implementation-defined.

We further do not propose to restrict in any way the set of input encodings or "physical source character set" accepted by compiler beyond mandating that this implementation-defined set contains at least the UTF-8 encoding.

We do not propose a standard mechanism to specify a different encoding per header. This may be explored in a separate paper.

Finally, conservation of code point sequences in phase 5 of translation when encoding a narrow literal character or string in the UTF-8 encoding is not proposed in this paper which focuses on phase 1 of translation. This is covered by [P2314R1](#) [6].

Impact on implementations

UTF-8 support

MSVC, EDG, Clang, GCC support compiling UTF-8 source files.

- Currently (Clang 11), Clang only support UTF-8 and assume all files are UTF-8. BOMs are ignored.
- GCC supports UTF-8 through `iconv` and the command line flag `-finput-charset=UTF-8` can be used to interpret source files as UTF-8. The default encoding is inferred from the environment and fallbacks to UTF-8 when not possible. BOMs are ignored.
- MSVC supports UTF-8 source files with the `/source-charset:UTF-8` command line flag. MSVC uses UTF-8 by default when a BOM is present.

Input validation

Compilers currently have very different strategies for handling invalid input:

²Unicode Standard, 23.8: *In those cases, it is not part of the textual content and should be removed before processing, because otherwise it may be mistaken for a legitimate zero width no-break space.*

- GCC will ensure that a non-UTF-8 input decodes cleanly and will emit an error otherwise. However, when the input is UTF-8 it is not decoded at all (GCC uses internally) and so the input is not validated. The handling of UTF-8 is then inconsistent with other encodings. We don't know if this is intentional.
- Clang does not check invalid comments. By reading the source code this is very intentional. However invalid Unicode is diagnosed (error) in string literals.
- By default, MSVC will emit a warning for invalid UTF-8, even in comments

```
main.cpp(1): warning C4828: The file contains a character starting at offset 0x2 that is illegal in the current source character set (codepage 65001).
```

As such, implementers have two strategies for the implementation of this proposal:

- Always diagnose invalid code unit sequences when interpreting a UTF-8 input.
- Provide conforming support for UTF-8 inputs, along with an implementation-defined "UTF-8 like" encoding that would behave like UTF-8 but maybe discard invalid code units sequences in comments as part of the "implementation-defined mapping" prescribed in phase 1 for non-UTF-8 encodings.

And so this proposal guarantees that users can have a way to ensure their source code is properly decoded while giving implementers the ability to offer more lenient options.

For example, for MSVC, the flags `/source-charset:utf-8 /we4828` are sufficient to be conforming with the current proposal.

UTF-8 source files is existing practice

- By default, VCPKG compiles all the packages in its repository with `/utf-8` - which sets UTF-8 as the source AND execution encoding ([Reference](#))
- Qt source files are UTF-8 - Users of Qt are recommended to use UTF-8 source files ([Reference](#))
- Chromium is built with UTF-8 (by virtue of being compiled with Clang)

SG16 Polls

See [Minutes for P2178](#)

It should be implementation-defined whether a UTF-8 BOM is used to inform the encoding of a source file

SF	F	N	A	SA
4	3	1	2	0

Consensus is in favor

The presence or absence of a BOM is a reasonable portable mechanism for detecting UTF-8 source file encoding

SF	F	N	A	SA
0	1	0	3	6

No consensus; or rather, consensus is that a BOM is not a reasonable portable mechanism for detection of source file encoding.

We agree that, for Unicode source files, that normalization is preserved through translation phases 1 and 5.

No objection to unanimous consent.

Polls from the 14-04-2021 (R2)

We wish to require implementations to support UTF-8 source files.

No objection to unanimous consent.

We wish to require implementations to be capable of accepting UTF-8 source files whether or not they begin with a U+FEFF byte order mark.

No objection to unanimous consent.

We wish to require implementations to have a mode in which they diagnose ill-formed UTF-8 source files (regardless of whether the ill-formedness is located in comments, header names or string literals).

SF	F	N	A	SA
8	2	0	0	0

Very strongly in favour

Wording



Phases of translation

[lex.phases]

The precedence among the syntax rules of translation is specified by the following phases.

[Editor's note: The following wording represent the CWG consensus. While this wording achieve the intent of this paper, the author finds it unfortunate that we use vague terms like "kind of source file" instead of talking about their encoding. Indeed, by notably keeping the explicit mention of records in the non utf-8 case - a concern that I think should exist outside of the standard, before phase 1 - this wording can be interpreted as utf-8 files stored in records as exempt from the well-formedness and codepoint preserving constraints, which isn't really motivated, and can be the object of further refinements in future papers.]

1. An implementation shall support input files that are a sequence of UTF-8 code units (UTF-8 files). It may also support an implementation-defined set of other kinds of input files, and, if so, the kind of an input file is determined in an implementation-defined manner that includes a means of designating input files as UTF-8 files, independent of their content. [*Note:* In other words, recognizing the U+FEFF Byte Order Mark is not sufficient. — *end note*]

If an input file is determined to be a UTF-8 file, then it shall be a well-formed UTF-8 code unit sequence and it is decoded to produce a sequence of UCS scalar values that constitutes the sequence of elements of the translation character set.

For any other kind of input file supported by the implementation, ~~Physical source file~~ characters are mapped, in an implementation-defined manner, to ~~the translation character set~~ a sequence of translation character set elements (introducing new-line characters for end-of-line indicators). ~~The set of physical source file characters accepted is implementation-defined.~~

2. If the first translation character is U+FEFF BYTE ORDER MARK, it is deleted. Each instance of a backslash character (\) immediately followed by zero or more whitespace characters (other than new-line character) followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice.



lexical conventions

[diff.cpp14.lex]

Valid C++ 2014 code that uses trigraphs may not be valid or may have different semantics in this revision of C++. Implementations may choose to translate trigraphs as specified in C++ 2014 if they appear outside of a raw string literal, as part of the mapping from ~~physical~~ input characters to the translation character set.

Acknowledgments

As usual, many people in SG-16 provided great feedbacks on this paper. In particular JeanHeyd Meneide offered the great insight that invalid UTF-8 can be supported by being considered a different encoding than UTF-8; Tom Honermann, Jens Maurer offered valuable 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. P2178R1: Misc lexing and string handling improvements. <https://wg21.link/p2178r1>, 7 2020.
- [4] Corentin Jabot. P2348R2: Whitespaces wording revamp. <https://wg21.link/p2348r2>, 10 2021.
- [5] Corentin Jabot and Peter Brett. P2194R0: The character set of the internal representation should be unicode. <https://wg21.link/p2194r0>, 8 2020.
- [6] Jens Maurer. P2314R1: Character sets and encodings. <https://wg21.link/p2314r1>, 3 2021.
- [7] Jens Maurer. P2314R2: Character sets and encodings. <https://wg21.link/p2314r2>, 5 2021.
- [Unicode] Unicode 13
<http://www.unicode.org/versions/Unicode13.0.0/>
- [N4901] Thomas Köppe *Working Draft, Standard for Programming Language C++*
<https://wg21.link/N4901>
- [UAX-14] *UNICODE LINE BREAKING ALGORITHM*
<https://www.unicode.org/reports/tr14/>
- [UAX-31] *UNICODE IDENTIFIER AND PATTERN SYNTAX*
<https://www.unicode.org/reports/tr31/>
- [1] Tom Honermann Clarify guidance for use of a BOM as a UTF-8 encoding signature <https://www.unicode.org/L2/L2021/21038-bom-guidance.pdf>