Correct UTF-8 handling during phase 1 of translation

Document #:	P2290R0
Date:	2021-02-15
Project:	Programming Language C++
Audience:	SG-16, EWG, SG-22

Reply-to: Corentin Jabot <corentin.jabot@gmail.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.

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 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.

Previous works

Original Proposal: N3463 [1] (Beman Dawes, 2012). More recent discussions about Normalizxation preservation, whitespaces, 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).

Whitespaces

Unicode recommends that codepoints with the `Pattern_White_Space` property are recognized by whitespaces by conforming parsers. Some sequences of whitespace characters are further described as line breaks in UAX14.

However, we exclude the following codepoints as they would otherwise affect the direction of text and have no use in code (outside of string and comments)

```
U+200E LEFT-TO-RIGHT MARK
U+200F RIGHT-TO-LEFT MARK
```

For compatibility with Unicode, UTF-EBCDIC, and a variety of existing systems, implementations, and text editors, we recommend specifying exactly what constitutes a new-line and a whitespace in C++.

New lines affect diagnostics, `__LINE__` and `source_location::line()`. They also affect line-endings in raw string literals, which is the subject of CWG1655 [5].

An open question is whether we try to use grammar terms to describes new lines and whitespaces in the wording. Some attempts at that proved the task somewhat tedious and the benefits unclear. The current paper does the minimum to be able to remove the ”(introducing new-line characters for end-of-line indicators)” wording, which alas seems necessary to guarantee codepoint preservation; This is why these two otherwise unrelated changes are bundled in the same paper. A more through wording change would impact at least `[lex]`, `[cpp]`, `[time.format]`.

It is also unclear what the `new-line` character is, whether in escape sequences (`\n`), raw line endings, etc. In the current core wording, it doesn't seem to matter much, implementers can use any character they want through the implementation-defined mapping in phase 5.

Invalid code units

Not all code units or code unit sequences represent valid code points. For example, in UTF8, 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 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. 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.

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

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 [4]. 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 encoded 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.

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.

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

- 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:

- 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 recommemded 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.


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

No objection to unanimous consent.
```

SG-16 was generally in favor of specifying what the whitespace characters are, but no polls were taken.

Wording

Note on wording

As SG-16 and Core try to improve the terminology and wording used to describe lexing in the standard, the proposed wording likely collides with other work happening in parallel.

As such, this wording is based on the current standard. Once EWG accepts the design, a CWG-targeting paper merging the different lexing related design changes may be brought forward. As such, the wording reflects the intent but not necessarily the terminology that SG-16 wants to use, nor is it intended to be merged in its current form in the working draft.

Notably, SG-16 and CWG are working on removing the mapping of characters to universal-character-name in phase 1, which has no observable impact on the behavior of C++ programs.

The proposed wording assumes P2223R1 has previously been applied to the wording draft

 Phases of translation [lex.phases]

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

1. A UTF-8 source file is a source file encoded as a sequence of UTF-8 code units representing a sequence of Unicode scalar values as defined in ISO/IEC 10646.

[*Note:* Invalid UTF-8 code unit sequences in a UTF-8 source file are ill-formed. — *end note*] An implementation accepts UTF-8 source files; The set of additional source file character sets accepted is implementation-defined.

How the character set of a source file is determined is implementation-defined.

~~Physical s~~ Source file characters are mapped, in an implementation-defined manner, to the basic source character set ~~(introducing new-line characters for end-of-line indicators) if necessary~~. ~~The set of physical source file characters accepted is implementation-defined~~. Any source file character not in the basic source character set is replaced by the *universal-character-name* that designates that character. If the source file is a UTF-8 source file, the scalar value of each source character shall be preserved. An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a *universal-character-name* (e.g., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.

2. If the first codepoint 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.

🔍 Preprocessing tokens [lex.pptoken]

new-line:
U+000A LINE FEED
U+000B VERTICAL TAB
U+000C FORM FEED
U+000D CARRIAGE RETURN
U+0085 NEXT LINE
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
U+000D U+000A

whitespace:
new-line
U+0009 HORIZONTAL TAB
U+0009 SPACE

Unless otherwise specified, whitespaces and whitespace characters refer to *whitespace*.

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.

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 and Peter Brett. P2194R0: The character set of the internal representation should be unicode. https://wg21.link/p2194r0, 8 2020.

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

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

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

[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/