The character set of C++ source code is Unicode

Document No.	**P2194 R0**	Date	2020-08-24
Reply To	Corentin Jabot corentin.jabot@gmail.com Peter Brett pbrett@cadence.com	Audience	SG16

Introduction

In C++20, the notional internal representation of C++ source code after phase 1 of translation is the Unicode character set. In SG16 meetings, the idea of relaxing the specification to require a superset of the Unicode character set was raised.

This paper describes the *status quo*, discusses the motivation for allowing a non-Unicode internal character set, and concludes that there is no compelling evidence in favour of such a change. The *status quo* enables portable and comprehensible C++ source code, and adding implementation-defined behaviour here would not provide any practical benefits for C++ programmers nor useful flexibility for implementers. This is an excellent opportunity not to give into "excessive inventiveness."

This paper is informative only and is intended to complement and facilitate SG16's ongoing work to improve the specification of C++ lexing and parsing [1].

Improving the specification of lexing and parsing

SG16's direction paper expected that, in C++20 and beyond, most of our work would be on library features for working with text, such as Unicode algorithms such as segmentation; text transcoding support; and text container and view types [2].

However, in recent months we have realised that the handling of C++ source code itself is underspecified in a Unicode world, and most of SG16's current work is in that area. For example, SG16 attendees have papers in flight that improve the specification of identifiers, character literal handling, and compile-time string concatenation. This work has motivated a fresh look at how C++ lexing and parsing is specified. There are many opportunities to improve the specification to reduce implementation divergence and improve the portability of standard C++ programs.

Unicode

The Unicode project is an ongoing effort to ensure consistent encoding, representation and handling of text from human writing systems. As of March 2020, the Unicode character set has 143 859 characters covering 154 scripts [3]. The Unicode character set became an international standard as ISO 10646-1:1993, long predating the first published C++ standard, ISO 14882:1998.

Unicode Transformation Format (UTF) character encoding schemes allow lossless encoding and transmission of Unicode text.

There are many benefits in directly using Unicode in the specification of C++:

- WG21 can focus better on C++-specific problems by normatively referencing another International Standard rather than doing something slightly different.

- Unicode aims to be a superset of other character sets. It provides specifications for how convert between UTF and non-Unicode text encodings.
- Unicode provides a way to describe characters and their properties. There is no other comprehensive system for doing so.
- Current industry practice is to specify text handling in terms of Unicode and to use UTF for interchange. UTF-8 is used by over 95% of websites.
- As Unicode becomes more ever more comprehensive, region-specific character encodings such as Shift-JIS are falling out of favour and UTF is being chosen for new projects.

The internal character set in C++20

[lex.phases]#1.1 says (emphasis added):

> **Any source file character not in the basic source character set is replaced by the *universal-character-name* that designates that character.** 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 ([lex.pptoken]) in a raw string literal.

This implementation-defined behaviour is constrained by [lex.charset]#2 (emphasis added):

> A *universal-character-name* designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of *hexadecimal-digits* in the *universal-character-name*. **The program is ill-formed if that number is not a code point** or if it is a surrogate code point.

In C++20, phase 1 of translation must therefore notionally produce a “source text” composed only of Unicode scalar values, no matter how those scalar values are encoded. This means that C++20 source code is Unicode text.

UCN reversal does not require non-Unicode characters

Suppose that a source code contains a raw string literal with something that looks like a *universal-character-name* (UCN) in it:

```
R"(é\u00E9)" // é = U+00E9
```

If following the wording of [lex.phases]#1.1, this would become:

```
R"(\u00E9\u00E9)"
```

[lex.pptoken]#3.1 allows for this transformation to be reversed during phase 3 of translation, in order to preserve the ‘rawness’ of the raw string literal (emphasis added):

> Between the initial and final double quote characters of the raw string, **any transformations performed in phases 1 and 2** (*universal-character-name*s and line splicing) **are reverted**; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified.

Suppose that an implementer sought to implement this literally. The implementation would need to make some distinction between non-basic source characters that were mapped to UCNs by the implementation, and UCNs appearing directly in the input source code.

In principle, an implementation could use an internal representation where 'normal' source characters were mapped to regular Unicode codepoint values, and 'literal' UCNs were mapped to some 'extended' codepoint value outside the valid Unicode range.

C++ is not actually implemented using the phased translation described in the standard. In practice, implementations do not introduce UCNs during phase 1 for non-basic source characters. They either decode the source file into a UTF, or use the source file encoding throughout. UCNs in string literals are processed during phase 5, i.e. while converting string and character literals to the literal encoding. This means that the UCN-like character sequence in the raw string literal above is never interpreted as a UCN.

We should probably avoid making an evolutionary change to the standard that would accommodate an entirely hypothetical implementation strategy. In fact, conforming implementations can already use a non-Unicode internal representation to implement UCN reversal because it would not be observable during compilation or execution.

Trigraphs do not require non-Unicode characters

Trigraphs were removed from C++ in the C++17 standard [4]. They are now an implementation extension to C++ that allows an alternative way to express a basic source character and therefore no longer lie within the scope of the standard. They are permitted by the 'implementation defined mapping' in phase 1. [lex.phases]#1.1 (emphasis added):

> **Physical 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.

Conforming implementations can provide trigraph support without needing non-Unicode characters in their internal representation.

Byte-value preservation does not require non-Unicode characters

There are a small number of source text encodings which contain duplicated characters, or which have multiple characters that map to the same Unicode codepoint. These include EBCDIC and Shift-JIS.

In principle, a hypothetical implementation might wish to ensure that, when the source and literal encodings are the same, then the exact byte sequence encountered in the source file would be preserved in the executable.

The authors believe that the source file encoding should not generally be observable by the running program. Compiling the same C++ program should produce the same executable no matter whether the input was an ASCII file, a UTF-8 file, an EBCDIC punchcard, the back of a pizza restaurant napkin, or chiselled into a rock.

Portable C++ programs should not rely on implementation fortuitously preserving exact byte values. C++ provides several facilities for placing exact byte values into a string or character literal. Furthermore, source code that relies on exact byte value sequences in esoteric encodings may be brittle in the face of many modern tools (editors, version control systems, and code search tools) using UTF internally and re-encoding at the edges.

Nevertheless, the C++ standard already permits implementations to preserve byte values. [lex.phases]#1.5 specifies implementation-defined behaviour (emphasis added):

> Each basic source character set member in a *character-literal* or a *string-literal*, as well as each escape sequence and *universal-character-name* in a character-literal or a non-raw string literal, is converted to the corresponding member of the execution character set ([lex.ccon], [lex.string]); **if there is no corresponding member, it is converted to an implementation-defined member** other than the null (wide) character.

The "no corresponding member" wording covers 2 scenarios: the obvious case of when a Unicode scalar value has no representation in the execution character set; and the subtle case of when a Unicode scalar value has multiple possible representations in the execution character set, i.e. no directly corresponding member.

For example, a conforming EBCDIC-EBCDIC compiler could use EBCDIC as its internal encoding. Its implementation strategy could be to recognize trigraphs directly, rather than by conversion to basic source characters; delay handling of UCN escape sequences to phase 5; and perform conversion from non-basic source characters to UCNs lazily using UTF-EBCDIC.

Such a compiler's internal representation would consist only of Unicode characters, because UTF-EBCDIC defines a mapping from every character in EBCDIC to a Unicode scalar value. The IBM Character Data Representation Architecture (CDRA) provides a fully-specified mapping between EBCDIC and Unicode code points [5]. C0 and C1 control characters do not have any intrinsic semantics that need to be interpreted by a C++ implementation.

Conforming implementations can provide byte-value preservation without needing non-Unicode characters in their internal representation.

Current implementation practice

GCC, MSVC and clang use UTF-8 as their internal representation, converting the entire source code in phase 1.

GCC can optionally be compiled to use UTF-EBCDIC as its internal representation.

GCC and MSVC support a variety of input encodings.

clang supports only UTF-8. The ongoing effort to use clang on z/OS platforms will convert to UTF-8 before clang sees the source file, using z/OS filesystem-level text conversion facilities.

EDG uses the source encoding as its internal representation and transcodes to the literal representation in phase 5.

xl C/C++, the current compiler for z/OS, uses EBCDIC as its internal representation.

Proposed wording

Editing notes

All wording is relative to the post-Prague C++ committee draft [6].

5.2 Phases of translation [lex.phases]

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

1. Physical source file characters are mapped, in an implementation-defined manner, to the Unicode character set 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 (5.3) is replaced by the *universal-character-name* that designates that character. 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 (5.4) in a raw string literal.

References

[1] C. Jabot, "P2178R1 Misc lexing and string handling improvements," 14 July 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2178r1.pdf.

[2] T. Honermann, C. Jabot, J. Meneide, M. Zeren, M. Fernandes, P. Bindels, S. Downey and Z. Laine, "P1238R0 SG16: Unicode Direction," 8 October 2018. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1238r0.html.

[3] The Unicode Consortium, "Unicode 13.0," 10 March 2020. [Online]. Available: https://www.unicode.org/versions/Unicode13.0.0/.

[4] R. Smith, "N3981 Removing trigraphs??!," 6 May 2014. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3981.html.

[5] IBM, "Character Data Representation Architecture," 2018. [Online]. Available: https://www.ibm.com/downloads/cas/G01BQVRV.

[6] R. Smith, T. Koeppe, J. Maurer and D. Perchik, "N4861 Working Draft, Standard for Programming Language C++," 8 April 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf.

Appendix: Unicode control character differentiation

This topic was discussed on the Unicode mailing list during June. The discussion is archived and may be useful during consideration of this paper. https://corp.unicode.org/pipermail/unicode/2020-June/008839.html