Conversion to literal encoding should not lead to loss of meaning

Document #:	P1854R1
Date:	2021-10-09
Programming Language C++	
Audience:	SG-16
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

It's just semantic! - Kevlin Henney

Target

C++23

Abstract

This paper makes non-encodable characters in character and string literals ill-formed and restrict the valid characters in a multi-character literals.

Revisions

R1

- Rebase the wording
- Propose to make non-basic characters in multi-characters ill-formed
- Add more motivation.

Non-encodable character-literals

Implementation defined behaviors related to conversion to literal encoding reduce the portability of C++ programs, and lead to silently incorrect programs as implementations are allowed to substitute the characters they cannot represent. Strings are text which carries intent and meaning. We believe an implementation should not be able to alter that meaning.

A program should either conserve the same sequence of abstract character as in the source, or be ill-formed.

This constitutes a breaking change in the wording, as well as some implementations(MSVC) and matches other existing implementations' behaviour.

This is a follow-up to P2362R3 [1] and a realization of the plan outlined in P2178R1 [2].

Impact on the standard and implementations

- Clang always use UTF-8 to encode narrow literals
- GCC emits a diagnostics

  > converting to execution character set: Invalid or incomplete multibyte or wide character

- MSVC uses `?` as a replacement character. For example, the string `"こんにちは"` becomes `' ?? ?? ', 00H`. MSVC does emits warning for this scenario (enabled by default).

```
<source>(4): warning C4566: character represented by universal-character-name
    '\u3053' cannot be represented in the current code page (20127)
<source>(4): warning C4566: character represented by universal-character-name
    '\u3093' cannot be represented in the current code page (20127)
<source>(4): warning C4566: character represented by universal-character-name
    '\u306B' cannot be represented in the current code page (20127)
<source>(4): warning C4566: character represented by universal-character-name
    '\u3061' cannot be represented in the current code page (20127)
<source>(4): warning C4566: character represented by universal-character-name
    '\u306F' cannot be represented in the current code page (20127)
```

A demonstration of existing behavior is available on Compiler Explorer.

We argue that the code which breaks never match the developer's intent.

Are we removing a capability?

- The exact nature of the literal encoding can be observed by a dedicated api P1885R7 [3], and in general the relying on non-encodable characters to detect the literal encoding is non-portable as it can only work on windows. It is also very difficult to use such clever tricks in a way that has no false positives or false negatives.
- `?` can be inserted in string and character literals.
- `u8` strings can be used portably.
- If the author of the code does not care about the content of a string being preserved, then presumably that character can be removed.

Impact on C

This makes a behavior that is implementation-defined in C ill-formed in C++. GCC exposes the same behavior (the one proposed by this paper) in all language modes.

Multi character literals

Narrow multicharacters literals such as `'intl'` are widely used. We are not proposing to removing them. However, 'é' (e, ACUTE ACCENT) or '🏳️‍🌈' (grapheme cluster), read as single characters. This is the same issue described for wide literals in P2362R3 [1].

Unlike what we proposed for wide-literals, we can't make all the multi-characters literals ill-formed. Instead, we propose that multicharacters literals can only contain characters from the basic-literal character sets.

This has 2 benefits

- It excludes all combining characters or characters that do not constitute a full grapheme. That takes care of the visual ambiguity.
- It makes multicharacters literals slightly less confusing as it is dificile to imagine how multiple codepoints over 0x80 fcould be stuffed into an `int`. in any sensible way.

And indeed, the documentation of GCC shows that codepoints that do not fit in a single bytes are not preserved, but instead truncated.

> The compiler evaluates a multi-character character constant a character at a time, shifting the previous value left by the number of bits per target character, and then or-ing in the bit-pattern of the new character truncated to the width of a target character. The final bit-pattern is given type int, and is therefore signed, regardless of whether single characters are signed or not. If there are more characters in the constant than would fit in the target int the compiler issues a warning, and the excess leading characters are ignored.
> For example, 'ab' for a target with an 8-bit char would be interpreted as `(int) ((unsigned char) 'a' * 256 + (unsigned char) 'b')`, and '\234a' as `(int) ((unsigned char) '\234'  256 + (unsigned char) 'a')`.

Impact on the standard and implementations

GCC, Clang ICC(EDG) emit a warning for any multi-characters literals in general. They also emit a warning when the computed value exceed the size of `int`.

No compiler emit a warning for Unicode in multi-character literals. Because this feature cannot produce sensible result, we do not think its removal would affect users.

Feature macro

No feature macros is proposed because the transformation to characters literals and string literals is not observable by the program.

Proposed wording

[Editor's note: The Magenta text with blue squigly lines correspond to wording previously removed by P2362R3 [1]. This wording should stay removed.]

� Character literals [lex.ccon]

A *non-encodable character literal* is a *character-literal* whose *c-char-sequence* consists of a single *c-char* that is not a *numeric-escape-sequence* and that specifies a character that either lacks representation in the literal's associated character encoding or that cannot be encoded as a single code unit. A *multicharacter literal* is a *character-literal* whose *c-char-sequence* consists of more than one *c-char*. A multicharacter literal shall not have an encoding prefix. Each c-char in a multicharacter literal shall be a member of the basic literal character set.

The *encoding-prefix* of a non-encodable character literal or a multicharacter literal shall be absent or L.

~~Such *character-literals*~~ Multicharacter literals are conditionally-supported.

The kind of a *character-literal*, its type, and its associated character encoding are determined by its *encoding-prefix* and its *c-char-sequence* as defined by [lex.ccon.literal]. ~~The special cases for non-encodable character literals and multicharacter literals take precedence over their respective base kinds.~~

[*Note:* The associated character encoding for ordinary and wide character literals determines encodability, but does not determine the value of non-encodable ordinary or wide character literals or ordinary or wide multicharacter literals. The examples in [lex.ccon.literal] for non-encodable ordinary and wide character literals assume that the specified character lacks representation in the execution character set or execution wide-character set, respectively, or that encoding it would require more than one code unit. — *end note*]

In translation phase 4, the value of a *character-literal* is determined using the range of representable values of the *character-literal*'s type in translation phase 7. A ~~non-encodable character literal or a~~ multicharacter literal has an implementation-defined value. The value of any other kind of *character-literal* is determined as follows:

- A *character-literal* with a *c-char-sequence* consisting of a single *basic-c-char*, *simple-escape-sequence*, or *universal-character-name* is the code unit value of the specified character as encoded in the literal's associated character encoding. ~~[Note:~~ If the specified character lacks representation in the literal's associated character encoding or if it cannot be encoded as a single code unit, then the literal is ~~a non-encodable character literal~~ ill-formed. ~~– end note]~~

Table 1: Character literals

Encoding prefix	Kind	Type	Associated character encoding	Example

none	*ordinary character literal*
	~~non-encodable ordinary character literal~~
	ordinary multicharacter literal
		`char`
		~~`int`~~
		`int`
			encoding of the execution character set	`'v'`
				~~`'\U0001F525'`~~
				`'abcd'`

`L`	*wide character literal*
	non-encodable wide character literal
	wide multicharacter literal
		`wchar_t`
		`wchar_t`
		`wchar_t`
			encoding of the execution wide-character set	`L'w'`
				`L'\U0001F32A'`
				`L'abcd'`

`u8`	*UTF-8 character literal*	`char8_t`	UTF-8	`u8'x'`
`u`	*UTF-16 character literal*	`char16_t`	UTF-16	`u'y'`
`U`	*UTF-32 character literal*	`char32_t`	UTF-32	`U'z'`

- A *character-literal* with a *c-char-sequence* consisting of a single *numeric-escape-sequence* that specifies an integer value v has a value as follows:
  - If v does not exceed the range of representable values of the *character-literal*'s type, then the value is v.
  - Otherwise, if the *character-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *character-literal*'s type, then the value is the unique value of the *character-literal*'s type `T` that is congruent to v modulo 2^N, where N is the width of `T`.
  - Otherwise, the *character-literal* is ill-formed.
- A *character-literal* with a *c-char-sequence* consisting of a single *conditional-escape-sequence* is conditionally-supported and has an implementation-defined value.

� String literals [lex.string]

String literal objects are initialized with the sequence of code unit values corresponding to the *string-literal*'s sequence of *s-char* s (for a non-raw string literal) and *r-char* s (for a raw string literal) in order as follows:

- The sequence of characters denoted by each contiguous sequence of *basic-s-char* s, *r-char* s, *simple-escape-sequence* s, and *universal-character-name* s is encoded to a code unit sequence using the *string-literal*'s associated character encoding. If a character lacks representation in the associated character encoding, then ~~:~~ the *string-literal* is ill-formed.
  - If the *string-literal*'s *encoding-prefix* is absent or `L`, then the *string-literal* is conditionally-supported and an implementation-defined code unit sequence is encoded.

– Otherwise, the *string-literal* is ill-formed.

When encoding a stateful character encoding, implementations should encode the first such sequence beginning with the initial encoding state and encode subsequent sequences beginning with the final encoding state of the prior sequence. [*Note:* The encoded code unit sequence can differ from the sequence of code units that would be obtained by encoding each character independently. — *end note*]

- Each *numeric-escape-sequence* that specifies an integer value v contributes a single code unit with a value as follows:
  – If v does not exceed the range of representable values of the *string-literal*'s array element type, then the value is v.
  – Otherwise, if the *string-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *string-literal*'s array element type, then the value is the unique value of the *string-literal*'s array element type `T` that is congruent to v modulo 2^N, where N is the width of `T`.
  – Otherwise, the *string-literal* is ill-formed.

  When encoding a stateful character encoding, these sequences should have no effect on encoding state.
- Each *conditional-escape-sequence* contributes an implementation-defined code unit sequence. When encoding a stateful character encoding, it is implementation-defined what effect these sequences have on encoding state.

Acknowledgments

Many thanks to JeanHeyd Meneide, Peter Bindels, Zach Laine, Tom Honermann and Steve Downey who reviewed this paper and offered valuable feedback.

References

[1] Peter Brett and Corentin Jabot. P2362R3: Remove non-encodable wide character literals and multicharacter wide character literals. https://wg21.link/p2362r3, 8 2021.

[2] Corentin Jabot. P2178R1: Misc lexing and string handling improvements. https://wg21.link/p2178r1, 7 2020.

[3] Corentin Jabot and Peter Brett. P1885R7: Naming text encodings to demystify them. https://wg21.link/p1885r7, 9 2021.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4885