Make obfuscating wide character literals ill-formed

Document No.	**P2362 R0**	Date	2021-04-14
Reply To	Peter Brett **pbrett@cadence.com** Corentin Jabot corentin.jabot@gmail.com	Audience:	SG16, EWG

Introduction

C++ currently permits writing a wide character literal with multiple characters or characters that cannot fit into a single `wchar_t` codeunit. For example:

```
wchar_t a = L'🤦'; // \U0001f926
wchar_t b = L'ab';
wchar_t c = L'é';  // \u0065\u0301
```

Wide non-encodable and multicharacter literals have wildly different interpretations across different implementations, and it is not feasible to specify a portable and consistent interpretation.

Make these literals ill-formed.

Design

Wide non-encodable character literals

The size of `wchar_t` is implementation-defined. On platforms where `wchar_t` is a 32-bit integer type (e.g. Linux), `L'🤦'` is interpreted as `0x01f926` without loss of information.

On platforms where `wchar_t` is a 16-bit integer type (e.g. Windows), the value is truncated, and there is significant implementation divergence.

MSVC first converts to UTF-16, and then truncates to the first codeunit, producing the invalid lone high surrogate `0xd83e` and a diagnostic (disabled by default). GCC with `-fshort-wchar` first converts to UTF-16, then truncates to the *second* codeunit, producing the invalid lone *low* surrogate `0xdd26` and a diagnostic.

Clang with `-fshort-wchar` treats the input as ill-formed.

Wide multicharacter literals

All the implementations we examined only ever interpret a single character in a wide multicharacter literal. However, there is divergence in which is chosen. MSVC takes the first, treating `L'ab'` as equivalent to `L'a'`, and emits a diagnostic (disabled by default). GCC and Clang take the last, treating `L'ab'` as equivalent to `L'b'`, and emit diagnostics.

`L'é'` may consist of either 1 or 2 *c-char*s depending on source normalization. In the composed form, `L'\u00e9'` produces the value `0xe9` when compiled by MSVC, GCC and Clang. There is divergence in handling the decomposed form `L'\u0065\u0301'`. MSVC produces `0x65`; GCC and Clang produce `0x0301`.

Therefore, what looks like a single *c-char* when reading the source file may, in fact, be a multi-character literal. This is the case in many scripts, including Korean, many Brahmic scripts, and emoji [1].

Proposal

There is irreconcilable implementation divergence in the handling of wide multicharacter literals.

Because all wide character literals have `wchar_t` storage, no implementation can interpret more than one wide codeunit from any wide character literal. The allowance for implementations to accept wide multicharacter literals is redundant.

Similarly, no implementation can handle a non-encodable wide character literal without loss of information.

Using any of the implementations examined, using a wide non-encodable or multicharacter literals provided no benefit whatsoever over using an equivalent 'normal' wide character literal. They only serve to obfuscate and reduce portability.

We propose that wide non-encodable and wide multicharacter literals should be ill-formed.

Ill-formedness will clear the design space for defining a useful, and portable, interpretation of wide non-encodable and/or multicharacter literals in a future revision of the standard, if there is widespread desire for them to be reintroduced.

This change was previously proposed in P2178 [1].

Impact on implementations

Implementations are already able to detect and diagnose wide non-encodable and multicharacter literals. We recommend that implementations update these diagnostics to errors and, for wide multicharacter literals, propose the change that the user should make fix the problem.

Impact on users

Because there is no possible meaningful interpretation of wide multicharacter literals, they are not used. The authors carried out a survey of open source code and found no occurrences outside compiler testsuites.

Summary

	`L'\U0001f926'`	`L'ab'`	`L'\u0065\u0301'`	`L'\u00e9'`
MSVC	⚠ `0xd83e`	⚠ `0x041`	⚠ `0x65`	`0xe9`
Clang -fshort-wchar	⯃ (error)	⚠ `0x042`	⚠ `0x0301`	`0xe9`
GCC -fshort-wchar	⚠ `0xdd26`	⚠ `0x042`	⚠ `0x0301`	`0xe9`
Clang	`0x01f926`	⚠ `0x042`	⚠ `0x0301`	`0xe9`
GCC	`0x01f926`	⚠ `0x042`	⚠ `0x0301`	`0xe9`

Cases marked with a ⚠ currently result in a warning diagnostic. Cases marked with a ⯃ currently result in a compilation error.

We propose that the cases marked with a ⚠ or ⯃ above will become ill-formed.

Proposed wording

Editing notes

All wording is relative to the March 2021 C++ working draft [3].

5.13.3 Character literals [lex.ccon]

Update ¶1:

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*. The *encoding-prefix* of a non-encodable character literal or a multicharacter literal shall be absent~~ or L~~. Such *character-literals* are conditionally-supported.

Update ¶2

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 Table 9. The special cases for non-encodable character literals and multicharacter literals take precedence over their respective base kinds.

[*Note 1*: 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 Table 9 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]

Update Table 9:

Encoding prefix	Kind	Type	Associated character encoding	Example
none	*ordinary character literal*	`char`	encoding of the execution character set	`'v'`
	non-encodable ~~ordinary~~ character literal	`int`		`'\U0001F525'`
	~~ordinary~~ multicharacter literal	`int`		`'abcd'`
`L`	*wide character literal*	`wchar_t`	encoding of the execution wide-character set	`L'w'`
	~~non-encodable wide character literal~~	~~`wchar_t`~~		~~`L'\U0001F32A'`~~
	~~wide multicharacter literal~~	~~`wchar_t`~~		~~`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'`

Update ¶3.2.2

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`.

References

[1] S. Downey, Z. Laine, T. Honermann, P. Bindels and J. Maurer, “P1949R6 C++ Identifier Syntax using Unicode Standard Annex 31,” 15th Sept 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1949r6.html.

[2] 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.

[3] T. Köppe, “N4885 Working Draft, Standard for Programming Language C++,” 17th Mar 2021. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4885.pdf.