Abstract

Character literals in preprocessor conditional should behave like they do in C++ expression. This proposal first appeared as part of P2178R1 [3].

Revisions

R2

Wording fixes.

R1

At EWG’s request we contacted implementers to confirm they had no issue with this proposal. We also added a note explaining why we do not propose a feature macro.

Motivation

Consider the following code:

```c
#if 'A' == '\x41'
  //...
#endif
if ('A' == 0x41){}
```

Both conditions are not guaranteed to yield a similar result, as the value of character literals in preprocessor conditional is not required to be identical to that of character literals in expressions.

However, a survey of the 1300+ open sources projects available on vcpkg shows that the primary use case for these macros is exactly to detect the narrow literal encoding at compile
time and all compilers available on compiler explorer treat these literals as if they were in the narrow literal encoding.

Notably, a few libraries use that pattern to detect EBCDIC or ASCII narrow literal encoding. Of the 50 usages of the pattern, all but one were in C libraries.

For example, this code is in `sqlite.c`

```c
/*
 ** Check to see if this machine uses EBCDIC. (Yes, believe it or
 ** not, there are still machines out there that use EBCDIC.)
 */
#if 'A' == '\301'
 # define SQLITE_EBCDIC 1
#else
 # define SQLITE_ASCII 1
#endif
```

While we think there should be a better way to detect encodings in C++ [1], there is no reason to deprecate that feature.

Instead, we recommend adopting the standard practice and user expectation of converting these literals to the narrow literal encoding before evaluating them.

**SG16 poll**

August 26th, 2020
Poll: Proposal 11 (of P2178R0 [2]): We agree that the same character encoding should be used for character literal in translation phase 4 and 7.

Attendees: 10
No objection to unanimous consent.

**Implementers feedback**

MSVC and EDG implementers reported having no issue with this proposal and confirmed it already matches existing behavior of their respective implementation. This is also the case for Clang and GCC (whose code is open source).

**Feature Macro**

Because this paper proposes to standardize the behavior of all existing implementations, we are not proposing the addition of a feature macro.
Wording

**Preprocessing directives**

The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of [expr.const] using arithmetic that has at least the ranges specified in [support.limits]. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, intmax_t or uintmax_t. **[Note: Thus on an implementation where std::numeric_limits<int>::max() is 0x7FFF and std::numeric_limits<unsigned int>::max() is 0xFFFF, the integer literal 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7. — end note]**

This includes interpreting character-literal s which may involve converting escape sequences into execution character set members. Whether the numeric value for these character-literal s matches the value obtained when an identical character-literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined. **[Note: Thus, the constant expression in the following #if directive and if statement is not guaranteed to evaluate to the same value in these two contexts:]

```c
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

— end note] Also, whether a single-character character-literal may have a negative value is implementation-defined. **This includes interpreting character-literal according to the rules in [lex.ccon]. [Note: The associated character encodings of literals are the same in #if and #elif directives and in any expression. — end note]**

Each subexpression with type bool is subjected to integral promotion before processing continues.

**Acknowledgments**

**References**


