Delimited escape sequences

Document #:	N2785
Date:	2021-07-28
Project:	Programming Language C
Audience:	Application programmers
Proposal category:	New feature

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Aaron Ballman <aaron@aaronballman.com>

Abstract

We propose an additional, clearly delimited syntax for octal, hexadecimal and universal character name escape sequences to clearly delimit the boundaries of the escape sequence. WG21 has shown an interest in adjusting this feature, and this proposal is intended to keep C and C++ in alignment. This feature is a pure extension to the language that should not impact existing code.

Motivation

universal-character-name escape sequences

As their name does not indicate, universal-character-name escape sequences represent Unicode scalar values, using either 4, or 8 hexadecimal digits, which is either 16 or 32 bits. However, the Unicode codespace is limited to 0-0x10FFFF, and all currently assigned codepoints can be written with 5 or less hexadecimal digits (Supplementary Private Use Area-B non-withstanding). As such, the ~50% of codepoints that need 5 hexadecimal digits to be expressed are currently a bit awkward to write: `\U0001F1F8`.

Octal and hexadecimal escape sequences have variable length

`\1`, \01, \001 are all valid escape sequences. `\17` is equivalent to `"0x0F"` while `\18` is equivalent to `"0x01" "8"`

While octal escape sequences accept 1 to 3 digits as arguments, hexadecimal sequences accept an arbitrary number of digits applying the maximal much principle.

This is how the Microsoft documentation describes this problem:

> Unlike octal escape constants, the number of hexadecimal digits in an escape sequence is unlimited. A hexadecimal escape sequence terminates at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters a through f, care must be exercised to make sure the escape sequence terminates at the intended digit. To avoid confusion, you can place octal or hexadecimal character definitions in a macro definition:
>
> ```
> #define Bell '\x07'
> ```
>
> For hexadecimal values, you can break the string to show the correct value clearly:
>
> ```
> "\xabc"    /* one character  */
> "\xab" "c" /* two characters */
> ```

As this documentation suggests, there are workarounds to this problem. However, neither solution completely solves the maintenance issue. It may not be clear why a string literal uses literal concatenation, so a future refactoring of the code may accidentally combine the strings. Further, literal concatenation is not a common pattern for arbitrary string literal uses.

Status of P2290R1 [2] in WG21

Lastly, we propose this feature for C for compatibility with the C++ proposal P2290R1 [2]. This feature is presented to the C committee to ensure that either C adopts it (for consistency) or does not object to C++ using this syntax. We hope that if C should adopt this feature in the future, it would do so with both the same syntax and semantics as what is proposed in P2290R1 [2]. Our hope is that WG14 will be interested in adopting this feature for C23 based on the merits of the feature. However, we recognize that WG14 may not be ready to adopt this proposal yet, and so our secondary goal is to identify concerns WG14 has with the feature to ensure that future work in this area within C is done in a way that is compatible with C++. We don't believe there is much room for different designs for this functionality, but we want to make sure WG21 doesn't pick curly braces while WG14 picks parentheses or other such superficial incompatibilities.

EWG took the following straw poll in July 2021:

> We would like to adopt this for C++23, assuming the Core wording is improved in the paper and assuming SG22 / WG14 intend to avoid divergence in C.
>
SF	F	N	A	SA
1	10	0	0	0

Proposed solution

We propose new syntaxes `\u{}`, `\o{}`, `\x{}` usable in places where \u, \x, \nnn currently are. \o{} accepts an arbitrary number of octal digits and `\u{}` and `\x{}` accept an arbitrary number of hexadecimal digit.

The values represented by these new syntaxes would, of course have the same requirements as existing escape sequences:

\u{nnnn} must represent a valid Unicode scalar value.

\x{nnnn} and \o{nnnn} must represent a value that can be represented in a single code unit of the encoding of string or character literal they are a part of.

Note that `"\x{4" "2}"` would not be valid as escape sequences are replaced before string concatenation, which we think is the right design.

We explicitly do not allow digit separators in these digit sequences. This is to avoid a parsing ambiguity for character literals where `'` already has special meaning, as in: `'\u12'34'`.

Should existing forms be deprecated?

No (we are not in the business of breakings everyone's code)!

Impact on existing implementations

No compiler currently accepts `\x{` or `\u{` as valid syntax. Furthermore, while `\o` is currently reserved for implementations, no tested implementation (GCC, Clang, MSVC, ICC, TCC, Tendra) makes use of it.

Identifiers

Universal character names can appear as part of identifiers, and the proposed new syntax to spell universal characters names would be applied to identifiers for consistency.

Prior art and alternatives considered

`\u{}` is a valid syntax in rust and javascript. The syntax is identical to that of P2071R0 [1]

`\x{}` is a bit more novel - It is present in Perl and some regex syntaxes. However, most languages (python, D, Perl, javascript, rust, PHP) specify hexadecimal sequences to be exactly 2 hexadecimal digits long (`\xFF`), which sidestep the issues described in this paper.

Most languages surveyed follow in C and C++ footstep for the syntax of octal numbers (no braces, 1-3 digits), so this would be novel indeed.

As such, for consistency with other C++ proposals and existing art, we have not considered other syntaxes.

Impact on EBCDIC programs

In some EBCDIC encodings, the `{ }` characters might not be avaible. This was raised in the context of the `filetag` pragma such that xCl users often have to specify the encoding of a source file.

```
??=pragma filetag ("IBM-1047")
```

In this specific context, hexadecimal and octal sequences are not useful - the string is not evaluated. *Universal characters name* s can be specified but, *universal characters name* s exclude most of the basic Latin 1 characters used to specify encodings. Encoding names are designed to safe for inter exchange and as such avoid non-basic-latin characters [1] [2], making the proposed feature of limited usefulness in the context of `pragma filetag` [3].

After this initial pragma, most code targeting EBCDIC platforms can use braces. Otherwise, users can either use the pre-existing syntaxes or use trigraphs.

Given the existing syntax for delimiting lists of elements with curly braces in C already (enumerations, initialization, structure members, etc) and the expectation that delimited escape sequence use is expected to be vanishingly rare in pragmas and identifiers, we believe delimiting with `{` and `}` is not an undue burden for implementers or users.

Wording

6.4.3 Universal character names

Syntax

hex-quad:
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

simple-hexadecimal-digit-sequence:
hexadecimal-digit
simple-hexadecimal-digit-sequence hexadecimal-digit

universal-character-name:
`\u` *hex-quad*
`\U` *hex-quad hex-quad*
`\u{` *simple-hexadecimal-digit-sequence* `}`

[Editor's note: Remove The following paragraph in red]

Constraints

A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 ($), 0040 (
@), or 0060 (
‘), nor one in the range D800 through DFFF inclusive.

Description Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Semantics ~~The universal character name \Unnnnnnnn designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is nnnnnnnn. Similarly, the universal character name \unnnn designates the character whose four-digit short identifier is nnnn (and whose eight-digit short identifier is 0000nnnn).~~

[Editor's note: Remove footnote 79]

A universal character name designates the character in ISO/IEC 10646 whose code point is the hexadecimal number represented by the sequence of hexadecimal digits in the universal character name. That code point shall not be in the range D800 through DFFF inclusive, nor less than 00A0, except for 0024 ($), 0040 (@), or 0060 (') [Editor's note: Attach footnote 78 here] .

6.4.4.4 Character constants

Syntax

numeric-escape-sequence:
octal-escape-sequence
hexadecimal-escape-sequence

simple-octal-digit-sequence:
octal-digit
simple-octal-digit-sequence octal-digit

octal-escape-sequence:
\ *octal-digit*
\ *octal-digit octal-digit*
\ *octal-digit octal-digit octal-digit*
\o{ *simple-octal-digit-sequence* }

hexadecimal-escape-sequence:
\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit
\x{ *simple-hexadecimal-digit-sequence* }

[Editor's note: Edit: Paragraph 5-7 as follow]

The octal digits ~~that follow the backslash~~ in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

The hexadecimal digits ~~that follow the backslash and the letter x~~ in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

References

[1] Tom Honermann and Peter Bindels. P2071R0: Named universal character escapes. https://wg21.link/p2071r0, 1 2020.

[2] Corentin Jabot. P2290R1: Delimited escape sequences. https://wg21.link/p2290r1, 6 2021.

[1] I. McDonald *IANA Charset MIB*
https://tools.ietf.org/html/rfc3808

[2] UNICODE CHARACTER MAPPING MARKUP LANGUAGE
https://www.unicode.org/reports/tr22/tr22-8.html#Charset_Alias_Matching

[3] #pragma filetag - z/OS XL C/C++ User's Guide
https://www.ibm.com/docs/en/zos/2.3.0?topic=descriptions-pragma-filetag

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