Document number:	N4219
Date:	2014-10-10
Working group:	SG12
Reply to:	David Krauss (david_work at me dot com)
Revises:	N3881

Fixing the specification of universal-character-names (rev. 2)

1. Background

Universal-character-names (UCNs) were introduced to allow C and C++ source code to express internationalization without dependence on source text encoding (as long as Unicode covers the desired characters). Although *universal-character-name* is a lexical construct, it serves a purpose similar to character encoding. Therefore support for this feature mixes issues of parsing and text encoding. As usual with encoding, there is a trade-off between compatibility and performance. To enable retrofitting, UCNs are designed to be suitable to embed Unicode in the basic source character set. Higher efficiency may be realized by other approaches such as UTF-8, though. This dilemma is acknowledged in the C++ standard by a normative aside ([lex.phases] 2.2/1):

> (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 (i.e., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.)

The C99 rationale document goes into more depth:

> Once this was adopted, there was still one problem, how to specify UCNs in the Standard. Both the C and C++ committees studied this situation and the available solutions, and drafted three models:
>
> A. Convert everything to UCNs in basic source characters as soon as possible, that is, in translation phase 1.
>
> B. Use native encodings where possible, UCNs otherwise.
>
> C. Convert everything to wide characters as soon as possible using an internal encoding that encompasses the entire source character set and all UCNs.
>
> Furthermore, in any place where a program could tell which model was being used, the standard should try to label those corner cases as undefined behavior.
>
> The C++ committee defined its Standard in terms of model A, just because that was the clearest to specify (used the fewest hypothetical constructs) because the basic source

character set is a well-defined finite set.

The situation is not the same for C given the already existing text for the standard, which allows multibyte characters to appear almost anywhere (the most notable exception being in identifiers), and given the more low-level (or “close to the metal”) nature of some uses of the language.

Thus C++ was specified in the most definite terms available at the time, with the supposition that other approaches could implement the same behavior.

2. Motivation

There are two main problems with the current specification of UCNs, which remains essentially the same as initially standardized.

1. Undefined behavior is an undesirable kind of specification, because it technically allows the implementation to quietly produce a defective executable file. There is consensus that errors in preprocessor usage should not have runtime consequences. There is even a CERT security advisory about UCNs causing undefined behavior. N3801 "Removing Undefined Behavior from the Preprocessor" directly addresses this by recommending to convert undefined behavior specifications to ill-formedness.

2. There are corner cases which are not specified as undefined behavior. These may be well-defined and misinterpreted, or ill-formed yet undiagnosed, by popular implementations, or simply excessively restrictive. Underspecified cases undermine the effect of N3801 by remaining wrongly defined.

The present paper will review several corner cases, considering the natural response of models A, B, and C, and the actual behavior of GCC and Clang. This leads to a specification permitting implementation flexibility without ambiguity.

3. Unstandardized corner cases

These cases seem to be unaccounted for in the current standard. Some are the subject of registered defect reports.

3.1. Escaped UCN

Within a character or string literal, a backslash preceding the sequence of characters that would otherwise form a UCN forms the escape sequence for a backslash, followed by the "U" and several hexadecimal digits.

This also occurs as a side effect of the stringize (#) operator applied twice to an identifier containing a UCN, which limits implementation latitude to use different internal encodings in strings and identifiers. This is the subject of CWG DR 1335.

```
      "\\u00aF" // translates same as R"(\u00aF)", not R"(\ï)".
```

Model A implementations will translate the escape sequence as specified. Model C suggests that the implementation will see an escape sequence containing an extended character, but the grammar specification does not allow for such a thing. There is no provision for diagnosis, and no possibility of recovering the original capitalization and quantity of digits unless the UCN-to-native translation is undone e.g. by re-reading the source. Model B could go either way, depending on the character.

3.2. Escaped basic source UCN

An implementation is required by [lex.phases] §2.2/1 to behave the same whether an extended character is translated to or specified as a UCN. Basic source characters may also be specified as UCNs but the requirement does not apply. There appears to be no other relevant specification.

```
      "\\u006e" // translates same as R"(\u006e)", not "\n".
```

Model A interprets this the same as the previous case. Because the allowance to use an internal encoding is normative, it may give permission to models B and C to convert the UCN to native format and form a newline escape sequence. If so, the specification is ambiguous.

If this example is unconvincing, consider that one backslash may be added by stringizing.

3.3. Escaped extended character

A backslash preceding an extended character is required to behave the same as an escaped UCN. The implementation must define its own translation of extended characters to UCNs, but it is not in general a documented (implementation-defined) behavior. Therefore the user generally cannot know what to expect. This is the subject of CWG DR 578.

```
"\ï" // May translate same as R"(\u00EF)", R"(\U000000eF)", etc.
```

Model A is required to behave somewhat unpredictably. Models B and C must generate a UCN from the internal encoding, contrary to their overall design, to emulate model A.

3.4. Commented control character

A control character (outside the basic source character set) appearing in a comment must be replaced by a UCN, as phase 1 cannot distinguish between comments and tokens. The intent is probably that it be replaced by the spelling of a UCN, since *universal-character-name* is a lexical production only appearing within identifiers and literals. Nevertheless, the spec appears to require that the translation occur, that it form an actual UCN, and that this UCN be diagnosed as illegal. This is the subject of CWG DR 1403.

```
/* ^G */ /* Implementation must diagnose, or document non-
diagnosis in terms of text decoding. */
```

Such diagnosis could be implemented in models A or B as part of phase 1, while generating UCNs. The corresponding strategy under model C could mis-diagnose `/* \u0007 */`.

3.5. UCN formed by adjacency

The stringize operator (#) provides an opportunity to form a UCN. An invalid UCN requires diagnosis.

```
#define bs() \ /* space */
#define s_lit(x) #x
#define s(x) s_lit(x)

s( bs()udddd ) // Error: surrogate code point
```

Models A and B are likely to catch such an error, and conversely to correctly process such a UCN if well-formed. Model C suggests that an implementation may not be prepared for such a dynamic UCN. (Other cases such cases are already specified as UB.)

3.6. Basic source UCN with _Pragma

The `_Pragma` operator takes a string, lexes its character sequence, and interprets the result in an implementation-specific way. The lexing is specified to occur before the implementation-specific behavior, so phase 3 diagnostic requirements apply.

```
_Pragma( "\u0063" ) // Error: basic source UCN in identifier
```

Model A is likely to catch such an error as long as it adheres to the spec, and avoids translating the string while stripping the quotes. Models B and C suggest an implementation that would either translate the UCN while forming the string, or never form a UCN in the first place.

3.7. Delayed raw string reversion

Concatenating a capital "R" before a string token causes it to become a raw string. UCNs are retroactively un-translated inside raw strings.

```
#define ucn_ï R ## "(ï)" // Produces R"(ï)", not R"(\u00AF)".
```

This required behavior is a consequence of the specification in [lex.pptoken] §2.5/3, which applies regardless of the provenance of the characters in a token.

> Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted …

This is impossible to implement reasonably in models A and C except by re-reading the source file. It is also not specific to UCNs. Line splicing and trigraphs in any tokens passed to a macro can potentially be reverted by stringizing and prepending an R.

Because the problem can be considered as an interaction between [lex.pptoken] §2.5/3 and [cpp.stringize] §16.3.2, and not particular to UCNs, it will not be treated further by this paper but deferred to another proposal.

3.8. UCN overflow

Unicode code points are by definition at most `\u0010FFFF`. The current specification specifically forbids surrogate pairs but seems to allow non-code points except for the requirement ([lex.charset] §2.3/2) that "The character designated by the universal-character-name \UNNNNNNNN is that character whose character short name in ISO/IEC 10646 is NNNNNNNN" and [lex.string] §2.14.5/15 "Within `char32_t` and `char16_t` literals, any universal-character-names shall be within the range 0x0 to 0x10FFFF." (Also note that the latter specifies character literal semantics in the string literal clause.) This is the subject of DR 1332.

```
L"\U00110000"; // Error: Numerically valid, but not a character.
u8"\U00110000"; // Validity may depend on version of UTF-8.
```

This does not relate to a particular model, and arguably it is already specified, but GCC and Clang treat this differently so the specification should be clearer.

In addition, "character short name" is not an official Unicode term and some UCNs seem to be allowed which are not characters according to Unicode. Moreover, noncharacters should be accessible to C++ for their intended "internal use" purpose. For example, a program may need to encode an internal-use codepoint such as `\uFFFF` in UTF-8 as an internal string delimiter.

3.9. Incomplete UCN

A `\u` or `\U` sequence not followed by the appropriate number of hexadecimal digits fails to form a UCN. In a string or character, it is an illegal escape code and must be diagnosed. In an identifier, the lexer is required to backtrack and commit the preceding characters as a complete identifier, then the backslash as its own token, and the characters starting with `u` as another identifier. The result is nonsense. Most users would prefer to see a diagnosis, which should be required. In diagnosing this as an error, EDG is nonconforming.

3.10. Special characters

Several codepoints are permissible in identifiers which are supposed to be reserved to the text editor. These include the byte order mark and the "replacement character" which may stand for a failure to translate some other character. From the Unicode 7 Standard:

> The Specials block contains code points that are interpreted as neither control nor graphic characters but that are provided to facilitate current software practices.

These characters are not even supposed to be part of a text stream. Failure to diagnose their presence is likely to cause deleterious interactions between fully conforming compilers and text editors which introduce and strip such characters in the intended ways, in particular during copy-paste operations from fully rendered sources.

4. Corner cases already specified

For completeness, this section reviews undefined behavior specified by [lex.phases] §2.2/1. The specification of undefined behavior is the subject of CWG DR 787 and N3801.

4.1. Line splicing

Generating a UCN in phase 2 could confuse a model C implementation that converts UCNs to extended characters in phase 1, and never handles them afterward. The subject of CWG DR 1775 is that such line splices may invoke UB before being reverted by a raw string literal context.

```
\u0\
0EF // UB per current spec
```

4.2. Token concatenation

Concatenating a backslash with an escape code potentially forms an identifier beginning with a UCN. This only works if one complete operand is a stray backslash token.

```
#define ucn_ï \ ## u00EF // UB per current spec
```

In fact the specification states "If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.3.3), the behavior is undefined." This does not consider whether the source characters came from separate tokens. Any UCN in an operand renders a concatenation undefined. These cases should be allowed.

```
#define GUARD_NAME ï ## _GUARD // UB per current spec
#define COLUMN "ï" ## _column // UB per current spec
```

5. Analysis

The foregoing list of cases is not intended to be complete, but to illustrate the variety of things that can go wrong. Essentially any preprocessor operation has some interaction with UCNs, with well-defined (or worse, ambiguously defined) behavior according to model A that is difficult to implement in another model.

N3801 suggests to resolve the undefined behavior problem by requiring diagnoses instead. This implies trapping of special cases, which tends to require each model to emulate the others. Model A may not recognize UCNs until phases 5 and 7, model C may translate them in phase 1 or 3, and model B is a hybrid. But diagnosis would require scanning for UCNs in phase 4, and moreover detecting new UCNs in the output that were not fully formed in the input. There may not be a general way to do this, but each above case would need code tailored to its effects. Again, we should presume that not all cases are known, and future language features will add more cases. Furthermore, if models B and C are required to recognize UCNs in phase 4 (and perhaps phase 2) in emulation of model A, it is just as easy to translate them as to emit a diagnosis.

Identical behavior across implementations is not necessary to guarantee that programs execute predictably, so implementation-specific semantics are a viable solution. A user is sufficiently protected if their implementation documents the output they can expect from any program, and the meaning of a program relying on conditional support may vary only in minor details, such as capitalization of implicit hexadecimal digits. Such specification is provided by conditional support, with limited implementation-defined semantics where necessary. We must nevertheless take care not to require diagnosis of conditional non-support that would be difficult to implement.

The following table extrapolates how implementations of the various models could conform to the specification in the next section. This will assume that model B opportunistically translates some UCNs in phase 3 but model C translates all in phase 1. Cases with potentially significant implementation effort are in boldface.

Case	Model A	Model B	Model C
3.1. Escaped UCN	Document support	Document supported UCN subset, unsupported as C	**Disable translation following backslash**
3.2. Escaped basic source UCN	Document support	Defer translation of basic source UCNs	**Disable translation following backslash**
3.3. Escaped extended character	Document UCN format	Document supported UCN subset, unsupported as C	Diagnose invalid escape code
3.4. Commented control character	Treat comments as "raw"	Treat comments as "raw"	No diagnosis
3.5. UCN formed by adjacency	Document support	Document supported UCN subset, unsupported as C	Diagnose invalid escape code
3.6. Basic source UCN with _Pragma	Diagnose	May diagnose	Do not diagnose
3.7, 3.8, 3.10	n/a	n/a	n/a
3.9. Incomplete UCN	Diagnose	Diagnose	Diagnose
4.1. Line splicing	Document support	Document support	Diagnose invalid escape code or stray backslash
4.2. Token concatenation	Document support	Document supported UCN subset, unsupported as C	Diagnose invalid escape code or stray backslash

The only cases in boldface appear under Model C, and reflect a feature required anyway to support innocuous strings such as `"C:\\udefaults"`: a phase 1 translator must be smart enough to recognize the `"\\"` escape sequence. The proposed specification thus reflects existing practice, and requires only documentary changes.

Clang and GCC (with the `-fextended-identifiers` option) behave almost identically: 3.5 and 3.6 are diagnosed correctly, 3.3 causes an invalid escape code diagnosis and does not spell out the UCN, 3.9 is diagnosed as a warning, and the other cases are processed with no diagnosis. Therefore they conform to the proposed specification, but do not conform to C++11 in cases 3.3 and perhaps 3.4, depending upon the meaning of the specification of translation phase 1. Under N3801, these current implementations would be non-conformant in cases 3.3, 4.1, and 4.2, and perhaps 3.4. Additionally, Clang diagnoses 3.8, but GCC does not.

6. Proposed wording

This specification intends to provide a clear continuum between conforming support for UCN manipulation, and conforming diagnosis of UCN abuse, following the principle of conditional support. A user may expect a corner case to work according to the conceptual model (corresponding to implementation model A), or otherwise to see a diagnosed error. Furthermore, the specific behavior of any implementation should be fully described by its documentation.

The present text also aims to resolve ambiguities in what UCNs may encode, and what constitutes a UCN.

These changes are relative to working draft N3797.

Amend [lex.phases] §2.2/1, in the specification of phase 1:

> Any source file character not in the basic source character set [lex.charset] is replaced by the spelling of a universal-character-name that designates that character.

Delete from [lex.phases] §2.2/1, in the specification of phase 2:

> ~~If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined.~~

Delete from [lex.phases] §2.2/1, in the specification of phase 4:

> ~~If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.3.3), the behavior is undefined.~~

Amend the second footnote in [lex.charset] §2.3/2:

> A sequence of characters resembling a universal-character-name in an r-char-sequence (2.14.5) or in a comment does not form a universal-character-name.

Modify [lex.charset] §2.3/2:

> ~~The character designated by the universal-character-name `\UNNNNNNNN` is that character whose character short name in ISO/IEC 10646 is `NNNNNNNN`; the character designated by the universal-character-name `\uNNNN` is that character whose character short name in ISO/IEC 10646 is `0000NNNN`. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed.~~ A universal-character-name `\UNNNNNNNN` shall designate the ISO/IEC 10646 Unicode scalar value with the short name `NNNNNNNN`; a universal-character-name `\uNNNN` shall designate the Unicode scalar value with the short name `0000NNNN`. *[Note:* Thus codepoints 0xD800 through 0xDFFF, inclusive, and values greater than 0x10FFFF, are ill-formed.*]*

Include a paragraph in [lex.charset] §2.3:

> A `\u` or `\U` source character sequence in a context where a universal-character-name may occur shall introduce a universal-character-name. Interpretation of the universal-character-name production by phase 3 except where produced by a sequence of characters which were consecutively output by phase 1 is conditionally-supported. *[Note:* A universal-character-name in a string or character literal is interpreted first by phase 3 while forming a token, before conversion by phase 5.*]* *[Note:* Such a universal-character-

> name may result from a line splice, token concatenation, the stringize operator, or the backslash-removing behavior of the `_Pragma` operator.*]* Treatment of a universal-character-name within a token as its constituent character sequence, and substitution of an extended character by phase 1 in a context where the universal-character-name production cannot occur, are conditionally-supported. The observable spelling of substitutions in these contexts is implementation-defined. *[Note:* For example, such occurs if the stringize operator adds a backslash before a universal-character-name, an extended character in a literal is prefixed by a backslash, or an extended character occurs in a comment.*]*

Delete from [lex.string] §2.14.5/15:

> ~~Within `char32_t` and `char16_t` literals, any universal-character-names shall be within the range 0x0 to 0x10FFFF.~~

Amend the index of implementation-defined behavior:

> mapping physical source file characters to the basic source character set and universal-character-names, where used, [lex.charset]
>
> preprocessing operations that may form a universal-character-name, if any, [lex.charset]

Modify [charname] §E.1:

> `FE47-FEFE, FF00-FFEF` ~~`FFFD`~~

7. Acknowledgements

Walter Brown provided helpful feedback and suggestions on the initial draft of this paper.

8. Revision history

N3881 — Initial revision.

N4219 — Account for incomplete UCNs.

Remove changes to [cpp.pragma.op] as redundant.

Forbid specials including BOM and REPLACEMENT CHARACTER.

Remove emptied “further work” section.