Proposal for C23

WG14 n2777

Title:	C Identifier Syntax using Unicode Standard Annex 31
Author, affiliation:	Robert C. Seacord, NCC Group, USA rcseacord@gmail.com Steve Downey, Bloomberg, USA <sdowney@gmail.com, sdowney2@bloomberg.net> Jens Gustedt, INRIA, France <jens.gustedt@inria.fr> Peter Bindels, TomTom, Netherlands, <dascandy@gmail.com>
Date:	2021-6-24
Proposal category:	Feature
Target audience:	Implementers
Abstract:	Adopt Unicode Annex 31 as part of C23
Prior art:	C++

C Identifier Syntax using Unicode Standard Annex 31

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: **n2777**

Reference Document: P1949R7 (http://wg21.link/p1949)

Date: 2021-7-03

This paper is a rewrite of P1949R7 (adopted by WG21 for C++23 and retroactive to previous releases) to address similar concerns in the C language and to promote interoperability between the two languages.

Unicode® Standard Annex #31 Unicode Identifier and Pattern Syntax has been adopted by C++, Rust, and Python. We propose adopting Unicode Annex 31 as part of C23 to address the problems identified in this paper and to maintain compatibility with C++ and other languages.

- That C identifiers match the pattern (**`XID_Start`** + **`_`**) + **`XID_Continue`***.
- That portable source is required to be normalized as NFC.
- That using unassigned code points is a constraint violation.

Change Log

2021-7-03:

· Initial version

1. PROBLEM DESCRIPTION

The allowed Unicode code points in identifiers include many that are unassigned or unnecessary, and others that are actually counter-productive. The current approach is defective in that invalid identifier code points (such as right-to-left modifiers) were included.

By adopting the recommendations of UAX #31, Unicode Identifier and Pattern Syntax, C will be easier to work with in international environments and less prone to accidental problems. It also aligns the C language with other current languages such as C++, Java, Python 3, Erlang, Rust, and JavaScript that defer to Unicode UAX #31 for identifier syntax. Adopting UAX 31 as the base allows C to work with a more sensible list of code points today and defer to experts for the future expansions of Unicode.

This proposal does not address some potential security concerns—so called homoglyph attacks—where letters that appear the same may be treated as distinct. Methods of defense against such attacks are complex and evolving, and requiring mitigation strategies would impose substantial implementation burden.

This proposal also recommends adoption of Unicode normalization form C (NFC) for identifiers to ensure that when compared, identifiers intended to be the same will compare as equal. Legacy encodings are generally naturally in NFC when converted to Unicode. Most tools will, by default, produce NFC text.

Some scripts require the use of characters such as joiners that are not allowed by base UAX #31, these will no longer be available as identifiers in C. Anecdotally this is similar to the English word "won't" being unavailable. There are workarounds that are unobjectionable to working programmers.

As a side-effect of adopting the identifier characters from UAX #31, using emoji in or as identifiers becomes ill-formed. Emoji, as a category, did not exist when C and the ranges of allowed characters were specified. The assigned characters that were in existence when the character ranges were standardized that became emoji were excluded from identifiers. As a result many emoji and emoji modifiers are disallowed today, so the status quo is broken. Allowed emoji are allowed because all unassigned code points were allowed.

What will this proposal change?

All emoji become excluded, instead of just some

Emoji with code points less than FFFF, such as ✌, and ❤ are currently excluded from identifiers. Signs, symbols, and color blocks are also excluded, meaning that many emoji sequences are invalid, for example:

```
bool 👷 = false; // Female Construction Worker
                 // ({Construction Worker}{ZWJ}{Female Sign})
```

but this is valid:

```
  bool 👷 = true; //  (Male) Construction Worker
```

Other oddities of disallowed vs allowed:

```
int ⏰ = 0; // invalid
```

```
int 🕐 = 0;
```

```
int ☠ = 0; // invalid
```

```
int 💀 = 0;


int ✋ = 0; // invalid

int 👊 = 0;


int ✈ = 0; // invalid

int 🚀 = 0;


int ☹ = 0; // invalid

int 😀 = 0;
```

Compiler Explorer Link

Zero Width Joiner and Zero Width Non-Joiner become excluded.

Some words in some scripts, such as Persian, Malayalam, and Sinhala, require the use of zero width joiners and non-joiners to render properly. These words will no longer be valid identifiers when using the zero-width joiners. Some stylistic ligatures in English also require zero width joiners and non-joiners, although it would be extremely unusual to find such code.

The available set of identifiers changes over time

As Unicode is extended, additional characters become available for use in identifiers that older compilers may not permit. However, this is mitigated as all common use scripts are already encoded in Unicode. The current situation is that all unassigned characters are allowed. UAX #31 assures that the set of allowed characters in each successive version stays the same or is expanded; at no point will valid programs become invalid when following it.

What will this proposal not change?

The validity of "extended" characters in identifiers

All current compilers allow characters outside the basic source character set directly in source today.

Why now

One driving factor for addressing this now is that GCC has fixed their long standing bug 67224 "UTF-8 support for identifier names in GCC". Clang has always supported all the allowed code points in source code. MSVC in its usual configuration defaults to code page 1252, but can be told to accept

UTF-8 source. With GCC now allowing it, the barrier to use of Unicode characters outside the basic source character set has dropped considerably. Use of characters via universal character names was always possible, but never widely used. Examples found in the wild of use of UCNs in identifiers come from compiler and related tool test suites.

Restricting the profile of characters is much easier if no one is depending on them.

Identifiers

An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter.

There are few names that are not identifiers, but in particular headers and header files are not named by identifiers. The introduction of a raw string literal is constrained to be composed of members of the basic source character set, and does not follow the identifier grammar.

Addressing identifiers in a more principled ways

UNICODE IDENTIFIER AND PATTERN SYNTAX [UAX31] is an attempt to provide a normative way of specifying definitions of general-purpose identifiers for use in programming languages. It has evolved significantly over the years, in particular since the time that C11 was specified. In particular, the characters that were allowed as identifiers, and the patterns, were not stable at the time of C11. In addition, at that time, ISO was promulgating advice suggesting a list of code points as the recommended method for ISO standards to specify identifiers.

Today the definitions in UAX31 can be used to provide stable definitions for programming language identifiers, with guarantees that an identifier will not be invalidated by later standards.

Originally, UAX31 relied on derived properties of characters, **`ID_Start`** and **`ID_Continue`**, however those properties relied on fundamental properties that could change over time. The Unicode database now provides **`XID_Start`** and **`XID_Continue`**, based on the same characteristics, but with an additional stability guarantee. The Unicode database now provides explicit classification of both. [UAX44]

The original definitions closely match the identifier syntax of C:

Properties	General Description of Coverage
`ID_Start`	**`ID_Start`** characters are derived from the Unicode **`General_Category`** of uppercase letters, lowercase letters, titlecase letters, modifier letters, other letters, letter numbers, plus

	Other_ID_Start, minus **Pattern_Syntax** and **Pattern_White_Space** code points. In set notation: [\p{L}\p{Nl}\p{Other_ID_Start}-\p{Pattern_Syntax}-\p{Pattern_White_Space}]
ID_Continue	**ID_Continue** characters include **ID_Start** characters, plus characters having the Unicode General_Category of nonspacing marks, spacing combining marks, decimal number, connector punctuation, plus **Other_ID_Continue** , minus **Pattern_Syntax** and **Pattern_White_Space** code points. In set notation: [\p{ID_Start}\p{Mn}\p{Mc}\p{Nd}\p{Pc}\p{Other_ID_Continue}-\p{Pattern_Syntax}-\p{Pattern_White_Space}]

The **XID_** versions of the properties started with the same elements, but are guaranteed stable in that the property values for an assigned code point will not change after assignment in subsequent Unicode standards.

Issues with base UAX31

XID_Continue does not include layout and format control characters

Some scripts require layout or format characters, such as Zero Width Joiner (U+200D) and Zero Width Non-Joiner (U+200C) to correctly render some words. UAX31 presents examples from Persian, Malayalam, and Sinhala. This does not mean that those scripts are entirely excluded, but that certain words cannot have their preferred spelling. This is similar to being unable to use "can't" or "won't" as identifiers, as their spelling requires the punctuation character Apostrophe (U+0027).

UAX31 outlines a mechanism in which identifiers containing only characters from affected scripts may allow formatting and control characters. This paper does not propose adopting that mechanism, as it requires access to the full Unicode database. In addition, UAX31 says that identifiers should compare the same with and without those characters, leading to additional complication as identifier comparison may be performed by the linker.

This paper also does not propose excluding any scripts categorically, regardless of their status as historic or obsolete. Characters from Anatolian Hieroglyphs would be available for use, to the extent that anyone wishes to do so.

Does not exclude homoglyph attack

Homoglyph attacks, where visually indistinguishable characters from different scripts are used to create confusion, such as between latin letter c and cyrillic letter c. This is covered by Unicode Technical Report #36 UNICODE SECURITY CONSIDERATIONS [UAX36]. It requires much more extensive analysis of text, using the full Unicode database, and for a compiled language would provide limited benefit.

Does not allow emoji

Currently allowed emoji are incomplete. Emoji with code points less than FFFF are excluded, such as ✌, and ❤. Signs, symbols, and color blocks are also excluded, meaning that many sequences are valid, for example:

```
bool 👷 = true; //  Construction Worker
```

but this isn't:

```
bool 👷 = false; // Woman Construction Worker

 // ({Construction Worker}{ZWJ}{Female Sign})
```

The Female Sign, ♀, has always been excluded from C identifiers, but is required to construct extended emoji sequences. The Female Sign code point is used to modify the display of a base emoji—which is in theory gender neutral. Allowing the full range of emoji would require significant work, revisiting which code points that are currently excluded for potential inclusion, as well as making the zero width joiner contextually available. This proposal extends the status quo of which classes of characters are allowed to the code points that have been added since 1999. UAX31 uses essentially the same criteria for characters allowed in identifiers as was used to generate the list of code points in §6.4.2, expressed in modern Unicode terms, and maintained as part of the Unicode Standard.

Adding true emoji support, including gender and skin tone modification, is complex, and if it is desired should be addressed comprehensively and intentionally. The status quo of emoji support is an accident. For example, testing if a sequence of code points is a valid emoji sequence is fairly complicated. UNICODE EMOJI [UTS51] currently has a regex to determine if a sequence *might* be valid, and there are no stability guarantees:

```
\p{RI} \p{RI}
| \p{Emoji}
  ( \p{EMod}
  | \x{FE0F} \x{20E3}?
  | [\x{E0020}-\x{E007E}]+ \x{E007F} )?
  (\x{200D} \p{Emoji}
     ( \p{EMod}
     | \x{FE0F} \x{20E3}?
     | [\x{E0020}-\x{E007E}]+ \x{E007F} )?
  )*
```

Emoji Sequences

This is insufficient for validity, merely testing that the sequence is not facially invalid.

Even if we were to adopt a subset, such as basic emoji as defined by UTS51, we would have to address the characters currently excluded from identifiers, and deal with stability issues where the Unicode standard is not making guarantees.

History

Using an explicit list of Unicode characters was considered a best practice for ISO standardization in TR 10176:2003 - Guidelines for the preparation of programming language standards.

National body comment CA 24 for C++11

> A list of issues related TR 10176:2003:
>
> - "Combining characters should not appear as the first character of an identifier." Reference: ISO/IEC TR 10176:2003 (Annex A) This is not reflected in FCD.
> - Restrictions on the first character of an identifier are not observed as recommended in TR 10176:2003. The inclusion of digits (outside of those in the basic character set) under identifier-nondigit is implied by FCD.
> - It is implied that only the "main listing" from Annex A is included for C++. That is, the list ends with the Special Characters section. This is not made explicit in FCD. Existing practice in C++03 as well as WG 14 (C, as of N1425) and WG 4 (COBOL, as of N4315) is to include a list in a normative Annex.
> - Specify width sensitivity as implied by C++03: \uFF21 is not the same as A. Case sensitivity is already stated in [lex.name].

N3146 in 2010-10-04 considered using the Default Identifier Syntax [N3146] of UAX #31, but at the time there were stability issues with identifiers, and instead came down on the side of using the Alternative Identifier Syntax [AltId] allowing all but smaller ranges of characters, including unassigned code points. The Alternative Identifier Syntax is now referred to as Immutable Identifiers in UAX #31.

From N3146:

> The set of UCNs disallowed in identifiers in C and C++ should exactly match the specification in [AltId], with the following additions: all characters in the Basic Latin (i.e., ASCII, basic source character) block, and all characters in the Unicode General Category "Separator, space".

The Unicode standard has since made additional stability guarantees about identifiers. The **`XID_Start`** and **`XID_Continue`** properties, originally provided for improved NFKC normalization, now also have stability over the **`ID_Start`** and **`ID_Continue`** properties that they are based on. This addresses the concerns in TR 10176:2003.

Normalization Discussion

Comparing Unicode strings can be complicated because there may be many ways of encoding a grapheme cluster that appears to be the same, and in fact may be canonically the same. There are combining characters such that there are two or more ways of, e.g., spelling À. It could be spelled as either 'LATIN CAPITAL LETTER A WITH GRAVE' (U+00C0), or as the combination of 'LATIN CAPITAL LETTER A (U+0041)' and 'COMBINING GRAVE ACCENT' (U+0300).

```
const int À = 1; // U+00C0
const int À = 2; // U+0041 U+0300
const int gv1 = \u00c0;
const int gv2 = A\u0300;
static_assert(gv1 == 1);
static_assert(gv2 == 2);
```

Compiler Explorer Link

Although changing the comparison rules could break code, this is a case that is frustrating for working programmers and removing the ability to spell À two different ways is not something that should be relied upon.

According to UAX31, the Unicode Normalization Form C is the most appropriate form for string equivalence checks for identifiers in languages that do not case fold, and SG16 [WG21] agrees on this. NFC compares based on combining all characters into canonical forms, so that characters that are canonically the same, such as the Angstrom sign and A with ring, are folded into a single code point for comparison. As many editors will do this automatically, and there is no way of visually detecting the difference, canonical equivalence is the most appropriate form to check.

However, linkers will compare identifiers by octets.

Therefore, normalization of identifiers must happen before object files are emitted. It is either the responsibility of programmers to provide normalized identifiers, and compilers to either warn or error on non-normalized ones, or for compilers to normalize Unicode input. For non-Unicode input, the transformation to normalized form is the simple table based lookup used to translate now.

Detection of un-normalized text is fairly straight-forward, and GCC 10 already produces a warning. Unicode Annex 15, Unicode Normalization Forms, provides a quick check algorithm to test if a string is

in one of the normalization forms, driven by tables in the Unicode database. See Detecting_Normalization_Forms in [UAX15]. The tables are available at DerivedNormalizationProps.txt. The check algorithm will sometimes need to normalize short ranges of text where detection of YES or NO is not possible for the single code point.

The preprocessor must also compare tokens by string matching, and modular header units mean this is also a cross translation unit concern. The preprocessor can also concatenate strings via pasting. We do not expect the preprocessor to normalize on concatenation, however nonetheless the results of concatenation used as an identifier shall be in NFC form.

There is implementation divergence on how the concat operator, ##, works with combining characters. The code below is flagged as an error today by GCC, it is accepted by Clang and MSVC.

```
#define accent(x) x##\u0300

const int accent(A) = 2;
const int gv2 = A\u0300;
static_assert(gv2 == 2, "whatever");
```

Compiler Explorer Link

The ability to form valid identifiers via token pasting with combining characters is not a goal of this paper.

Conformance Points for UAX31

UAX31-C2. An implementation claiming conformance to this specification shall describe which of the following requirements it observes:

- UAX31-R1. Default Identifiers
- UAX31-R1a. Restricted Format Characters
- UAX31-R1b. Stable Identifiers
- UAX31-R2. Immutable Identifiers
- UAX31-R3. Pattern_White_Space and Pattern_Syntax Characters
- UAX31-R4. Equivalent Normalized Identifiers
- UAX31-R5. Equivalent Case-Insensitive Identifiers
- UAX31-R6. Filtered Normalized Identifiers
- UAX31-R7. Filtered Case-Insensitive Identifiers
- UAX31-R8. Hashtag Identifiers

UAX31 Requirements Met

- UAX31-R1 : via a profile
- UAX31-R4 : via requiring NFC normalization

UAX31 Requirements Not Met

- UAX31-R1a : Format characters are not allowed
- UAX31-R1b : Identifiers are not stable as defined by UAX31
- UAX31-R2 : Identifiers are not immutable over time
- UAX31-R3 : C does not need identifiers to describe patterns
- UAX31-R5 : C is case sensitive
- UAX31-R6 : No filtering is performed
- UAX31-R7 : No filtering is performed
- UAX31-R8 : Hashtags are not relevant

Details of conformance

R1 Default Identifiers

C meets UAX31-R1 by adopting a profile adding LOW LINE to the set of allowed start characters. In the terms of the grammar used in UAX31:

```
<Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*

<Start> := XID_Start + U+005F
<Continue> := <Start> + XID_Continue
<Medial> :=
```

R1a Restricted Format Characters

Format characters are not allowed in identifiers.

R1b. Stable Identifiers

No explicit stability guarantees beyond what is provided by UAX31.

R2. Immutable Identifiers

Identifiers are not immutable. Additional identifiers may be available in the future.

R3. Pattern_White_Space and Pattern_Syntax Characters

C does not describe character patterns as part of the language, deferring to library components.

R4. Equivalent Normalized Identifiers

Identifiers that compare the same under NFC are equivalent.

R5. Equivalent Case-Insensitive Identifiers

There are no case insensitive comparisons or case folding.

R6. Filtered Normalized Identifiers

No filtering is performed, and all characters shall be normalized in identifiers.

R7. Filtered Case-Insensitive Identifiers

C is case sensitive.

R8. Hashtag Identifiers

There are no hashtag identifiers

Phases of Translation

Identifiers used in phase 4 preprocessing directives and macro invocations shall be in Normalization Form C. Tokens that are not identifiers, such as pp-numbers, are not required to be normalized. Note that well formed integer and floating point literals are inherently normalized due to the allowed characters. The terms identifier-start and identifier-continue are added to match the Unicode character classes **`XID_Start`** and **`XID_Continue`**.

Identifiers in phase 7 of translation shall be in Normalization Form C.

Wording for UAX #31 identifiers

These changes should be applied to the N2596 working draft — December 11, 2020.

Add two entries in Section 2 following paragraph 7

The Unicode Consortium. Unicode Standard Annex, UAX #44, *Unicode Character Database* [online]. Edited by Ken Whistler and Laurențiu Iancu. Available at http://www.unicode.org/reports/tr44/

The Unicode Consortium. The Unicode Standard, Derived Core Properties. Available at https://www.unicode.org/Public/UCD/latest/ucd/DerivedCoreProperties.txt

Change in Section 6.4 paragraphs 1:

preprocessing-token:

header-name

identifier

pp-number

character-constant

string-literal

punctuator

each universal-character-name that cannot be one of the above

each non-white-space character that cannot be one of the above

Change in Section 6.4 paragraphs 2:

Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator. A single *universal-character-name* shall match one of the other preprocessing token categories.

Change in Section 6.4 paragraphs 3:

A token is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single *universal-character-names* and single non-white-space characters that do not lexically match the other preprocessing token categories.[73)] If a' or a " character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (described later), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

Change in §6.4.2.1 paragraph 1:

identifier:

~~*identifier-nondigit*~~ *identifier-start*

identifier ~~*identifier-nondigit*~~ *identifier-continue*

~~*identifier-digit*~~

~~*identifier-nondigit:*~~

~~*nondigit*~~

~~*universal-character-name*~~

~~*other implementation-defined characters*~~

identifier-start:

nondigit

universal-character-name of class XID_Start

identifier-continue:

digit

nondigit

universal-character-name of class XID_Continue

nondigit: one of

_ a b c d e f g h i j k l m

n o p q r s t u v w x y z

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Change in §6.4.8 paragraph 1:

pp-number:

digit

. digit

~~*pp-number digit*~~

~~*pp-number identifier-nondigit*~~

pp-number identifier-continue

pp-number **e** *sign*

pp-number **E** *sign*

pp-number **p** *sign*

pp-number **P** *sign*

Modify §6.4.2.1 paragraph 2:

Constraints

An *identifier* shall conform to Normalization Form C as specified in ISO/IEC 10646.[yy] ~~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.[78]~~

[yy] Annex D provides an overview of the conforming identifiers.

~~78)The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF-16).~~

Replace §6.4.2.1 paragraph 3 with:

The character classes XID_Start and XID_Continue are Derived Core Properties as described by UAX #44.[nn]An implementation may allow additional universal character names and multibyte characters that are not part of the basic source character set to appear in identifiers in the positions of characters from the sets XID_Start and XID_Continue; which universal character names and which extended characters and their correspondence to universal character names is implementation-defined.

[nn] On systems that cannot accept extended characters in external identifiers, an encoding of the *universal-character-name* may be used in forming such identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the \u in a *universal-character-name*.

NOTE 1 Upper- and lower-case letters are considered different for all identifiers.

NOTE 2 In translation phase 4, the term identifier also includes those preprocessing tokens (6.4.8) differentiated as keywords (6.4.1) in the later translation phase 7.

Add an entry to the Bibliography

The Unicode Consortium. Unicode Standard Annex, UAX #31, *Unicode Identifier and Pattern Syntax* [online]. Edited by Mark Davis. Revision 33; issued for Unicode 13.0.0. 2020-02-13 [viewed 2020-05-27]. Available at https://www.unicode.org/reports/tr31/tr31-33.html

Replace Annex D

As the normative requirements are delegated to a different international standard, the previous version of normative Annex D becomes obsolete with these changes. We propose to replace it by an informative summary of the features proposed in UAX #31

Annex D (informative) Universal character names for identifiers

D.1

This subclause describes the choices made in application of UAX #31 (“Unicode Identifier and Pattern Syntax”) to C of the requirements from UAX #31 and how they do or do not apply to C. For UAX #31, C conforms by meeting the requirements R1 "Default Identifiers" and R4 "Equivalent Normalized Identifiers". The other requirements, also listed below, are either alternatives not taken or do not apply to C.

D.2 R1 Default Identifiers

UAX #31 specifies a default syntax for identifiers based on properties from the Unicode Character Database, UAX #44. The general syntax is

```
<Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
```

where <Start> has the XID_Start property, <Continue> has the XID_Continue property, and <Medial> is a list of characters permitted between continue characters. For C we add the character U+005F, LOW LINE, or _, to the set of permitted Start characters, the Medial set is empty, and the Continue characters are unmodified. In the grammar used in UAX #31, this is

```
<Identifier> := <Start> <Continue>*
<Start> := XID_Start + U+005F
<Continue> := <Start> + XID_Continue
```

This is described in the C grammar (6.4.2.1), where *identifier* is formed from *identifier-start* or *identifier* followed by *identifier-continue*.

D.2.1 R1a. Restricted Format Characters

If an implementation of UAX #31 wishes to allow format characters such as ZERO WIDTH JOINER or ZERO WIDTH NON-JOINER it must define a profile allowing them, or describe precisely which combinations are permitted.

C does not allow format characters in identifiers, so this does not apply.

D.2.2 R1b. Stable Identifiers

An implementation of UAX #31 may choose to guarantee that identifiers are stable across versions of the Unicode Standard. Once a string qualifies as an identifier it does so in all future versions.

C does not make this guarantee, except to the extent that UAX #31 guarantees the stability of the XID_Start and XID_Continue properties.

D.3 R2. Immutable Identifiers

An implementation may choose to guarantee that the set of identifiers will never change by fixing the set of code points allowed in identifiers forever.

C does not choose to make this guarantee. As scripts are added to Unicode, additional characters in those scripts may become available for use in identifiers.

D.4 R3. Pattern_White_Space and Pattern_Syntax Characters

UAX #31 describes how languages that use or interpret patterns of characters, such as regular expressions or number formats, may describe that syntax with Unicode properties.

C does not do this as part of the language, deferring to library components for such usage of patterns. This requirement does not apply to C.

D.5 R4. Equivalent Normalized Identifiers

UAX #31 requires that implementations describe how identifiers are compared and considered equivalent.

C requires that identifiers be in Normalized Form C and therefore identifiers that compare the same under NFC are equivalent. This is described in subclause 6.4.2.

D.6 R5. Equivalent Case-Insensitive Identifiers

C considers case to be significant in identifier comparison, and does not do any case folding. This requirement does not apply to C.

D.7 R6. Filtered Normalized Identifiers

If any characters are excluded from normalization, UAX #31 requires a precise specification of those exclusions.

C does not make any such exclusions.

D.8 R7. Filtered Case-Insensitive Identifiers

C identifiers are case sensitive, and therefore this requirement does not apply.

D.9 R8. Hashtag Identifiers

There are no hashtags in C, so this requirement does not apply.

NL 029

Comment

Allowed characters include those from U+200b until U+206x; these are zero-width and control characters that lead to impossible to type names, indistinguishable names and unusable code & compile errors (such as those accidentally including RTL modifiers).

Proposed Change

Disallow invisible characters in this range

4.0 Acknowledgements

I would like to recognize the following people for their help with this work: Zach Laine, Tom Honermann, Peter Bindels, Jens Maurer, and Aaron Ballman.

5.0 References

[AltId] Unicode Standard Annex.

http://www.unicode.org/reports/tr31/tr31-11.html#Alternative_Identifier_Syntax

[DefId] Unicode Standard Annex.

http://www.unicode.org/reports/tr31/tr31-11.html#Default_Identifier_Syntax

[N3146] Clark Nelson. 2010. Recommendations for extended identifier characters for C and C++.

https://wg21.link/n3146

[UAX15] Ken Whistler. Unicode Normalization Forms.

http://www.unicode.org/reports/tr15

[UAX31] Mark Davis. Unicode Identifier and Pattern Syntax.

http://www.unicode.org/reports/tr31

[UAX36] Mark Davis and Michel Suignard. Unicode Security Considerations.

http://www.unicode.org/reports/tr36

[UAX44] Ken Whistler and Laurențiu Iancu. Unicode Character Database.

http://www.unicode.org/reports/tr44

[UTS51] Mark Davis and Peter Edberg. Unicode Emoji.

http://www.unicode.org/reports/tr51