Reformulating Lexing in terms of unicode

Document #:
Date: 2020-06-09
Project: Programming Language C++
Audience:
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Intent of changes

- Make lossy conversion ill-formed as per P1854
- Mandate support for UFT-8 source files (N3463 [1])
- Mandate that unicode encoded source are not normalized in phase 1
- Mandate that conversion to unicode encoded literals doesn't normalize
- Specify that conversion to non-unicode encoded literals may result in a different number of code points
- Specify th behavior of a at the end of the file
- Specify what constitute a new line
- Universal character names are substituted to codepoints in phase 2, internal representation is unicode
- Apply changes from P2029

TODO:

- use more terminology from P1859
- remove remaining reference to basic source charatacter set

Issue fixed

- CWG1332 [4]
- CWG1655 [5]
- CWG1335 [6]
- CWG578 [7]
- CWG1403 [3]
- CWG411 [2]

Wording

🔲 Syntax notation [syntax]

In the syntax notation used in this document, syntactic categories are indicated by *italic* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase "one of". If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript "$_{opt}$", so

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces.

Unless otherwise specified, the glyphs used to describe the syntax notation refer to codepoint in the Basic Latin Unicode block described in ISO/IEC 10646.

Names for syntactic categories have generally been chosen according to the following rules:

- *X-name* is a use of an identifier in a context that determines its meaning (e.g., *class-name*, *typedef-name*).
- *X-id* is an identifier with no context-dependent meaning (e.g., *qualified-id*).
- *X-seq* is one or more *X*'s without intervening delimiters (e.g., *declaration-seq* is a sequence of declarations).
- *X-list* is one or more *X*'s separated by intervening commas (e.g., *identifier-list* is a sequence of identifiers separated by commas).

🔲 Terms and definitions [intro.defs]

For the purposes of this document, the terms and definitions given in ISO/IEC 2382-1:1993, the terms, definitions, and symbols given in ISO 80000-2:2009, and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp
- IEC Electropedia: available at http://www.electropedia.org/

Terms that are used only in a small portion of this document are defined where they are used and italicized where they are defined.

🔲 multibyte character [defns.multibyte]

sequence of one or more ~~bytes~~ code units representing a member of the ~~extended character set of either the source or the execution environment~~ that characterś associated character set

[*Note:* The extended character set is a superset of the basic character set lex.charset. — *end note*]

Unicode [defns.unicode]

Unicode and Unicode character set refers to the character set described in ISO/IEC 10646.

Lexical conventions [lex]

Phases of translation [lex.phases]

The precedence among the syntax rules of translation is specified by the following phases.[1]

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. Any source file character not in the basic source character set is replaced by the *universal-character-name* that designates that character. 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* (e.g., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.

   If the physical source character is the Unicode character set, each code point in the source file is converted to the internal representation of that same code point. Codepoints that are surrogate codepoints or invalid code points are ill-formed.

   Otherwise, each abstract character in the source file is mapped in an implementation-defined manner to a sequence of Unicode codepoint representing the same abstract character. (introducing new-line characters for end-of-line indicators if necessary).

   An implementation may use any internal encoding able to represent uniquely any Unicode codepoint. If an abstract character in the source file is not representable in the Unicode character set, the program is ill-formed.

   An implementation supports source files representing a sequence of UTF-8 code units. Any additional physical source file character sets accepted are implementation-defined. How the the character set of a source is determined is implementation defined.

2. Each implementation-defined line termination sequence of characters is replaced by a new-line character (U+000A). Each instance of a backslash character (\) immediately followed by a new-line character or at the end of a file is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. Except for splices reverted in a raw string literal,

[1] Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

if a splice results in a character sequence that matches the syntax of a *universal-character-name*, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file.

Sequences of whitespace characters at the end of each line are removed.

Each *universal-character-name* is replaced by the unicode code point it designates.

3. The source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[2] Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* See the handling of < within a `#include` preprocessing directive. — *end example*]

4. Preprocessing directives are executed, macro invocations are expanded, and `_Pragma` unary operator expressions are executed. If a character sequence that matches the syntax of a *universal-character-name* is produced by token concatenation, the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each basic source character set member in a *character-literal* or a *string-literal*, as well as each escape sequence and *universal-character-name* in a *character-literal* or a non-raw string literal, is converted to the corresponding member of the execution character set (**??**, **??**); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[3]

   Each codepoint in a *character-literal* or a *string-literal*, is converted to a sequence of code units in the literal's associated character encoding representing the same abstract character. If the abstract character is not representable in the literal associated character set, the program is ill-formed. If that literal's associated character encoding encodes the Unicode charatacter set, the converted sequence of code units represents the same code points as the internal representation.

   Each escape sequence in a *character-literal* or a non-raw string literal, is converted to the corresponding member of the execution character set.

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing

[2] A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing " or >. A partial comment would arise from a source file ending with an unclosed /* comment.

[3] ~~An implementation need not convert all non-corresponding source characters to the same execution character.~~

token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens. — *end note*] It is implementation-defined whether the sources for module units and header units on which the current translation unit has an interface dependency (**??**, **??**) are required to be available. [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]

8. Translated translation units and instantiation units are combined as follows: [*Note:* Some or all of these may be supplied from a library. — *end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* This may include instantiations which have been explicitly requested. — *end note*] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [*Note:* An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — *end note*] All the required instantiations are performed to produce *instantiation units*. [*Note:* These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — *end note*] The program is ill-formed if any instantiation fails.

9. All external entity references are resolved. Library components are linked to satisfy external references to entities not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

🔲 Character sets [lex.charset]

The *basic source character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:[4]

```
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
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

The *universal-character-name* construct provides a way to name other characters.

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

[4] ~~The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.~~

universal-character-name:
\u *hex-quad*
\U *hex-quad hex-quad*

A *universal-character-name* designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of *hexadecimal-digit* s in the *universal-character-name*. The program is ill-formed if that number is not a code point or if it is a surrogate code point. Noncharacter code points and reserved code points are considered to designate separate characters distinct from any ISO/IEC 10646 character. If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal* (in either case, including within a *user-defined-literal*) corresponds to a ~~control character or to a character in the basic source character set~~ character in the Basic Latin Unicode Block, the program is ill-formed.[5] [*Note:* ~~ISO/IEC 10646 code points are integers in the range $[0, 10FFFF]$ (hexadecimal).~~ A surrogate code point is a value in the range $[D800, DFFF]$ (hexadecimal). ~~A control character is a character whose code point is in either of the ranges $[0, 1F]$ or $[7F, 9F]$ (hexadecimal).~~ — *end note*]

The *basic character set* contains the following characters:

```
   U+0000 NULL
   U+0007 BELL
   U+0008 BACKSPACE
   U+0009 CHARACTER TABULATION
   U+000A LINE FEED
   U+000B LINE TABULATION
   U+000C FORM FEED
   U+000D CARRIAGE RETURN
   U+0020 SPACE
!  U+0021 EXCLAMATION MARK
"  U+0022 QUOTATION MARK
#  U+0023 NUMBER SIGN
%  U+0025 PERCENT SIGN
&  U+0026 AMPERSAND
'  U+0027 APOSTROPHE
(  U+0028 LEFT PARENTHESIS
)  U+0029 RIGHT PARENTHESIS
*  U+002A ASTERISK
+  U+002B PLUS SIGN
,  U+002C COMMA
-  U+002D HYPHEN-MINUS
.  U+002E FULL STOP
/  U+002F SOLIDUS
0  U+0030 DIGIT ZERO
1  U+0031 DIGIT ONE
2  U+0032 DIGIT TWO
3  U+0033 DIGIT THREE
4  U+0034 DIGIT FOUR
5  U+0035 DIGIT FIVE
6  U+0036 DIGIT SIX
```

[5] A sequence of characters resembling a *universal-character-name* in an *r-char-sequence* does not form a *universal-character-name*.

```
7  U+0037 DIGIT SEVEN
8  U+0038 DIGIT EIGHT
9  U+0039 DIGIT NINE
:  U+003A COLON
;  U+003B SEMICOLON
<  U+003C LESS-THAN SIGN
=  U+003D EQUALS SIGN
>  U+003E GREATER-THAN SIGN
?  U+003F QUESTION MARK
A  U+0041 LATIN CAPITAL LETTER A
B  U+0042 LATIN CAPITAL LETTER B
C  U+0043 LATIN CAPITAL LETTER C
D  U+0044 LATIN CAPITAL LETTER D
E  U+0045 LATIN CAPITAL LETTER E
F  U+0046 LATIN CAPITAL LETTER F
G  U+0047 LATIN CAPITAL LETTER G
H  U+0048 LATIN CAPITAL LETTER H
I  U+0049 LATIN CAPITAL LETTER I
J  U+004A LATIN CAPITAL LETTER J
K  U+004B LATIN CAPITAL LETTER K
L  U+004C LATIN CAPITAL LETTER L
M  U+004D LATIN CAPITAL LETTER M
N  U+004E LATIN CAPITAL LETTER N
O  U+004F LATIN CAPITAL LETTER O
P  U+0050 LATIN CAPITAL LETTER P
Q  U+0051 LATIN CAPITAL LETTER Q
R  U+0052 LATIN CAPITAL LETTER R
S  U+0053 LATIN CAPITAL LETTER S
T  U+0054 LATIN CAPITAL LETTER T
U  U+0055 LATIN CAPITAL LETTER U
V  U+0056 LATIN CAPITAL LETTER V
W  U+0057 LATIN CAPITAL LETTER W
X  U+0058 LATIN CAPITAL LETTER X
Y  U+0059 LATIN CAPITAL LETTER Y
Z  U+005A LATIN CAPITAL LETTER Z
[  U+005B LEFT SQUARE BRACKET
\  U+005C REVERSE SOLIDUS
]  U+005D RIGHT SQUARE BRACKET
^  U+005E CIRCUMFLEX ACCENT
_  U+005F LOW LINE
a  U+0061 LATIN SMALL LETTER A
b  U+0062 LATIN SMALL LETTER B
c  U+0063 LATIN SMALL LETTER C
d  U+0064 LATIN SMALL LETTER D
e  U+0065 LATIN SMALL LETTER E
f  U+0066 LATIN SMALL LETTER F
g  U+0067 LATIN SMALL LETTER G
h  U+0068 LATIN SMALL LETTER H
i  U+0069 LATIN SMALL LETTER I
j  U+006A LATIN SMALL LETTER J
k  U+006B LATIN SMALL LETTER K
```

```
l  U+006C LATIN SMALL LETTER L
m  U+006D LATIN SMALL LETTER M
n  U+006E LATIN SMALL LETTER N
o  U+006F LATIN SMALL LETTER O
p  U+0070 LATIN SMALL LETTER P
q  U+0071 LATIN SMALL LETTER Q
r  U+0072 LATIN SMALL LETTER R
s  U+0073 LATIN SMALL LETTER S
t  U+0074 LATIN SMALL LETTER T
u  U+0075 LATIN SMALL LETTER U
v  U+0076 LATIN SMALL LETTER V
w  U+0077 LATIN SMALL LETTER W
x  U+0078 LATIN SMALL LETTER X
y  U+0079 LATIN SMALL LETTER Y
z  U+007A LATIN SMALL LETTER Z
{  U+007B LEFT CURLY BRACKET
|  U+007C VERTICAL LINE
}  U+007D RIGHT CURLY BRACKET
~  U+007E TILDE
```

The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose value is 0.

For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. The *execution character set* and the *execution wide-character set* are implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

The *execution character set* and the *execution wide-character set* are implementation-defined supersets of the *basic character set* For each execution character set, the values of the members shall be non-negative and distinct from one another and the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. The value of the NULL character shall be 0. The values of the other members of the execution character sets and the sets of additional members are implementation-defined.

The *execution encoding* refers to the implementation-defined character encodings used to encode character and string literals in the *execution character set*.

The *execution wide-encoding* refers to the implementation-defined character encodings used to encode wide character and string literals in the *execution wide-character set*.

🔲 Preprocessing tokens [lex.pptoken]

preprocessing-token:
header-name
import-keyword
module-keyword
export-keyword
identifier
pp-number
character-literal
user-defined-character-literal
string-literal
user-defined-string-literal
preprocessing-op-or-punc
~~*each universal-character-name* that cannot be one of the above~~
each non-white-space ~~character~~ codepoint that cannot be one of the above

Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a literal, or an operator or punctuator.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, placeholder tokens produced by preprocessing `import` and `module` directives (*import-keyword*, *module-keyword*, and *export-keyword*), identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space ~~characters~~ codepoint that do not lexically match the other preprocessing token categories. 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, or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in **??**, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

If the input stream has been parsed into preprocessing tokens up to a given character:

- If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed ~~in phases 1 and 2 (*universal-character-names* and line splicing)~~ in phase 2 (*universal-character-name*s, line termination replacement, line splicing and whitespace trimming) are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern
  $_{opt}$*encoding-prefix* `R` *raw-string*
- Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessing token by itself and not as the first character of the alternative token `<:`.

- Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* is only formed
  - after the `include` or `import` preprocessing token in an `#include` or `import` directive, or
  - within a *has-include-expression*.

[*Example:*

```
#define R "x"
const char* s = R"y";           // ill-formed raw string, not "x" "y"
```

— end example]

The *import-keyword* is produced by processing an `import` directive, the *module-keyword* is produced by preprocessing a `module` directive, and the *export-keyword* is produced by preprocessing either of the previous two directives. [*Note:* None has any observable spelling. *— end note*]

[*Example:* The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid *integer-literal* or *floating-point-literal* token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` might produce a valid expression (for example, if `foo` were a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid *floating-point-literal* token), whether or not `E` is a macro name. *— end example*]

[*Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. *— end example*]

 Alternative tokens [lex.digraph]

Alternative token representations are provided for some operators and punctuators.[6]

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[7] The set of alternative tokens is defined in .

[6] These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative *preprocessing-token* s is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

[7] Thus the "stringized" values of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

Table 1: Alternative tokens

Alternative	Primary	Alternative	Primary	Alternative	Primary
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

Tokens [lex.token]

token:
identifier
keyword
literal
operator-or-punctuator

There are five kinds of tokens: identifiers, keywords, literals,[8] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. [*Note:* Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note*]

Comments [lex.comment]

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [*Note:* The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment. — *end note*]

Header names [lex.header]

header-name:
< *h-char-sequence* >
" *q-char-sequence* "

h-char-sequence:
h-char
h-char-sequence h-char

[8] Literals include strings and character and numeric literals.

h-char:
any member of the source character set except new-line and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except new-line and "

[*Note:* Header name preprocessing tokens only appear within a `#include` preprocessing directive, a `__has_include` preprocessing expression, or after certain occurrences of an `import` token (see **??**). — *end note*] The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external source file names as specified in **??**.

The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally-supported with implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*.[9]

🔲 Preprocessing numbers [lex.ppnumber]

pp-number:
digit
. digit
pp-number identifier-continue
~~*pp-number ' digit*~~
~~*pp-number ' nondigit*~~
pp-number ' identifier-continue
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `p` *sign*
pp-number `P` *sign*
pp-number .

Preprocessing number tokens lexically include all *integer-literal* tokens and all *floating-point-literal* tokens.

A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an *integer-literal* token or a *floating-point-literal* token.

🔲 Identifiers [lex.name]

identifier:
identifier-start
identifier identifier-continue

[9] Thus, a sequence of characters that resembles an escape sequence might result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.

identifier-start:
~~*nondigit*~~
_
universal-character-name of class XID_Start

identifier-continue:
~~*nondigit*~~
~~*digit*~~
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

The character classes XID_Start and XID_Continue are specified in UAX 44. [10]

The identifiers in have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Unless otherwise specified, any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Table 2: Identifiers with special meaning

`final`	`import`	`module`	`override`

In addition, some identifiers are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

- Each identifier that contains a double underscore `__` or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.
- Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

🔲 Keywords [lex.key]

keyword:
any identifier listed in
import-keyword
module-keyword
export-keyword

[10] On systems in which linkers cannot accept ~~extended characters, an encoding of the *universal-character-name* may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the \u in a *universal-character-name*. Extended characters~~ some codepoints, an implementation defined scheme may be used in forming valid external identifiers. This may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.

The identifiers shown in are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token*. [*Note:* The `register` keyword is unused but is reserved for future use. — *end note*]

Table 3: Keywords

alignas	constinit	false	public	true
alignof	const_cast	float	register	try
asm	continue	for	reinterpret_cast	typedef
auto	co_await	friend	requires	typeid
bool	co_return	goto	return	typename
break	co_yield	if	short	union
case	decltype	inline	signed	unsigned
catch	default	int	sizeof	using
char	delete	long	static	virtual
char8_t	do	mutable	static_assert	void
char16_t	double	namespace	static_cast	volatile
char32_t	dynamic_cast	new	struct	wchar_t
class	else	noexcept	switch	while
concept	enum	nullptr	template	
const	explicit	operator	this	
consteval	export	private	thread_local	
constexpr	extern	protected	throw	

Furthermore, the alternative representations shown in for certain operators and punctuators are reserved and shall not be used otherwise.

Table 4: Alternative representations

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

🔲 Operators and punctuators [lex.operators]

The lexical representation of C++ programs includes a number of preprocessing tokens that are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc:
preprocessing-operator
operator-or-punctuator

preprocessing-operator: one of
`#` `##` `%:` `%:%:`

operator-or-punctuator: one of

```
{        }        [        ]        (        )
<:       :>       <%       %>       ;        :        ...
?        ::       .        .*       ->       ->*      ~
!        +        -        *        /        %        ^        &        |
=        +=       -=       *=       /=       %=       ^=       &=
|=
==       !=       <        >        <=       >=       <=>      &&
||
<<       >>       <<=      >>=      ++       --       ,
and      or       xor      not      bitand   bitor    compl
and_eq   or_eq    xor_eq   not_eq
```

Each *operator-or-punctuator* is converted to a single token in translation phase 7.

Literals [lex.literal]

Kinds of literals [lex.literal.kinds]

There are several kinds of literals.[11]

literal:
integer-literal
character-literal
floating-point-literal
string-literal
boolean-literal
pointer-literal
user-defined-literal

Integer literals [lex.icon]

integer-literal:
binary-literal $_{opt}$*integer-suffix*
octal-literal $_{opt}$*integer-suffix*
decimal-literal $_{opt}$*integer-suffix*
hexadecimal-literal $_{opt}$*integer-suffix*

binary-literal:
`0b` *binary-digit*
`0B` *binary-digit*
binary-literal $_{opt}$ `'` *binary-digit*

octal-literal:
`0`
octal-literal $_{opt}$ `'` *octal-digit*

decimal-literal:
nonzero-digit
decimal-literal $_{opt}$ `'` *digit*

[11] The term “literal” generally designates, in this document, those tokens that are called “constants” in ISO C.

hexadecimal-literal:
hexadecimal-prefix hexadecimal-digit-sequence

binary-digit: one of
`0 1`

octal-digit: one of
`0 1 2 3 4 5 6 7`

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

hexadecimal-prefix: one of
`0x 0X`

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence $_{opt}$ *'* *hexadecimal-digit*

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix $_{opt}$*long-suffix*
unsigned-suffix $_{opt}$*long-long-suffix*
long-suffix $_{opt}$*unsigned-suffix*
long-long-suffix $_{opt}$*unsigned-suffix*

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

long-long-suffix: one of
`ll LL`

In an *integer-literal*, the sequence of *binary-digit* s, *octal-digit* s, *digit* s, or *hexadecimal-digit* s is interpreted as a base N integer as shown in table ; the lexically first digit of the sequence of digits is the most significant. [*Note:* The prefix and any optional separating single quotes are ignored when determining the value. — *end note*]

Table 5: Base of *integer-literal*s

Kind of *integer-literal*	**base** N
binary-literal	2
octal-literal	8
decimal-literal	10
hexadecimal-literal	16

The *hexadecimal-digit* s `a` through `f` and `A` through `F` have decimal values ten through fifteen. [*Example:* The number twelve can be written `12`, `014`, `0XC`, or `0b1100`. The *integer-literal* s `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value. — *end example*]

The type of an *integer-literal* is the first type in the list in corresponding to its optional *integer-suffix* in which its value can be represented. An *integer-literal* is a prvalue.

Table 6: Types of *integer-literal* s

integer-suffix	***decimal-literal***	***integer-literal* other than *decimal-literal***

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

If an *integer-literal* cannot be represented by any type in its list and an extended integer type can represent its value, it may have that extended integer type. If all of the types in the list for the *integer-literal* are signed, the extended integer type shall be signed. If all of the types in the list for the *integer-literal* are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an *integer-literal* that cannot be represented by any of the allowed types.

🔒 Character literals [lex.ccon]

character-literal:
    $_{opt}$*encoding-prefix* ' *c-char-sequence* '

encoding-prefix: one of
    `u8` `u` `U` `L`

c-char-sequence:
    c-char
    c-char-sequence c-char

c-char:
    any ~~member of the basic source character set~~ codepoint except the single-quote ', backslash \, or new-line character
    escape-sequence
    ~~*universal-character-name*~~

escape-sequence:
simple-escape-sequence
numeric-escape-sequence
~~*octal-escape-sequence*~~
~~*hexadecimal-escape-sequence*~~

simple-escape-sequence: one of
`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`

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

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

Table X specifies the kinds of *character-literals* and their properties. The type of a character-literal is the associated code unit type. Each *character-literal* kind has an associated encoding and an associated characters set. [*Note:* A single *character-literal* may not be able to encode all members of its associated characters set. — *end note*] *multicharacter literals* are distinguished from *ordinary character literals* by the presence of a *c-char-sequence* that contains more than one *c-char*; *multicharacter literals* are conditionally supported.

Table 7: Types of *character-literal* s

Kind	*encoding-prefix*	code unit type	character encoding	example
ordinary character literal	none	`char`	execution encoding	'v'
multicharacter literal	none	`char`	implementation-defined	'123'
wide character literal	none	`wchar_t`	execution wide-encoding	L'w'
UTF-8 character literal	u8	`char8_t`	UTF-8	u8'v'
UTF-16 character literal	u	`char16_t`	UTF-16	u'v'
UTF-32 character literal	U	`char32_t`	UTF-32	U'v'

The value of a character literal is as follows.

- The value of a *multicharacter literal* is implementation defined.
- The value of a *character literal* consisting of a single *basic-c-char* or *character-escape-sequence* is the code unit value of the specified codepoint encoded with the character literal's associated character encoding.
- The value of a character literal consisting of multiple basic-c-char is the code unit value of a single code point representing the same abstract character as the literal's *c-char-sequence*.

- The value of a character literal consisting of a single numeric-escape-sequence is the numeric value of the octal or hexadecimal number. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. If the numeric value exceeds the range of the character literal's associated code unit type, then, the character literal is ill-formed.

A character literal whose associated character set is the Unicode character set and consisting of multiple basic-c-char or character-escape-sequence is ill-formed.

If a *character literal* that is not a *multicharacter literal* cannot be represented in a single code unit of that *character literal*'s associated encoding, the program is ill-formed.

A *character-literal* that does not begin with `u8`, `u`, `U`, or `L` is an *ordinary character literal*. An ordinary character literal that contains a single *c-char* representable in the execution character set has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal, or an ordinary character literal containing a single *c-char* not representable in the execution character set, is conditionally-supported, has type `int`, and has an implementation-defined value.

A *character-literal* that begins with `u8`, such as `u8'w'`, is a *character-literal* of type `char8_t`, known as a *UTF-8 character literal*. The value of a UTF-8 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-8 code unit. [*Note:* That is, provided the code point value is in the range $[0, 7\mathrm{F}]$ (hexadecimal). — *end note*] If the value is not representable with a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing multiple *c-char*s is ill-formed.

A *character-literal* that begins with the letter `u`, such as `u'x'`, is a *character-literal* of type `char16_t`, known as a *UTF-16 character literal*. The value of a UTF-16 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value is representable with a single 16-bit code unit. [*Note:* That is, provided the code point value is in the range $[0, \mathrm{FFFF}]$ (hexadecimal). — *end note*] If the value is not representable with a single 16-bit code unit, the program is ill-formed. A UTF-16 character literal containing multiple *c-char*s is ill-formed.

A *character-literal* that begins with the letter `U`, such as `U'y'`, is a *character-literal* of type `char32_t`, known as a *UTF-32 character literal*. The value of a UTF-32 character literal containing a single *c-char* is equal to its ISO/IEC 10646 code point value. A UTF-32 character literal containing multiple *c-char*s is ill-formed.

A *character-literal* that begins with the letter `L`, such as `L'z'`, is a *wide-character literal*. A wide-character literal has type `wchar_t`.[12] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined. [*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set (see **??**). — *end note*] The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

[12] They are intended for character sets where a character does not fit into a single byte.

Certain non-graphic characters, the single quote ', the double quote ", the question mark ?, and the backslash \, can be represented according to [lex.ccon.esc]. The double quote " and the question mark ?, can be represented as themselves or by the escape sequences \" and \? respectively, but the single quote ' and the backslash \ shall be represented by the escape sequences \' and \\ respectively. Escape sequences in which the character following the backslash is not listed in are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

The character specified by a simple-escape-sequence is specified in table 8[13].

Table 8: ~~E~~ Simple escape sequences

~~new-line~~ NEW LINE	~~NL(LF)~~ U+000A	\n
~~horizontal tab~~ CHARACTER TABULATION	~~HT~~ U+0009	\t
~~vertical tab~~ LINE TABULATION	~~VT~~ U+000B	\v
~~backspace~~ BACKSPACE	~~BS~~ U+0008	\b
~~carriage return~~ CARRIAGE RETURN	~~CR~~ U+000D	\r
~~form feed~~ FORM FEED	~~FF~~ U+000C	\f
~~alert~~ BELL	~~BEL~~ U+0007	\a
~~backslash~~ REVERSE SOLIDUS	\	\\
~~question mark~~ QUESTION MARK	?	\?
~~single quote~~ APOSTROPHE	'	\'
~~double quote~~ QUOTATION MARK	"	\"
~~octal number~~	~~*ooo*~~	~~*ooo*~~
~~hex number~~	~~*hhh*~~	~~\x*hhh*~~

The escape *ooo* consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \x*hhh* consists of the backslash followed by x followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a *character-literal* is implementation-defined if it falls outside of the implementation-defined range defined for char (for *character-literal* s with no prefix) or wchar_t (for *character-literal* s prefixed by L). [*Note:* If the value of a *character-literal* prefixed by u, u8, or U is outside the range defined for its type, the program is ill-formed. — *end note*]

A *universal-character-name* is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the *universal-character-name* is translated to an implementation-defined encoding. [*Note:* In translation phase 1, a *universal-character-name* is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of *universal-character-name*s. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — *end note*]

[13] Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C.

🔲 Floating-point literals [lex.fcon]

floating-point-literal:
decimal-floating-point-literal
hexadecimal-floating-point-literal

decimal-floating-point-literal:
fractional-constant $_{opt}$*exponent-part* $_{opt}$*floating-point-suffix*
digit-sequence exponent-part $_{opt}$*floating-point-suffix*

hexadecimal-floating-point-literal:
hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part $_{opt}$*floating-point-suffix*
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part $_{opt}$*floating-point-suffix*

fractional-constant:
$_{opt}$*digit-sequence* `.` *digit-sequence*
digit-sequence `.`

hexadecimal-fractional-constant:
$_{opt}$*hexadecimal-digit-sequence* `.` *hexadecimal-digit-sequence*
hexadecimal-digit-sequence `.`

exponent-part:
`e` $_{opt}$*sign digit-sequence*
`E` $_{opt}$*sign digit-sequence*

binary-exponent-part:
`p` $_{opt}$*sign digit-sequence*
`P` $_{opt}$*sign digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence $_{opt}$ `'` *digit*

floating-point-suffix: one of
`f l F L`

The type of a *floating-point-literal* is determined by its *floating-point-suffix* as specified in .

Table 9: Types of *floating-point-literal*s

floating-point-suffix	**type**
none	`double`
`f` or `F`	`float`
`l` or `L`	`long double`

The *significand* of a *floating-point-literal* is the *fractional-constant* or *digit-sequence* of a *decimal-floating-point-literal* or the *hexadecimal-fractional-constant* or *hexadecimal-digit-sequence* of a *hexadecimal-floating-point-literal*. In the significand, the sequence of *digit* s or *hexadecimal-digit* s and optional period are interpreted as a base N real number s, where N is 10 for a *decimal-floating-point-literal* and 16 for a *hexadecimal-floating-point-literal*. [*Note:* Any optional

separating single quotes are ignored when determining the value. — *end note*] If an *exponent-part* or *binary-exponent-part* is present, the exponent e of the *floating-point-literal* is the result of interpreting the sequence of an optional *sign* and the *digit* s as a base 10 integer. Otherwise, the exponent e is 0. The scaled value of the literal is $s \times 10^e$ for a *decimal-floating-point-literal* and $s \times 2^e$ for a *hexadecimal-floating-point-literal*. [*Example:* The *floating-point-literal*s `49.625` and `0xC.68p+2` have the same value. The *floating-point-literal*s `1.602'176'565e-19` and `1.602176565e-19` have the same value. — *end example*]

If the scaled value is not in the range of representable values for its type, the program is ill-formed. Otherwise, the value of a *floating-point-literal* is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.

🔒 String literals [lex.string]

string-literal:
    $_{opt}$*encoding-prefix* " $_{opt}$*s-char-sequence* "
    $_{opt}$*encoding-prefix* `R` *raw-string*

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char:
    any ~~member of the basic source character set~~ codepoint except the double-quote ", backslash \, or new-line character
    escape-sequence
    universal-character-name

raw-string:
    " $_{opt}$*d-char-sequence* ($_{opt}$*r-char-sequence*) $_{opt}$*d-char-sequence* "

r-char-sequence:
    r-char
    r-char-sequence r-char

r-char:
    any ~~member of the basic source character set~~ codepoint, except a right parenthesis) followed by
        the initial *d-char-sequence* (which may be empty) followed by a double quote ".

d-char-sequence:
    d-char
    d-char-sequence d-char

d-char:
    any ~~member of the basic source character set~~ codepoint except:

space, the left parenthesis (, the right parenthesis), the backslash \, and the control characters
        representing horizontal tab, vertical tab, form feed, and newline.

A *string-literal* that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a

delimiter. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

[*Note:* The characters `'('` and `')'` are permitted in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`. — *end note*]

[*Note:* A source-file new-line in a raw string literal results in a new-line in the resulting execution string literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char* p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

— *end note*]

[*Example:* The raw string

```
R"a(
)\
a"
)a"
```

is equivalent to `"\n)\\\na\"\n"`. The raw string

```
R"(x = "\"y\"")"
```

is equivalent to `"x = \"\\\"y\\\"\""`. — *end example*]

Table X specifies the kinds of *string-literal* s and their properties.

Table 10: Types of *character-literal* s

Kind	*encoding-prefix*	code type	character encoding	examples
ordinary string literal	none	`char`	execution encoding	'v'
wide string literal	none	`wchar_t`	execution wide-encoding	L'w'
UTF-8 string literal	u8	`char8_t`	UTF-8	u8'v'
UTF-16 string literal	u	`char16_t`	UTF-16	u'v'
UTF-32 string literal	U	`char32_t`	UTF-32	U'v'

After translation phase 6, a *string-literal* that does not begin with an *encoding-prefix* is an *ordinary string literal*. An ordinary string literal has type "array of *n* `const char`" where *n* is the size of the string as defined below, has static storage duration, and is initialized with the given characters.

A *string-literal* that begins with `u8`, such as `u8"asdf"`, is a *UTF-8 string literal*. A UTF-8 string literal has type "array of *n* `const char8_t`", where *n* is the size of the string as defined below;

each successive element of the object representation has the value of the corresponding code unit of the UTF-8 encoding of the string.

Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals.

String literal objects are initialized with the sequence of code unit values corresponding to the string-literal's as follows:

- If the string literal associated character set is the Unicode character set, each basic-s-char, r-char, character-escape-sequence are encoded to a code unit sequence using the string-literal's associated character encoding.
- Otherwise, codepoint in the *s-char-sequence* is mapped to a sequence of code units representing the same abstract character in the string literal's associated encoding. If an abstract character lacks representation in the associated character set the program is ill formed.
- Each numeric-escape-sequence ([lex.ccon]) contributes a single code unit value with the numeric value of the octal or hexadecimal number. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. If the numeric value exceeds the range of the string-literal's code unit type, then the string-literal is ill-formed.

A *string-literal* that begins with `u`, such as `u"asdf"`, is a *UTF-16 string literal*. A UTF-16 string literal has type "array of *n* `const char16_t`", where *n* is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-16 encoding of the string. [*Note:* A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs. A surrogate pair is a representation for a single code point as a sequence of two 16-bit code units. — *end note*]

A *string-literal* that begins with `U`, such as `U"asdf"`, is a *UTF-32 string literal*. A UTF-32 string literal has type "array of *n* `const char32_t`", where *n* is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-32 encoding of the string.

A *string-literal* that begins with `L`, such as `L"asdf"`, is a *wide string literal*. A wide string literal has type "array of *n* `const wchar_t`", where *n* is the size of the string as defined below; it is initialized with the given characters.

In translation phase 6, adjacent *string-literal*s are concatenated. If both *string-literal*s have the same *encoding-prefix*, the resulting concatenated *string-literal* has that *encoding-prefix*. If one *string-literal* has no *encoding-prefix*, it is treated as a *string-literal* of the same *encoding-prefix* as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (~~after each character from a *string-literal* has been translated into a value from the appropriate character set~~ after the string literal contents have been encoded in the string-literal's associated character encoding),

a *string-literal*'s initial rawness has no effect on the interpretation or well-formedness of the concatenation. — *end note*] has some examples of valid concatenations.

Table 11: String literal concatenations

Source		Means	Source		Means	Source		Means
u"a"	u"b"	u"ab"	U"a"	U"b"	U"ab"	L"a"	L"b"	L"ab"
u"a"	"b"	u"ab"	U"a"	"b"	U"ab"	L"a"	"b"	L"ab"
"a"	u"b"	u"ab"	"a"	U"b"	U"ab"	"a"	L"b"	L"ab"

Characters in concatenated strings are kept distinct.

[*Example:*

```
"\xA" "B"
```

contains the two characters `'\xA'` and `'B'` after concatenation (and not the single hexadecimal character `'\xAB'`). — *end example*]

After any necessary concatenation, in translation phase 7, ~~`'\0'`~~ a null code unit is appended to every *string-literal* so that programs that scan a string can find its end.

Escape sequences and *universal-character-name*s in non-raw string literals have the same meaning as in *character-literal* s, except that the single quote `'` is representable either by itself or by the escape sequence `\'`, and the double quote `"` shall be preceded by a `\`, and except that a *universal-character-name* in a UTF-16 string literal may yield a surrogate pair. In a narrow string literal, a *universal-character-name* may map to more than one `char` or `char8_t` element due to *multibyte encoding*. The size of a `char32_t` or wide string literal is the total number of escape sequences, *universal-character-name*s, and other characters, plus one for the terminating `U'\0'` or `L'\0'`. The size of a UTF-16 string literal is the total number of escape sequences, *universal-character-name*s, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating `u'\0'`. [*Note:* The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*] [*Note:* Any *universal-character-name*s are required to correspond to a code point in the range $[0, \mathrm{D800})$ or $[\mathrm{E000}, \mathrm{10FFFF}]$ (hexadecimal). — *end note*] The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each *universal-character-name*, plus one for the terminating `'\0'`.

The size of a string-literal is the number of code unit in the string literal including the null character appened in phase 7.

Evaluating a *string-literal* results in a string literal object with static storage duration~~, initialized from the given characters as specified above~~. Whether all *string-literal* s are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified. [*Note:* The effect of attempting to modify a *string-literal* is undefined. — *end note*]

Boolean literals [lex.bool]

boolean-literal:
`false`
`true`

The Boolean literals are the keywords `false` and `true`. Such literals are prvalues and have type `bool`.

Pointer literals [lex.nullptr]

pointer-literal:
`nullptr`

The pointer literal is the keyword `nullptr`. It is a prvalue of type `std::nullptr_t`. [*Note:* `std::nullptr_t` is a distinct type that is neither a pointer type nor a pointer-to-member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See **??** and **??**. — *end note*]

User-defined literals [lex.ext]

user-defined-literal:
user-defined-integer-literal
user-defined-floating-point-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix
binary-literal ud-suffix

user-defined-floating-point-literal:
fractional-constant $_{opt}$*exponent-part ud-suffix*
digit-sequence exponent-part ud-suffix
hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
identifier

If a token matches both *user-defined-literal* and another *literal* kind, it is treated as the latter. [*Example:* `123_km` is a *user-defined-literal*, but `12LL` is an *integer-literal*. — *end example*] The syntactic non-terminal preceding the *ud-suffix* in a *user-defined-literal* is taken to be the longest sequence of characters that could match that non-terminal.

A *user-defined-literal* is treated as a call to a literal operator or literal operator template. To determine the form of this call for a given *user-defined-literal* *L* with *ud-suffix* *X*, the *literal-operator-id* whose literal suffix identifier is *X* is looked up in the context of *L* using the rules for unqualified name lookup. Let *S* be the set of declarations found by this lookup. *S* shall not be empty.

If *L* is a *user-defined-integer-literal*, let *n* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `unsigned long long`, the literal *L* is treated as a call of the form

```
operator "" X(nULL)
```

Otherwise, *S* shall contain a raw literal operator or a numeric literal operator template but not both. If *S* contains a raw literal operator, the literal *L* is treated as a call of the form

```
operator "" X("n")
```

Otherwise (*S* contains a numeric literal operator template), *L* is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where *n* is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

If *L* is a *user-defined-floating-point-literal*, let *f* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `long double`, the literal *L* is treated as a call of the form

```
operator "" X(fL)
```

Otherwise, *S* shall contain a raw literal operator or a numeric literal operator template but not both. If *S* contains a raw literal operator, the *literal* *L* is treated as a call of the form

```
operator "" X("f")
```

Otherwise (*S* contains a numeric literal operator template), *L* is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where *f* is the source character sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

If *L* is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). If *S* contains a literal operator template with a non-type template parameter for which *str* is a well-formed *template-argument*, the literal *L* is treated as a call of the form

```
operator "" X<str>()
```

Otherwise, the literal *L* is treated as a call of the form

```
operator "" X(str, len)
```

If *L* is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. *S* shall contain a literal operator whose only parameter has the type of *ch* and the literal *L* is treated as a call of the form

```
operator "" X(ch)
```

[*Example:*

```
long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, std::size_t);
unsigned operator "" _w(const char*);
int main() {
    1.2_w;            // calls operator "" _w(1.2L)
    u"one"_w;         // calls operator "" _w(u"one", 3)
    12_w;             // calls operator "" _w("12")
    "two"_w;          // error: no applicable literal operator
}
```

— *end example*]

In translation phase 6, adjacent *string-literal* s are concatenated and *user-defined-string-literal*s are considered *string-literal* s for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in **??**. At the end of phase 6, if a *string-literal* is the result of a concatenation involving at least one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

[*Example:*

```
int main() {
    L"A" "B" "C"_x;   // OK: same as L"ABC"_x
    "P"_x "Q" "R"_y;  // error: two different ud-suffixes
}
```

— *end example*]

References

[1] Beman Dawes. N3463: Portable program source files. https://wg21.link/n3463, 11 2012.

[2] James Kanze. CWG411: Use of universal-character-name in character versus string literals. https://wg21.link/cwg411, 4 2003.

[3] David Krauss. CWG1403: Universal-character-names in comments. https://wg21.link/cwg1403, 10 2011.

[4] Mike Miller. CWG1332: Handling of invalid universal-character-names. https://wg21.link/cwg1332, 6 2011.

[5] Mike Miller. CWG1655: Line endings in raw string literals. https://wg21.link/cwg1655, 4 2013.

[6] Johannes Schaub. CWG1335: Stringizing, extended characters, and universal-character-names. https://wg21.link/cwg1335, 7 2011.

[7] Martin Vejnár. CWG578: Phase 1 replacement of characters with universal-character-names. https://wg21.link/cwg578, 5 2006.

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++* https://wg21.link/N4861