Down with ”character”

Document #: P2749R0
Date: 2023-01-26
Programming Language C++
Audience: SG-16, CWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to replace incorrect and ambiguous use of the term ”character” by the technically correct term during translation.

Motivation

The term ”character” is often incorrect, imprecise, and often ambiguous. the ”translation set” and its elements are a C++ invention.

By using terms of art, we hope to make the wording clearer, easier to interpret, and harder to missinterpret.

This resolves NB comment FR-020-014 5.3.

Changes

In phase 1 of lexing, and when describing constraints on *universal-character-name*, we use the term ”Unicode scalar value” which describe the constraints placed on each code point. Everywhere else, we prefer the term ”Unicode code point” as a few people have expressed finding ”scalar value” to be too obscure of a terminology, and both terms being interchangeable once we established the constraints. All Unicode code points during translation are scalar values.

Uses of the term character are retained when describing specific C++ elements such as ”character literal” or ”character type. Further more, when ”character” refers to a clearly identified abstract character, such as in the expressions ”new-line character” or ”quotation character”, we keep the term to avoid unecessary changes.

Neither the library sections, nor the definitions pertaining to library clauses in [intro] are modified.

~~Changes in green~~ and red are the addition and removal respectively associated to the replacement of the term ”character”.

~~Changes in blue~~ and pink are the addition and removal respectively associated to the replacement of the term ”translation character set”.

Changes in purple are some of the changes made in ”Referencing the Unicode Standard”

Wording

� Separate translation [lex.separate]

The text of the program is kept in units called *source files* in this document. A source file together with all the headers[headers] and source files included[cpp.include] via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion[cpp.cond] preprocessing directives, is called a *translation unit*. [*Note:* A C++ program need not all be translated at the same time. — *end note*]

[*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate[basic.link] by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program[basic.link]. — *end note*]

� Phases of translation [lex.phases]

The precedence among the syntax rules of translation is specified by the following phases. [*Footnote:* Implementations behave as if these separate phases occur, although in practice different phases can be folded together. — *end note*]

1. An implementation shall support input files that are a sequence of UTF-8 code units (UTF-8 files). It may also support an implementation-defined set of other kinds of input files, and, if so, the kind of an input file is determined in an implementation-defined manner that includes a means of designating input files as UTF-8 files, independent of their content. [*Note:* In other words, recognizing the u+feff byte order mark is not sufficient. — *end note*] If an input file is determined to be a UTF-8 file, then it shall be a well-formed UTF-8 code unit sequence and it is decoded to produce a sequence of Unicode scalar values ~~that constitutes the sequence of elements of the translation character set~~. In the resulting sequence, each pair of ~~characters~~ Unicode scalar values in the input sequence consisting of u+000d carriage return followed by u+000a line feed, as well as each u+000d carriage return not immediately followed by a u+000a line feed, is replaced by a single new-line character.

   For any other kind of input file supported by the implementation, abstract characters are mapped, in an implementation-defined manner, to a sequence of ~~translation character set elements~~ Unicode scalar values~~[lex.charset]~~, representing end-of-line indicators as new-line characters.

2. If the first ~~translation character~~ Unicode code point is u+feff byte order mark, it is deleted. Each sequence of ~~a backslash character (\)~~ u+005C backslash immediately followed by

zero or more whitespace characters other than new-line followed by a new-line character 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, if a splice results in a ~~character~~ Unicode code point 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 splice, shall be processed as if an additional new-line character were appended to the file.

3. The source file is decomposed into preprocessing tokens[lex.pptoken] and sequences of whitespace characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. [*Footnote:* A partial preprocessing token would arise from a source file ending in the first portion of a multi-~~character~~ Unicode code point token that requires a terminating sequence of ~~character~~ Unicode code points, 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. — *end note*] Each comment is replaced by one ~~space character~~ U+0020 SPACE character. New-line characters are retained. Whether each nonempty sequence of whitespace characters other than new-line is retained or replaced by one ~~space character~~ U+0020 SPACE character is unspecified. As ~~characters~~ Unicode code points from the source file are consumed to form the next preprocessing token (i.e., not being consumed as part of a comment or other forms of whitespace), except when matching a *c-char-sequence*, *s-char-sequence*, *r-char-sequence*, *h-char-sequence*, or *q-char-sequence*, *universal-character-name* s are recognized and replaced by the designated ~~element of the translation character set~~ Unicode code point. The process of dividing a source file's ~~characters~~ code points 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. 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. For a sequence of two or more adjacent *string-literal* tokens, a common *encoding-prefix* is determined as specified in **??**. Each such *string-literal* token is then considered to have that common *encoding-prefix*.

6. Adjacent *string-literal* tokens are concatenated[lex.string].

7. Whitespace characters separating tokens are no longer significant. Each preprocessing token is converted into a token[lex.token]. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens can occasionally result in one token being replaced by a sequence of other tokens[temp.names]. — *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[module.unit,module.import] 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 can be supplied from a library. *— end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* This can include instantiations which have been explicitly requested[temp.explicit]. *— 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 can choose to 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 *translation character set* consists of the following elements:

- each character named by ISO/IEC 10646, as identified by its unique UCS scalar value, and
- a distinct character for each UCS scalar value where no named character is assigned.

[*Note:* ISO/IEC 10646 code points are integers in the range $[0, 10\mathrm{FFFF}]$ (hexadecimal). A surrogate code point is a value in the range $[\mathrm{D800}, \mathrm{DFFF}]$ (hexadecimal). A UCS scalar value is any code point that is not a surrogate code point. — *end note*]

The *basic character set* is a subset of the ~~translation~~ Unicode character set, consisting of 96 characters as specified in [lex.charset.basic]. [*Note:* Unicode short names are given only as a means to identifying the ~~character~~ code point; the numerical value has no other meaning in this context. — *end note*]

The *universal-character-name* construct provides a way to name other ~~characters~~ Unicode code points.

n-char: one of
  ~~any member of the translation character set~~ Unicode code point except the u+007d right curly bracket or new-line character

n-char-sequence:
  n-char
  n-char-sequence n-char

named-universal-character:
  `\N{` *n-char-sequence* `}`

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

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

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

A *universal-character-name* of the form `\u` *hex-quad*, `\U` *hex-quad hex-quad*, or `\u{`*simple-hexadecimal-digit-sequence* `}` designates the ~~character in the translation character set whose~~ Unicode scalar value ~~is~~ equal to 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 Unicode scalar value.

A *universal-character-name* that is a *named-universal-character* designates the ~~character~~ Unicode code point named by its *n-char-sequence*. A ~~character~~ Unicode code point is so named if the *n-char-sequence* is equal to

Table 1: Basic character set

character		glyph
u+0009	character tabulation	
u+000b	line tabulation	
u+000c	form feed	
u+0020	space	
u+000a	line feed	new-line
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	/
u+0030 .. u+0039	digit zero .. nine	0 1 2 3 4 5 6 7 8 9
u+003a	colon	:
u+003b	semicolon	;
u+003c	less-than sign	<
u+003d	equals sign	=
u+003e	greater-than sign	>
u+003f	question mark	?
u+0041 .. u+005a	latin capital letter a .. 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
u+005b	left square bracket	[
u+005c	reverse solidus	\
u+005d	right square bracket	]
u+005e	circumflex accent	^
u+005f	low line	_
u+0061 .. u+007a	latin small letter a .. 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
u+007b	left curly bracket	{
u+007c	vertical line	|
u+007d	right curly bracket	}
u+007e	tilde	~

- the associated character name or associated character name alias specified in ISO/IEC 10646 subclause “Code charts and lists of character names” or
- the control code alias given in [lex.charset.ucn]. [*Note:* The aliases in [lex.charset.ucn] are provided for control characters which otherwise have no associated character name or character name alias. These names are derived from the Unicode Character Database’s `NameAliases.txt`. For historical reasons, control characters are formally unnamed. — *end note*]

[*Note:* None of the associated character names, associated character name aliases, or control code aliases have leading or trailing spaces. — *end note*]

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 character set, the program is ill-formed. [*Note:* A sequence of ~~characters~~ code points resembling a *universal-character-name* in an *r-char-sequence* [lex.string] does not form a *universal-character-name*. — *end note*]

The *basic literal character set* consists of all abstract characters of the basic character set, plus the control characters specified in [lex.charset.literal]. [*Note:* The alias bell for U+0007 shown in ISO 10646 is ambiguous with U+1F514 bell. — *end note*]

A *code unit* is an integer value of character type[basic.fundamental]. ~~Characters~~ Unicode code points in a *character-literal* other than a multicharacter or non-encodable character literal or in a *string-literal* are encoded as a sequence of one or more code units, as determined by the *encoding-prefix* [lex.ccon,lex.string]; this is termed the respective *literal encoding*. The *ordinary literal encoding* is the encoding applied to an ordinary character or string literal. The *wide literal encoding* is the encoding applied to a wide character or string literal.

A literal encoding or a locale-specific encoding of one of the execution character sets[character.seq] encodes each element of the basic literal character set as a single code unit with non-negative value, distinct from the code unit for any other such element. [*Note:* A ~~character~~ Unicode code point not in the basic literal character set can be encoded with more than one code unit; the value of such a code unit can be the same as that of a code unit for an element of the basic literal character set. — *end note*] The U+0000 null character is encoded as the value `0`. No other ~~element of the translation character set~~ Unicode code point is encoded with a code unit of value `0`. The code unit value of each decimal digit character after the digit `0` (U+0030) shall be one greater than the value of the previous. The ordinary and wide literal encodings are otherwise implementation-defined. For a UTF-8, UTF-16, or UTF-32 literal, ~~the UCS scalar value corresponding to each character of the translation character set~~ each Unicode scalar value is encoded as specified in ISO/IEC 10646 for the respective UCS encoding form.

Table 2: Control code aliases

u+0000 null	u+007f delete
u+0001 start of heading	u+0082 break permitted here
u+0002 start of text	u+0083 no break here
u+0003 end of text	u+0084 index
u+0004 end of transmission	u+0085 next line
u+0005 enquiry	u+0086 start of selected area
u+0006 acknowledge	u+0087 end of selected area
u+0007 alert	u+0088 character tabulation set
u+0008 backspace	u+0088 horizontal tabulation set
u+0009 character tabulation	u+0089 character tabulation with justification
u+0009 horizontal tabulation	u+0089 horizontal tabulation with justification
u+000a line feed	u+008a line tabulation set
u+000a new line	u+008a vertical tabulation set
u+000a end of line	u+008b partial line forward
u+000b line tabulation	u+008b partial line down
u+000b vertical tabulation	u+008c partial line backward
u+000c form feed	u+008c partial line up
u+000d carriage return	u+008d reverse line feed
u+000e shift out	u+008d reverse index
u+000e locking-shift one	u+008e single shift two
u+000f shift in	u+008e single-shift-2
u+000f locking-shift zero	u+008f single shift three
u+0010 data link escape	u+008f single-shift-3
u+0011 device control one	u+0090 device control string
u+0012 device control two	u+0091 private use one
u+0013 device control three	u+0091 private use-1
u+0014 device control four	u+0092 private use two
u+0015 negative acknowledge	u+0092 private use-2
u+0016 synchronous idle	u+0093 set transmit state
u+0017 end of transmission block	u+0094 cancel character
u+0018 cancel	u+0095 message waiting
u+0019 end of medium	u+0096 start of guarded area
u+001a substitute	u+0096 start of protected area
u+001b escape	u+0097 end of guarded area
u+001c information separator four	u+0097 end of protected area
u+001c file separator	u+0098 start of string
u+001d information separator three	u+009a single character introducer
u+001d group separator	u+009b control sequence introducer
u+001e information separator two	u+009c string terminator
u+001e record separator	u+009d operating system command
u+001f information separator one	u+009e privacy message
u+001f unit separator	u+009f application program command

Table 3: Additional control characters in the basic literal character set

~~character~~ Unicode code point	
u+0000	null
u+0007	alert
u+0008	backspace
u+000d	carriage return

�   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 non-whitespace ~~character~~ Unicode code point that cannot be one of the above

Each preprocessing token that is converted to a token[lex.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. In this document, glyphs are used to identify elements of the basic character set[lex.charset]. 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-whitespace ~~characters~~ Unicode code points that do not lexically match the other preprocessing token categories. If a u+0027 apostrophe or a u+0022 quotation mark character matches the last category, the behavior is undefined. If any ~~characters~~ Unicode code point not in the basic character set matches the last category, the program is ill-formed. Preprocessing tokens can be separated by whitespace; this consists of comments[lex.comment], or whitespace characters (u+0020 space, u+0009 character tabulation, new-line, u+000b line tabulation, and u+000c form feed), or both. As described in **??**, in certain circumstances during translation phase 4, whitespace (or the absence thereof) serves as more than preprocessing token separation. Whitespace 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~~ Unicode code point:

- If the next ~~character~~ Unicode code point begins a sequence of ~~characters~~ Unicode code points 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 phase 2 (line splicing) 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~~ Unicode code points that matches the raw-string pattern

  encoding-prefix$_{opt}$ `R` *raw-string*

- Otherwise, if the next three characters are `<::` and the subsequent ~~character~~ Unicode code point 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~~ Unicode code points that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* [lex.header] is only formed

  - after the `include` or `import` preprocessing token in an `#include`[cpp.include] or `import`[cpp.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[cpp.import], the *module-keyword* is produced by preprocessing a `module` directive[cpp.module], 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` can produce a valid expression (for example, if `foo` is 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` can yield a correct expression. — *end example*]

� Alternative tokens [lex.digraph]

Alternative token representations are provided for some operators and punctuators. [*Footnote:* These include “digraphs” and additional reserved words. The term “digraph” (token consisting of two ~~characters~~ Unicode code points) is not perfectly descriptive, since one of the alternative *preprocessing-token* s is `%:%:` and of course several primary tokens contain two ~~characters~~ Unicode code points. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs". *— end note*]

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling. [*Footnote:* Thus the "stringized" values[cpp.stringize] of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged. *— end note*] The set of alternative tokens is defined in [lex.digraph].

Table 4: 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,[*Footnote:* Literals include strings and character and numeric literals. *— end note*] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "whitespace"), as described below, are ignored except as they serve to separate tokens. [*Note:* Some whitespace 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 whitespace 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~~ Unicode code points. 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

h-char:
any ~~member of the translation character set~~ Unicode code point except new-line and U+003E greater-than sign

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

q-char:
any ~~member of the translation character set~~ Unicode code point except new-line and U+0022 quotation mark

[*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*. [*Footnote:* Thus, a sequence of ~~characters~~ Unicode code points that resembles an escape sequence can result in an error, be interpreted as the ~~character~~ Unicode code point corresponding to the escape sequence, or have a completely different meaning, depending on the implementation. — *end note*]

� Preprocessing numbers [lex.ppnumber]

pp-number:
digit
. *digit*
pp-number identifier-continue
pp-number ' *digit*
pp-number ' *nondigit*
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[lex.icon] and all *floating-point-literal* tokens[lex.fcon].

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

identifier-start:
nondigit
~~an element of the translation character set of class~~ a Unicode code point with the Unicode property XID_Start

identifier-continue:
digit
nondigit
~~an element of the translation character set of class~~ a Unicode code point with the Unicode property 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 Derived Core Properties as described by UAX44. [*Footnote:* On systems in which linkers cannot accept extended characters, an encoding of the *universal-character-name* can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the \u in a *universal-character-name*. ~~Extended characters can produce a long external identifier, but~~ C++ does not place a translation limit on significant characters for external identifiers. — *end note*]

The program is ill-formed if an *identifier* does not conform to Normalization Form C as specified in ISO/IEC 10646. [*Note:* Identifiers are case-sensitive. — *end note*] [*Note:* In translation phase 4, *identifier* also includes those *preprocessing-token* s[lex.pptoken] differentiated as keywords[lex.key] in the later translation phase 7[lex.token]. — *end note*]

The identifiers in [lex.name.special] 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 5: 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 in the basic character set is reserved to the implementation for any use.

  [Editor's note: "letter" here is very confusing, but existing implementations and the history of the wording strongly imply the intent]

- 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 [lex.key]
  import-keyword
  module-keyword
  export-keyword

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

Table 6: 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 [lex.key.digraph] for certain operators and punctuators[lex.digraph] are reserved and shall not be used otherwise.

Table 7: 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[lex.phases].

� Literals [lex.literal]

� Kinds of literals [lex.literal.kinds]

There are several kinds of literals.

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

[*Note:* When appearing as an *expression*, a literal has a type and a value category[expr.prim.literal]. — *end note*]

� Integer literals [lex.icon]

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

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*

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 long-suffix$_{opt}$
  unsigned-suffix long-long-suffix$_{opt}$
  unsigned-suffix size-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$
  long-long-suffix unsigned-suffix$_{opt}$
  size-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
  `u U`

long-suffix: one of
  `l L`

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

size-suffix: one of
`z Z`

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 [lex.icon.base]; 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 8: 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 [lex.icon.type] corresponding to its optional *integer-suffix* in which its value can be represented.

If an *integer-literal* cannot be represented by any type in its list and an extended integer type[basic.fundamental] 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:
encoding-prefix$_{opt}$ `'` *c-char-sequence* `'`

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

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

c-char:
basic-c-char
escape-sequence
universal-character-name

Table 9: 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`
`z` or `Z`	the signed integer type corresponding to `std::size_t`[support.types.layout]	the signed integer type corresponding to `std::size_t` `std::size_t`
Both `u` or `U` and `z` or `Z`	`std::size_t`	`std::size_t`

basic-c-char:
any ~~member of the translation character set~~ Unicode code point except the
U+0027 apostrophe,
U+005C reverse solidus, or new-line character

escape-sequence:
simple-escape-sequence
numeric-escape-sequence
conditional-escape-sequence

simple-escape-sequence:
\ *simple-escape-sequence-char*

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

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

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

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

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

conditional-escape-sequence:
\ *conditional-escape-sequence-char*

conditional-escape-sequence-char:
any member of the basic character set that is not an *octal-digit*, a *simple-escape-sequence-char*, or the characters N, o, u, U, or x

A *non-encodable character literal* is a *character-literal* whose *c-char-sequence* consists of a single *c-char* that is not a *numeric-escape-sequence* and that specifies a ~~character~~ Unicode code point that either lacks representation in the literal's associated character encoding or that cannot be encoded as a single code unit. A *multicharacter literal* is a *character-literal* whose *c-char-sequence* consists of more than one *c-char*. The *encoding-prefix* of a non-encodable character literal or a multicharacter literal shall be absent. Such *character-literal* s are conditionally-supported.

The kind of a *character-literal*, its type, and its associated character encoding[lex.charset] are determined by its *encoding-prefix* and its *c-char-sequence* as defined by [lex.ccon.literal]. The special cases for non-encodable character literals and multicharacter literals take precedence over the base kind. [*Note:* The associated character encoding for ordinary character literals determines encodability, but does not determine the value of non-encodable ordinary character literals or ordinary multicharacter literals. The examples in [lex.ccon.literal] for non-encodable ordinary character literals assume that the specified ~~character~~ Unicode code point lacks representation in the ordinary literal encoding or that encoding the character would require more than one code unit. — *end note*]

Table 10: Character literals

Encoding prefix	Kind	Type	Associated character encoding	Example
none	*ordinary character literal*	`char`	ordinary literal encoding	`'v'`
	non-encodable ordinary character literal	`int`		`'\U0001F525'`
	ordinary multicharacter literal	`int`		`'abcd'`
`L`	*wide character literal*	`wchar_t`	wide literal encoding	`L'w'`
`u8`	*UTF-8 character literal*	`char8_t`	UTF-8	`u8'x'`
`u`	*UTF-16 character literal*	`char16_t`	UTF-16	`u'y'`
`U`	*UTF-32 character literal*	`char32_t`	UTF-32	`U'z'`

In translation phase 4, the value of a *character-literal* is determined using the range of representable values of the *character-literal*'s type in translation phase 7. A non-encodable character literal or a multicharacter literal has an implementation-defined value. The value of any other kind of *character-literal* is determined as follows:

- A *character-literal* with a *c-char-sequence* consisting of a single *basic-c-char*, *simple-escape-sequence*, or *universal-character-name* is the code unit value of the specified ~~character~~ Unicode code point as encoded in the literal's associated character encoding. [*Note:* If the specified ~~character~~ Unicode code point lacks representation in the literal's associated character encoding or if it cannot be encoded as a single code unit, then the literal is a non-encodable character literal. — *end note*]
- A *character-literal* with a *c-char-sequence* consisting of a single *numeric-escape-sequence* has a value as follows:
  - Let v be the integer value represented by the octal number comprising the sequence of *octal-digit*s in an *octal-escape-sequence* or by the hexadecimal number comprising the sequence of *hexadecimal-digit*s in a *hexadecimal-escape-sequence*.
  - If v does not exceed the range of representable values of the *character-literal*'s type, then the value is v.
  - Otherwise, if the *character-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *character-literal*'s type, then the value is the unique value of the *character-literal*'s type `T` that is congruent to v modulo 2^N, where N is the width of `T`.
  - Otherwise, the *character-literal* is ill-formed.

- A *character-literal* with a *c-char-sequence* consisting of a single *conditional-escape-sequence* is conditionally-supported and has an implementation-defined value.

The ~~character~~ Unicode code point specified by a *simple-escape-sequence* is specified in [lex.ccon.esc]. [*Note:* Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C. — *end note*]

Table 11: Simple escape sequences

~~character~~ **Unicode code point**		***simple-escape-sequence***
u+000a	line feed	`\n`
u+0009	character tabulation	`\t`
u+000b	line tabulation	`\v`
u+0008	backspace	`\b`
u+000d	carriage return	`\r`
u+000c	form feed	`\f`
u+0007	alert	`\a`
u+005c	reverse solidus	`\\`
u+003f	question mark	`\?`
u+0027	apostrophe	`\'`
u+0022	quotation mark	`\"`

� Floating-point literals [lex.fcon]

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

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

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

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

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

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

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

sign: one of
  `+ -`

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

floating-point-suffix: one of
  `f l f16 f32 f64 f128 bf16 F L F16 F32 F64 F128 BF16`

The type of a *floating-point-literal* [basic.fundamental,basic.extended.fp] is determined by its *floating-point-suffix* as specified in [lex.fcon.type]. [*Note:* The floating-point suffixes `f16`, `f32`, `f64`, `f128`, `bf16`, `F16`, `F32`, `F64`, `F128`, and `BF16` are conditionally-supported. See **??**. — *end note*]

Table 12: Types of *floating-point-literal*s

floating-point-suffix	**type**
none	`double`
`f` or `F`	`float`
`l` or `L`	`long double`
`f16` or `F16`	`std::float16_t`
`f32` or `F32`	`std::float32_t`
`f64` or `F64`	`std::float64_t`
`f128` or `F128`	`std::float128_t`
`bf16` or `BF16`	`std::bfloat16_t`

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

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

s-char:
  basic-s-char
  escape-sequence
  universal-character-name

basic-s-char:
  any ~~member of the translation character set~~ Unicode code point except the u+0022 quotation mark,
  u+005c reverse solidus, or new-line character

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

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

r-char:
  any~~member of the translation character set~~ Unicode code point, except a u+0029 right parenthesis followed by
  the initial *d-char-sequence* (which may be empty) followed by a u+0022 quotation mark

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

d-char:
  any member of the basic character set except:
  u+0020 space, u+0028 left parenthesis, u+0029 right parenthesis, u+005c reverse solidus,
  u+0009 character tabulation, u+000b line tabulation, u+000c form feed, and new-line

The kind of a *string-literal*, its type, and its associated character encoding[lex.charset] are determined by its encoding prefix and sequence of *s-char* s or *r-char* s as defined by [lex.string.literal] where n is the number of encoded code units as described below.

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~~ Unicode code points as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 ~~characters~~ Unicode code points.

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

Table 13: String literals

Encoding prefix	Kind	Type	Associated character encoding	Examples
none	*ordinary string literal*	array of n `const char`	ordinary literal encoding	`"ordinary string"` `R"(ordinary raw string)"`
`L`	*wide string literal*	array of n `const wchar_t`	wide literal encoding	`L"wide string"` `LR"w(wide raw string)w"`
`u8`	*UTF-8 string literal*	array of n `const char8_t`	UTF-8	`u8"UTF-8 string"` `u8R"x(UTF-8 raw string)x"`
`u`	*UTF-16 string literal*	array of n `const char16_t`	UTF-16	`u"UTF-16 string"` `uR"y(UTF-16 raw string)y"`
`U`	*UTF-32 string literal*	array of n `const char32_t`	UTF-32	`U"UTF-32 string"` `UR"z(UTF-32 raw string)z"`

[*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*]

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

The common *encoding-prefix* for a sequence of adjacent *string-literal* s is determined pairwise as follows: If two *string-literal*s have the same *encoding-prefix*, the common *encoding-prefix* is that *encoding-prefix*. If one *string-literal* has no *encoding-prefix*, the common *encoding-prefix* is that of the other *string-literal*. Any other combinations are ill-formed. [*Note:* A *string-literal*'s rawness has no effect on the determination of the common *encoding-prefix*. — *end note*]

In translation phase 6[lex.phases], adjacent *string-literal* s are concatenated. The lexical structure and grouping of the contents of the individual *string-literal* s is retained. [*Example:*

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

represents the code unit `'\xA'` and the character `'B'` after concatenation (and not the single code unit `'\xAB'`). Similarly,

```
R"(\u00)" "41"
```

represents six characters, starting with a backslash and ending with the digit `1` (and not the single character `'A'` specified by a *universal-character-name*).

[lex.string.concat] has some examples of valid concatenations. — *end example*]

Table 14: 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"

Evaluating a *string-literal* results in a string literal object with static storage duration[basic.stc]. 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 object is undefined. — *end note*]

String literal objects are initialized with the sequence of code unit values corresponding to the *string-literal*'s sequence of *s-char* s (originally from non-raw string literals) and *r-char* s (originally from raw string literals), plus a terminating U+0000 null character, in order as follows:

- The sequence of ~~characters~~ Unicode code points denoted by each contiguous sequence of *basic-s-char* s, *r-char* s, *simple-escape-sequence* s[lex.ccon], and *universal-character-name* s[lex.charset] is encoded to a code unit sequence using the *string-literal*'s associated character encoding. If a ~~character~~ Unicode code point lacks representation in the associated character encoding, then the *string-literal* is conditionally-supported and an implementation-defined code unit sequence is encoded. [*Note:* No ~~character~~ Unicode code point lacks representation in any of the UCS encoding forms. — *end note*] When encoding a stateful character encoding, implementations should encode the first such sequence beginning with the initial encoding state and encode subsequent sequences beginning with the final encoding state of the prior sequence. [*Note:* The encoded code unit sequence can differ from the sequence of code units that would be obtained by encoding each ~~character~~ Unicode code point independently. — *end note*]

- Each *numeric-escape-sequence* [lex.ccon] contributes a single code unit with a value as follows:
  - Let v be the integer value represented by the octal number comprising the sequence of *octal-digit*s in an *octal-escape-sequence* or by the hexadecimal number comprising the sequence of *hexadecimal-digit*s in a *hexadecimal-escape-sequence*.
  - If v does not exceed the range of representable values of the *string-literal*'s array element type, then the value is v.
  - Otherwise, if the *string-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *string-literal*'s array element type, then the value is the unique value of the *string-literal*'s array element type `T` that is congruent to v modulo 2^N, where N is the width of `T`.
  - Otherwise, the *string-literal* is ill-formed.

  When encoding a stateful character encoding, these sequences should have no effect on encoding state.
- Each *conditional-escape-sequence* [lex.ccon] contributes an implementation-defined code unit sequence. When encoding a stateful character encoding, it is implementation-defined what effect these sequences have on encoding state.

� Boolean literals [lex.bool]

boolean-literal:
  `false`
  `true`

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

� Pointer literals [lex.nullptr]

pointer-literal:
  `nullptr`

The pointer literal is the keyword `nullptr`. It has 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 exponent-part$_{opt}$ *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~~ Unicode code points that could match that non-terminal.

A *user-defined-literal* is treated as a call to a literal operator or literal operator template[over.literal]. To determine the form of this call for a given *user-defined-literal* *L* with *ud-suffix* *X*, first let *S* be the set of declarations found by unqualified lookup for the *literal-operator-id* whose literal suffix identifier is *X*[basic.lookup.unqual]. *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[over.literal] 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~~ code point sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain ~~characters~~ code points from the basic 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[over.literal] 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~~ code point sequence $c_1c_2...c_k$. [*Note:* The sequence $c_1c_2...c_k$ can only contain ~~characters~~ code points from the basic 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[over.literal] 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[lex.phases], 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*]

� Basics [basic]

� Preamble [basic.pre]

Two names are *the same* if

- they are *identifier*s composed of the same ~~character~~ Unicode code point sequence, or
- they are *operator-function-id*s formed with the same operator, or
- they are *conversion-function-id*s formed with equivalent[temp.over.link] types, or
- they are *literal-operator-id*s[over.literal] formed with the same literal suffix identifier.

� Preprocessing directives [cpp]

� Preamble [cpp.pre]

[...]

A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: At the start of translation phase 4, the first token in the sequence, referred to as a *directive-introducing token*, begins with the first ~~character~~ Unicode code point in the source file (optionally after whitespace containing no new-line characters) or follows whitespace containing at least one new-line character, and is [...].

� Source file inclusion [cpp.include]

[...]

The implementation shall provide unique mappings for sequences consisting of one or more *nondigit*s or *digit*s[lex.name] followed by a period (.) and a single *nondigit*. The first ~~character~~ Unicode code point shall not be a *digit*. The implementation may ignore distinctions of ~~alphabetical~~ case.

[Editor's note: "case" is mentioned by Unicode. Lets not consider how an implementation can ignore casing in this paper :).]

� Pragma operator [cpp.pragma.op]

A unary operator expression of the form: `_Pragma` (*string-literal*)
is processed as follows: The *string-literal* is *destringized* by deleting the `L` prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of ~~characters~~ Unicode code points is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

� Annex B (normative) Implementation quantities[implimits]

The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

- Nesting levels of parenthesized expressions[expr.prim.paren] within a full-expression [256].
- Number of ~~characters~~ Unicode code points in an internal identifier[lex.name] or macro name[cpp.replace] [1 024].
- Number of ~~characters~~ Unicode code points in an external identifier[lex.name,basic.link] [1 024].
- External identifiers[basic.link] in one translation unit [65 536].
- Identifiers with block scope declared in one block[basic.scope.block] [1 024].

� C++ and ISO C++ 2014 [diff.cpp14]

� ??: lexical conventions [diff.cpp14.lex]

Change: Removal of trigraph support as a required feature.
Rationale: Prevents accidental uses of trigraphs in non-raw string literals and comments.
Effect on original feature: Valid C++ 2014 code that uses trigraphs may not be valid or may have different semantics in this revision of C++. Implementations may choose to translate

trigraphs as specified in C++ 2014 if they appear outside of a raw string literal, as part of the implementation-defined mapping from input source file characters to ~~the translation character set~~ Unicode.

References

[N4892] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4892