# Whitespaces Wording Revamp


| | |
|---|---|
| Document #: | P2348R3 |
| Date: | 2022-09-11 |
| Programming Language C++ | |
| Audience: | EWG, CWG |


Reply-to: Corentin Jabot <corentin.jabot@gmail.com>


## Revisions

### Revision 3

- Fix an example which was accidentally normative
- remplace ”non-whitespace character” by non-whitespace
- typos
- ”Whitespace characters can appear within a preprocessing token only as part of a header name…” is now a note
- Rebase [cpp] on top of P2334R1 [1]
- Rebase [lex.phases] on top of P2223R2 [2]
- Rebase [lex.phases] on top of P2314R4 [6]

### Revision 2

- Wording fixes

### Revision 1

- The previous revisions classified Vertical Tab and Form Feed as vertical spaces. This was consistent with Unicode, but not with the current wording, nor with any of the existing implementations. As this paper is trying not to modify the status quo, and because the author cannot find a reason to challenge the status quo, this version treats these codepoints as horizontal whitespaces.
- Add a note about `\n\r` and other not spelled-out sequences.
- Align the grammar in [lex.whitespaces] with terminology used in P2314R2 [5] and rebase the rest of the wording on top of P2314R2 [5].

## Design

This paper aims to clarify what constitutes a new-line and a whitespace, using Unicode terminology (which is consistent with P2314R1 [4]). No breaking behavior change is intended.

**This paper resolves CWG2002 [9] and CWG1655 [7].**

### Make whitespace grammar elements

For clarity, this paper introduces a grammar for whitespaces, including comments, and then refers to whitespace as grammar terms. This also makes it easier to extend the list of whitespaces later. However, the question of extending the list of Unicode characters treated as whitespace is not explored in this paper. Non-Unicode source encoding would continue to map their set of whitespaces and new-line to what is currently designated in the standard: CR, FF, LF, VT, SPACE, TAB.

### Comments

The proposed wording change makes vertical tabs and form-feed allowed in // comments, rather than make them ill-formed no diagnostic required.

### VT and FF are treated as horizontal whitespaces

In R0, `VT` `FF` were treated as vertical, line-breaking whitespaces. This was consistent with Unicode. However, I failed to realize that all implementations treat them as non-breaking whitespaces. As this paper is not trying to challenge the status quo, which frankly would have little value, it classifies `VT` and `FF` as horizontal whitespaces. As such, an implementation must support them in comments and string/character literals, which is consistent with implementations.

Compiler explorer.

Despite the contradition with Unicode, further research showed that treating vertical space and form feed is standard practice in many languages including C#, Rust, Java, JavaScript and others.

### \n\r

On BBC Micro and other Acorn systems, `\n\r` was used to delimited new lines. Similarly, some systems may not use any of the sequences for new lines outlined in this paper. This is consistent with both the status quo and Unicode. Assuming once could run a modern compiler on a BBC micro (note that this proposal only affects translation, not execution), the implementation-defined mapping provision allows an implementation to map platform/encoding specific line breaks to \n in phase 1 of translation.

Similarly, while this paper tries to preserve sequences of line breaks through phases 1-6 of translation, it is not observable whether a sequence of line breaks is treated as one or

several line-breaks except in the value of `__LINE__` and `source_location` whose values are implementation-defined

### Replace the term *new-line* by *line-break*

`new-line` in the wording refers to both an unspecified character to which all source line terminator map to and a specific, implementation-defined character in the literal character set. Using different terminology makes it clearer which is which.

### Do not perform whitespace replacement

All whitespaces (including comments) are conserved by the proposed wording through phases 1-7. In practice, this is not observable (and we may want a note stating that somewhere).

Note that some compilers (namely GCC) will emit a diagnostic for the presence of vertical tabs in preprocessing directives. With this, such diagnostic in pedantic mode is not necessary for conformance, but might still be useful [Compiler Explorer].

This fixes CWG2002 [9].

> According to Clause 15 [cpp] paragraphg 4,
>
> > The only whitespace characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other whitespace characters in translation phase 3).
>
> The effect of this restriction is unclear, however, since translation phase 3 is permitted to transform all white space characters and comments into spaces. The relationship between these two rules should be clarified.

The list of whitespaces is sufficient to address the needs of codepoints conservations through phases 1-6 of P2295R2 [3]. The clause mentioned in CWG2002 is deleted.

### The set of character is not expanded

We do not propose new characters as whitespaces. However, following Unicode guidelines, we clarify that CRLF is to be considered a single line break.

### Raw-string literals

For clarity, that line-breaks are mapped to a new-line in raw-string-literal is made normative. We intend that this paper addresses CWG1655 [7]: by using different terms for *line-break* and *new-line*, we make clear that these things are not related.

## Wording

### [lex.phases] Phases of translation

[Editor's note: The wording is based on P2314R2 [5]]

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

1. ~~Physical source file characters are mapped, in an implementation-defined manner, to the translation character set (introducing new-line characters for end-of-line indicators)~~ The physical source file is mapped, in an implementation-defined manner, to a sequence of translation character set elements.

   [Editor's note: The intent of this reformulation is to get rid of the term "end-of-line-indicator" which is not defined, while supporting source files stored as records.]

   The set of physical source file characters accepted is implementation-defined.

2. Each sequence of a backslash character (\) immediately followed by zero or more ~~whitespace characters other than new-line~~ *horizontal-whitespace-character*s followed by a ~~new-line character~~ *line-break* 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 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 splice, shall be processed as if an additional ~~new-line character~~ *line-break* were appended to the file.

3. The source file is decomposed into preprocessing tokens and ~~sequences of whitespace characters (including comments)~~ *whitespaces*. A source file shall not end in a partial preprocessing token or in a partial comment. ~~Each comment is replaced by one 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 is unspecified.~~ As characters 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~~ *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. 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-c-char*, *basic-s-char*, and *r-char* 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 encoded in the literal's associated character encoding as specified in **??** and **??**.

6. Adjacent *string-literal* s are concatenated and a null character is appended to the result as specified in **??**.

7. The ~~Whitespace characters~~ *whitespaces* separating tokens are no longer significant. Each preprocessing 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 can 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*]

## 🔲 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~~ character that is not part of a *whitespace* and ~~cannot be~~ is not part of 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. 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~~ *whitespace* 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. Prepro-

cessing tokens can be separated by ~~whitespace;~~ *whitespaces*. ~~this consists of comments, 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 [cpp], in certain circumstances during translation phase 4, ~~whitespace~~ *whitespaces* (or the absence thereof) serves as more than preprocessing token separation.

[Note:

~~Whitespace~~ *whitespaces* and other whitespace characters 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

–end note].

[Editor's note: "Whitespace characters" is intentional here: this may refer to unicode characters not considered whitespaces for the purpose of lexing, and do not refer to elements used to separate tokens.]

## Tokens [lex.token]

*token:*
*identifier*
*keyword*
*literal*
*operator-or-punctuator*

~~There are five kinds of tokens: identifiers, keywords, literals,operators, and other separators~~.

[Editor's note: This is somewhat of a drive-by partial fix of CWG1901 [8]. However the use of "separator" is somewhat misleading here. Was it meant to be mean whitespace? The following sentence is explicitely stating that whitespace are not tokens. ]

Blanks, horizontal and vertical tabs, new-lines, form-feeds, and comments (collectively, "whitespace"), as described below, are ignored except as they serve to separate tokens. [*Note:* Some ~~whitespace~~ *whitespace* is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note* ]

## Whitespaces [lex.whitespaces]

*whitespace:*
*horizontal-whitespace*
*line-break*

*horizontal-whitespace:*
*horizontal-whitespace-character*
*comment*

*comment:*
*single-line-comment*
*multi-line-comment*

*single-line-comment:*
//
*single-line-comment single-line-comment-elem*

*single-line-comment-elem:*
any member of the translation character set except *line-break-character*

*multi-line-comment:*
/* *multi-line-comment-elem-seq*$_{opt}$ */

*multi-line-comment-elem-seq:*
*multi-line-comment-elem*
*multi-line-comment-elem-seq multi-line-comment-elem*

*multi-line-comment-elem:*
any member of the translation character set except * immediately followed by /

*line-break:*
*line-break-character*
U+000D CARRIAGE RETURN immediately followed by U+000A LINE FEED

*line-break-character:*
U+000A LINE FEED
U+000D CARRIAGE RETURN

*horizontal-whitespace-character:*
U+0009 HORIZONTAL TAB
U+000C FORM FEED
U+000B VERTICAL TAB
U+0020 SPACE

A *whitespace* is the longest sequence of characters that could constitute a *whitespace*.

[ *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* ]

## Comments [lex.comment]

~~The characters `/*` start a comment, which terminates with the characters `*/`. 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. 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 source character set except ~~new-line~~ *line-break-character* and >

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

*q-char:*
any member of the source character set except ~~new-line~~ *line-break-character* 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*.

## 🔲 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*

*basic-c-char:*
any member of the basic source character set except the single-quote `'`, back-slash `\`, or any character that matches ~~new-line character~~ *line-break-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*

*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*

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

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

//...

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

Table 1: Simple escape sequences

| | | |
|---|---|---|
| new-line | NL(LF) | `\n` |
| horizontal tab | HT | `\t` |
| vertical tab | VT | `\v` |
| backspace | BS | `\b` |
| carriage return | CR | `\r` |
| form feed | FF | `\f` |
| alert | BEL | `\a` |
| backslash | \ | `\\` |
| question mark | ? | `\?` |
| single quote | ' | `\'` |
| double quote | " | `\"` |

## 🔲 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 except the double-quote ", backslash \, or any character that matches ~~new-line character~~ *line-break-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 source character set, 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 except:
~~space~~ *any charachter that matches horizontal-whitespace-character,* *or* ~~*new-line character*~~ *line-break-character*, 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~~ Each longest sequence of characters that matches the grammar of a ~~source-file new-line~~ *line-break* in a raw string literal ~~results in a new-line in the resulting execution string literal~~ denotes a new-line.

*[Example:* 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]*

[*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* ]

# [cpp]

## ~~[cpp.pre]~~ Preamble [cpp.pre]

*preprocessing-file:*
*group*$_{opt}$
*module-file*

*module-file:*
*pp-global-module-fragment*$_{opt}$ *pp-module group*$_{opt}$ *pp-private-module-fragment*$_{opt}$

*pp-global-module-fragment:*
`module` ; ~~*new-line*~~ *line-break group*$_{opt}$

*pp-private-module-fragment:*
`module` : `private` ; ~~*new-line*~~ *line-break group*$_{opt}$

*group:*
*group-part*
*group group-part*

*group-part:*
*control-line*
*if-section*
*text-line*
*# conditionally-supported-directive*

*control-line:*

`# include` *pp-tokens* ~~*new-line*~~ *line-break*

*pp-import*

`# define` *identifier replacement-list* ~~*new-line*~~ *line-break*

`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list* ~~*new-line*~~ *line-break*

`# define` *identifier lparen* `...` `)` *replacement-list* ~~*new-line*~~ *line-break*

`# define` *identifier lparen identifier-list* `,` `...` `)` *replacement-list* ~~*new-line*~~ *line-break*

`# undef` *identifier* ~~*new-line*~~ *line-break*

`# line` *pp-tokens* ~~*new-line*~~ *line-break*

`# error` *pp-tokens*$_{opt}$ ~~*new-line*~~ *line-break*

`# pragma` *pp-tokens*$_{opt}$ ~~*new-line*~~ *line-break*

`#` ~~*new-line*~~ *line-break*

*if-section:*

*if-group elif-groups*$_{opt}$ *else-group*$_{opt}$ *endif-line*

*if-group:*

`# if` *constant-expression* ~~*new-line*~~ *line-break group*$_{opt}$

`# ifdef` *identifier* ~~*new-line*~~ *line-break group*$_{opt}$

`# ifndef` *identifier* ~~*new-line*~~ *line-break group*$_{opt}$

*elif-groups:*

*elif-group*

*elif-groups elif-group*

*elif-group:*

`# elif` *constant-expression* ~~*new-line*~~ *line-break group*$_{opt}$

`# elifdef` *constant-expression* ~~*new-line*~~ *line-break group*$_{opt}$

`# elifndef` *constant-expression* ~~*new-line*~~ *line-break group*$_{opt}$

*else-group:*

`# else` ~~*new-line*~~ *line-break group*$_{opt}$

*endif-line:*

`# endif` ~~*new-line*~~ *line-break*

*elif-group:*

`# elif` *constant-expression* ~~*new-line*~~ *line-break group*$_{opt}$

*else-group:*

`# else` ~~*new-line*~~ *line-break group*$_{opt}$

*endif-line:*

`# endif` ~~*new-line*~~ *line-break*

*text-line:*

*pp-tokens*$_{opt}$ ~~*new-line*~~ *line-break*

*conditionally-supported-directive:*

*pp-tokens* ~~*new-line*~~ *line-break*

*lparen:*

a `(` character not immediately preceded by ~~whitespace~~ *whitespace*

*identifier-list:*

*identifier*

*identifier-list* `,` *identifier*

*replacement-list:*
*pp-tokens*$_{opt}$

*pp-tokens:*
*preprocessing-token*
*pp-tokens preprocessing-token*

*new-line:*
the new-line character

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 in the source file (optionally after whitespace containing no new-line characters)~~ non-whitespace or follows a sequence of ~~whitespace~~ *whitespaces* containing at least one ~~new-line character~~ *line-break*, and is

- a `#` preprocessing token, or
- an `import` preprocessing token immediately followed on the same logical line by a *header-name*, `<`, *identifier*, *string-literal*, or `:` preprocessing token, or
- a `module` preprocessing token immediately followed on the same logical line by an *identifier*, `:`, or `;` preprocessing token, or
- an `export` preprocessing token immediately followed on the same logical line by one of the two preceding forms.

The last token in the sequence is the first token within the sequence that is immediately followed by ~~whitespace~~ *whitespaces* containing a ~~new-line character~~ *line-break*. [*Note:* Thus, preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all ~~whitespace~~ *whitespace* is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in **??**, for example). — *end note*] [*Note:* A ~~new-line character~~ *line-break* ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro. — *end note*]

[*Example:*

```
#                       // preprocessing directive
module ;                // preprocessing directive
export module leftpad;  // preprocessing directive
import <string>;        // preprocessing directive
export import "squee";  // preprocessing directive
import rightpad;        // preprocessing directive
import :part;           // preprocessing directive

module                  // not a preprocessing directive
;                       // not a preprocessing directive

export                  // not a preprocessing directive
import                  // not a preprocessing directive
foo;                    // not a preprocessing directive

export                  // not a preprocessing directive
```

```
import foo;             // preprocessing directive (ill-formed at phase 7)

import ::               // not a preprocessing directive
import ->               // not a preprocessing directive
```

— *end example* ]

A sequence of preprocessing tokens is only a *text-line* if it does not begin with a directive-introducing token. A sequence of preprocessing tokens is only a *conditionally-supported-directive* if it does not begin with any of the directive names appearing after a `#` in the syntax. A *conditionally-supported-directive* is conditionally-supported with implementation-defined semantics.

At the start of phase 4 of translation, the *group* of a *pp-global-module-fragment* shall contain neither a *text-line* nor a *pp-import*.

When in a group that is skipped, the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following ~~new-line character~~ *line-break* .

The only whitespace characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the directive-introducing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other whitespace characters in translation phase 3).

Only *horizontal-whitespace*s can appear between preprocessing tokens within a preprocessing directive.

[Editor's note: multi-line comments containing line-breaks are considered atomic *horizontal-whitespace*s after phase 3]

The implementation can process and skip sections of source files conditionally, include other source files, import macros from header units, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[ *Example:* In:

```
#define EMPTY
EMPTY   #   include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a `#` at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced. — *end example* ]

## 🔀 Conditional inclusion [cpp.cond]

*defined-macro-expression:*
`defined` *identifier*
`defined (` *identifier* `)`

*h-preprocessing-token:*
any *preprocessing-token* other than >

*h-pp-tokens:*
*h-preprocessing-token*
*h-pp-tokens h-preprocessing-token*

*header-name-tokens:*
*string-literal*
< *h-pp-tokens* >

*has-include-expression:*
`__has_include (` *header-name* `)`
`__has_include (` *header-name-tokens* `)`

*has-attribute-expression:*
`__has_cpp_attribute (` *pp-tokens* `)`

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below [1] ecause the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc. and it may contain zero or more *defined-macro-expression*s and/or *has-include-expression*s and/or *has-attribute-expression*s as unary operator expressions.

A *defined-macro-expression* evaluates to `1` if the identifier is currently defined as a macro name (that is, if it is predefined or if it has one or more active macro definitions, for example because it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), `0` if it is not.

The second form of *has-include-expression* is considered only if the first form does not match, in which case the preprocessing tokens are processed just as in normal text.

The header or source file identified by the parenthesized preprocessing token sequence in each contained *has-include-expression* is searched for as if that preprocessing token sequence were the *pp-tokens* in a `#include` directive, except that no further macro expansion is performed. If such a directive would not satisfy the syntactic requirements of a `#include` directive, the program is ill-formed. The *has-include-expression* evaluates to `1` if the search for the source file succeeds, and to `0` if the search fails.

Each *has-attribute-expression* is replaced by a non-zero *pp-number* matching the form of an *integer-literal* if the implementation supports an attribute with the name specified by interpreting the *pp-tokens*, after macro expansion, as an *attribute-token*, and by `0` otherwise. The program is ill-formed if the *pp-tokens* do not match the form of an *attribute-token*.

[1] B

For an attribute specified in this document, the value of the *has-attribute-expression* is given by . For other attributes recognized by the implementation, the value is implementation-defined. [ *Note:* It is expected that the availability of an attribute can be detected by any non-zero result. — *end note* ]

Table 2: `__has_cpp_attribute` values

| Attribute | Value |
|---|---|
| `carries_dependency` | `200809L` |
| `deprecated` | `201309L` |
| `fallthrough` | `201603L` |
| `likely` | `201803L` |
| `maybe_unused` | `201603L` |
| `no_unique_address` | `201803L` |
| `nodiscard` | `201907L` |
| `noreturn` | `200809L` |
| `unlikely` | `201803L` |

The `#ifdef` and `#ifndef` directives, and the `defined` conditional inclusion operator, shall treat `__has_include` and `__has_cpp_attribute` as if they were the names of defined macros. The identifiers `__has_include` and `__has_cpp_attribute` shall not appear in any context not mentioned in this subclause.

Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token.

Preprocessing directives of the forms

`# if` *constant-expression* ~~*new-line*~~ *line-break* *group*$_{opt}$
`# elif` *constant-expression* ~~*new-line*~~ *line-break* *group*$_{opt}$

check whether the controlling constant expression evaluates to nonzero.

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined.

After all replacements due to macro expansion and evaluations of *defined-macro-expression* s, *has-include-expression* s, and *has-attribute-expression* s have been performed, all remaining identifiers and keywords, except for `true` and `false`, are replaced with the *pp-number* `0`, and then each preprocessing token is converted into a token. [ *Note:* An alternative token is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement. — *end note* ]

The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of **??** using arithmetic that has at least the ranges specified in **??**. For the purposes of this token conversion and evaluation all signed and unsigned integer types act as if they have the same representation as, respectively, `intmax_t` or `uintmax_t`. [ *Note:* Thus on an implementation where `std::numeric_limits<int>::max()` is `0x7FFF` and `std::numeric_-`

`limits<unsigned int>::max()` is `0xFFFF`, the integer literal `0x8000` is signed and positive within a `#if` expression even though it is unsigned in translation phase 7. — *end note* ] This includes interpreting *character-literal* s, which may involve converting escape sequences into execution character set members. Whether the numeric value for these *character-literal* s matches the value obtained when an identical *character-literal* occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined. [ *Note:* Thus, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in these two contexts:

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

— *end note* ] Also, whether a single-character *character-literal* may have a negative value is implementation-defined. Each subexpression with type `bool` is subjected to integral promotion before processing continues.

Preprocessing directives of the forms

`# ifdef` *identifier* ~~*new-line*~~ *line-break* *group*$_{opt}$
`# ifndef` *identifier* ~~*new-line*~~ *line-break* *group*$_{opt}$
`# elifdef` *identifier* ~~*new-line*~~ *line-break* *group*$_{opt}$
`# elifndef` *identifier* ~~*new-line*~~ *line-break* *group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined` *identifier*, `#if !defined` *identifier*, `#elif defined` *identifier*, and `#elif !defined` *identifier*, respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[2]s indicated by the syntax, a preprocessing token cannot follow a `#else` or `#endif` directive before the terminating ~~new-line character~~ *line-break* . However, comments can appear anywhere in a source file, including within a preprocessing directive.

[ *Example:* This demonstrates a way to include a library `optional` facility only if it is available:

```
#if __has_include(<optional>)
#  include <optional>
#  if __cpp_lib_optional >= 201603
#    define have_optional 1
#  endif
#elif __has_include(<experimental/optional>)
#  include <experimental/optional>
#  if __cpp_lib_experimental_optional >= 201411
#    define have_optional 1
#    define experimental_optional 1
```

[2]A

```
#  endif
#endif
#ifndef have_optional
#  define have_optional 0
#endif
```

— *end example* ]

[ *Example:* This demonstrates a way to use the attribute `[[acme::deprecated]]` only if it is available.

```
#if __has_cpp_attribute(acme::deprecated)
#  define ATTR_DEPRECATED(msg) [[acme::deprecated(msg)]]
#else
#  define ATTR_DEPRECATED(msg) [[deprecated(msg)]]
#endif
ATTR_DEPRECATED("This function is deprecated") void anvil();
```

— *end example* ]

## 🔲 Source file inclusion [cpp.include]

A `#include` directive shall identify a header or source file that can be processed by the implementation.

A preprocessing directive of the form `# include <` *h-char-sequence* `>` ~~*new-line*~~ *line-break* searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

A preprocessing directive of the form `# include "` *q-char-sequence* `"` ~~*new-line*~~ *line-break* causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read `# include <` *h-char-sequence* `>` ~~*new-line*~~ *line-break* with the identical contained sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form `# include` *pp-tokens* ~~*new-line*~~ *line-break* (that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined. [3]ote that adjacent *string-literal* s are not concatenated into a single

[3] N

*string-literal* (see the translation phases in **??**); thus, an expansion that results in two *string-literal* s is an invalid directive. The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

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

A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

If the header identified by the *header-name* denotes an importable header, it is implementation-defined whether the `#include` preprocessing directive is instead replaced by an `import` directive of the form `import` *header-name* ; ~~*new-line*~~ *line-break*

[ *Note:* An implementation can provide a mechanism for making arbitrary source files available to the < > search. However, using the < > form for headers provided with the implementation and the " " form for sources outside the control of the implementation achieves wider portability. For instance:

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

— *end note* ]

[ *Example:* This illustrates macro-replaced `#include` directives:

```
#if VERSION == 1
#define INCFILE  "vers1.h"
#elif VERSION == 2
#define INCFILE  "vers2.h"  // and so on
#else
#define INCFILE  "versN.h"
#endif
#include INCFILE
```

— *end example* ]

## ❓ Module directive [cpp.module]

*pp-module:*
*export*$_{opt}$ `module` *pp-tokens*$_{opt}$ ; ~~*new-line*~~ *line-break*

A *pp-module* shall not appear in a context where `module` or (if it is the first token of the *pp-module*) `export` is an identifier defined as an object-like macro.

Any preprocessing tokens after the `module` preprocessing token in the `module` directive are processed just as in normal text. [*Note:* Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens. — *end note*]

The `module` and `export` (if it exists) preprocessing tokens are replaced by the *module-keyword* and *export-keyword* preprocessing tokens respectively. [*Note:* This makes the line no longer a directive so it is not removed at the end of phase 4. — *end note*]

## 🔲 Header unit importation [cpp.import]

*pp-import:*
*export*$_{opt}$ *import* *header-name* *pp-tokens*$_{opt}$ ; ~~*new-line*~~ *line-break*
*export*$_{opt}$ *import* *header-name-tokens* *pp-tokens*$_{opt}$ ; ~~*new-line*~~ *line-break*
*export*$_{opt}$ *import* *pp-tokens* ; ~~*new-line*~~ *line-break*

A *pp-import* shall not appear in a context where `import` or (if it is the first token of the *pp-import*) `export` is an identifier defined as an object-like macro.

The preprocessing tokens after the `import` preprocessing token in the `import` *control-line* are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). [*Note:* An `import` directive matching the first two forms of a *pp-import* instructs the preprocessor to import macros from the header unit denoted by the *header-name*, as described below. — *end note*] The *point of macro import* for the first two forms of *pp-import* is immediately after the ~~*new-line*~~ *line-break* terminating the *pp-import*. The last form of *pp-import* is only considered if the first two forms did not match, and does not have a point of macro import.

//...

## 🔲 Macro replacement [cpp.replace]

### 🔲 General [cpp.replace.general]

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and ~~whitespace~~ *whitespace* separation, where all ~~whitespace separations~~ sequences of one or more *whitespaces* are considered identical.

An identifier currently defined as an object-like macro (see below) may be redefined by another `#define` preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a function-like macro (see below) may be redefined by another `#define` preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

[*Example:* The following sequence is valid:

```
#define OBJ_LIKE      (1-1)
```

```
#define OBJ_LIKE      /* whitespace */ (1-1) /* other */
#define FUNC_LIKE(a)   ( a )
#define FUNC_LIKE( a )(     /* note the whitespace */ \
a /* other stuff on this line
*/ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)         // different token sequence
#define OBJ_LIKE    (1 - 1)     // different whitespace
#define FUNC_LIKE(b) ( a )      // different parameter usage
#define FUNC_LIKE(b) ( b )      // different parameter spelling
```

— *end example* ]

There shall be ~~whitespace~~ one or more *whitespaces* between the identifier and the replacement list in the definition of an object-like macro.

If the *identifier-list* in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be at least as many arguments in the invocation as there are parameters in the macro definition (excluding the `...`). There shall exist a `)` preprocessing token that terminates the invocation.

The identifiers `__VA_ARGS__` and `__VA_OPT__` shall occur only in the *replacement-list* of a function-like macro that uses the ellipsis notation in the parameters.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any ~~whitespace characters~~ *whitespaces* preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a `#` preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive can begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form `# define` *identifier replacement-list* ~~*new-line*~~ *line-break* defines an *object-like macro* that causes each subsequent instance of the macro name [4]ince, by macro-replacement time, all *character-literal* s and *string-literal* s are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see **??**, translation phases), they are never scanned for macro names or parameters.  to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. [5]n alternative token is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.  The replacement list is then rescanned for more macro names as specified below.

[4] S
[5] A

[ *Example:* The simplest use of this facility is to define a “manifest constant”, as in

```
#define TABSIZE 100
int table[TABSIZE];
```

— *end example* ]

A preprocessing directive of the form

`#` `define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list* ~~*new-line*~~ *line-break*
`#` `define` *identifier lparen* `...` `)` *replacement-list* ~~*new-line*~~ *line-break*
`#` `define` *identifier lparen identifier-list* `,` `...` `)` *replacement-list* ~~*new-line*~~ *line-break*

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, ~~new-line is considered a normal whitespace character~~ *line-break* is considered like other *whitespaces*.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, [6] *conditionally-supported-directive* is a preprocessing directive regardless of whether the implementation supports it. the behavior is undefined.

[ *Example:* The following defines a function-like macro whose value is the maximum of its arguments. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.
— *end example* ]

If there is a `...` immediately preceding the `)` in the function-like macro definition, then the trailing arguments (if any), including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is either equal to or one more than the number of parameters in the macro definition (excluding the `...`).

[6] A

### The # operator [cpp.stringize]

Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the *stringizing argument* be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each ~~occurrence of whitespace~~ sequence of one or more *whitespaces* between the stringizing argument's preprocessing tokens becomes a single space character in the character string literal. All ~~whitespace~~ *whitespaces* before the first preprocessing token and after the last preprocessing token comprising the stringizing argument ~~is~~ are deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of *string-literal* s and *character-literal* s: a \ character is inserted before each " and \ character of a *character-literal* or *string-literal* (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty stringizing argument is "". The order of evaluation of # and ## operators is unspecified.

### The ## operator [cpp.concat]

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead. [7]lacemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

[ *Example:* The sequence

```
#define str(s)      # s
```

[7]P

```
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
x ## s, x ## t)
#define INCFILE(n)  vers ## n
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4')        // this goes away
== 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello, world"
```

The presence of ~~Space~~ *whitespace* around the # and ## tokens in the macro definition is optional. — *end example* ]

[ *Example:* In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y);          // equivalent to char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
```

```
"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the `##` operator. — *end example* ]

[ *Example:* To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
    t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
    10, 11, 12, };
```

— *end example* ]

### 🔲 Scope of macro definitions [cpp.scope]

A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase 4.

A preprocessing directive of the form `# undef` *identifier* ~~*new-line*~~ *line-break*
causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

## 🔲 Line control [cpp.line]

The *string-literal* of a `#line` directive, if present, shall be a character string literal.

The *line number* of the current source line is one greater than the number of ~~new-line characters~~ *line-breaks* ~~read or introduced in~~ resulting from translation phase 1 while processing the source file to the current token.

A preprocessing directive of the form `# line` *digit-sequence* ~~*new-line*~~ *line-break*
causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

A preprocessing directive of the form `# line` *digit-sequence* " *s-char-sequence*$_{opt}$ " ~~*new-line*~~ *line-break*
sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

A preprocessing directive of the form `# line` *pp-tokens* ~~*new-line*~~ *line-break*
(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

## 🔲 Error directive [cpp.error]

A preprocessing directive of the form `# error` *pp-tokens*$_{opt}$ ~~*new-line*~~ *line-break*
causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

## 🔲 Pragma directive [cpp.pragma]

A preprocessing directive of the form `# pragma` *pp-tokens*$_{opt}$ ~~*new-line*~~ *line-break*
causes the implementation to behave in an implementation-defined manner. The behavior may cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

## 🔲 Null directive [cpp.null]

A preprocessing directive of the form `#` ~~*new-line*~~ *line-break*
has no effect.

## Acknowledgments

Thanks to Peter Brett, Hubert Tong, Jens Maurer and the entierty of SG-16 who provided valuable feedback!

## References

[1] Melanie Blower. P2334R1: Add support for preprocessing directives elifdef and elifndef. https://wg21.link/p2334r1, 4 2021.

[2] Corentin Jabot. P2223R2: Trimming whitespaces before line splicing. https://wg21.link/p2223r2, 4 2021.

[3] Corentin Jabot. P2295R2: Support for utf-8 as a portable source file encoding. https://wg21.link/p2295r2, 4 2021.

[4] Jens Maurer. P2314R1: Character sets and encodings. https://wg21.link/p2314r1, 3 2021.

[5] Jens Maurer. P2314R2: Character sets and encodings. https://wg21.link/p2314r2, 5 2021.

[6] Jens Maurer. P2314R4: Character sets and encodings. https://wg21.link/p2314r4, 10 2021.

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

[8] Richard Smith. CWG1901: punctuator referenced but not defined. https://wg21.link/cwg1901, 3 2014.

[9] Richard Smith. CWG2002: White space within preprocessing directives. https://wg21.link/cwg2002, 9 2014.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* new-line https://wg21.link/N4885