6.10 Preprocessing directives

Syntax

```
1
              preprocessing-file:
                      group<sub>opt</sub>
              group:
                      group-part
                      group group-part
              group-part:
                      if-section
                      control-line
                      text-line
                      # non-directive
              if-section:
                      if-group elif-groups<sub>opt</sub> else-group<sub>opt</sub> endif-line
              if-group:
                                     constant-expression new-line group<sub>opt</sub>
                      # if
                                     identifier new-line group<sub>opt</sub>
                      # ifdef
                      # ifndef identifier new-line group opt
              elif-groups:
                      elif-group
                      elif-groups elif-group
              elif-group:
                                     constant-expression new-line group<sub>opt</sub>
                      # elif
              else-group:
                      # else
                                     new-line group<sub>opt</sub>
              endif-line:
```

endif new-line

```
control-line:
        # include pp-tokens new-line
        # define
                      identifier replacement-list new-line
                      identifier lparen identifier-list<sub>opt</sub>)
          define
                                             replacement-list new-line
          define
                      identifier lparen ... ) replacement-list new-line
        # define
                      identifier lparen identifier-list , ... )
                                             replacement-list new-line
        # undef
                       identifier new-line
          line
                      pp-tokens new-line
          error
                      pp-tokens<sub>opt</sub> new-line
                      pp-tokens<sub>opt</sub> new-line
          pragma
                      new-line
text-line:
       pp-tokens<sub>opt</sub> new-line
non-directive:
       pp-tokens new-line
lparen:
       a ( character not immediately preceded by white-space
replacement-list:
       pp-tokens<sub>opt</sub>
pp-tokens:
       preprocessing-token
       pp-tokens preprocessing-token
new-line:
        the new-line character
```

Description

A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence. A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an

¹⁶⁵⁾ Thus, preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 6.10.3.2, for example).

invocation of a function-like macro.

- A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.
- When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

The only white-space 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 white-space characters in translation phase 3).

Semantics

- The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.
- 7 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.
- 8 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.

6.10.1 Conditional inclusion

Constraints

or

The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below; ¹⁶⁶⁾ and it may contain unary operator expressions of the form

```
defined identifier
defined ( identifier )
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is

¹⁶⁶⁾ Because 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.

predefined or if 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.

2 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 (6.4).

Semantics

- 3 Preprocessing directives of the forms
 - **if** constant-expression new-line group_{ont}
 - # elif constant-expression new-line 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 4 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 the **defined** unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types intmax t and uintmax_t defined in the header <stdint.h>. 167) This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined. Also, whether a single-character character constant may have a negative value is implementation-defined.

¹⁶⁷⁾ Thus, on an implementation where **INT_MAX** is **Ox7FFF** and **UINT_MAX** is **OxFFFF**, the constant **Ox8000** is signed and positive within a **#if** expression even though it would be unsigned in translation phase 7.

¹⁶⁸⁾ 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.

- 5 Preprocessing directives of the forms
 - # ifdef identifier new-line group opt
 - # ifndef identifier new-line group_{opt}

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and **#if !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. 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. 169)

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.20.1.5).

6.10.2 Source file inclusion

Constraints

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

Semantics

- 2 A preprocessing directive of the form
 - # include <h-char-sequence> new-line

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.

- 3 A preprocessing directive of the form
 - # include "q-char-sequence" new-line

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

¹⁶⁹⁾ As indicated by the syntax, a preprocessing token shall not follow a **#else** or **#endif** directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

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

with the identical contained sequence (including > characters, if any) from the original directive.

- 4 A preprocessing directive of the form
 - # include pp-tokens new-line

(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. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms. 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 nondigits or digits (6.4.2.1) followed by a period (•) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.
- 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 (see 5.2.4.1).
- 7 EXAMPLE 1 The most common uses of **#include** preprocessing directives are as in the following:

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

¹⁷⁰⁾ Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

8 EXAMPLE 2 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
```

Forward references: macro replacement (6.10.3).

6.10.3 Macro replacement

Constraints

- 1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.
- An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **#define** preprocessing directive unless 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.
- 3 There shall be white-space 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 more arguments in the invocation than there are parameters in the macro definition (excluding the ...). There shall exist a) preprocessing token that terminates the invocation.
- 5 The identifier __**VA_ARGS**__ 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.

Semantics

7 The identifier immediately following the **define** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list

for either form of macro.

- 8 If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.
- 9 A preprocessing directive of the form
 - # define identifier replacement-list new-line

defines an *object-like macro* that causes each subsequent instance of the macro name¹⁷¹⁾ to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

- 10 A preprocessing directive of the form
 - # define identifier lparen identifier-list_{opt}) replacement-list new-line
 - # define identifier lparen ...) replacement-list new-line
 - # define identifier lparen identifier-list , ...) replacement-list new-line

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, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. 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 white-space character.

- 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, ¹⁷²⁾ the behavior is undefined.
- 12 If there is a . . . in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item:

¹⁷¹⁾ Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

¹⁷²⁾ Despite the name, a non-directive is a preprocessing directive.

the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the . . .).

6.10.3.1 Argument substitution

- After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.
- An identifier __va_args__ that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

6.10.3.2 The # operator

Constraints

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.

Semantics

2 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. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of # and ## operators is unspecified.

6.10.3.3 The ## operator

Constraints

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

Semantics

- 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. ¹⁷³⁾
- 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.
- 4 EXAMPLE In the following fragment:

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.

¹⁷³⁾ Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

6.10.3.4 Rescanning and further replacement

- After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemarker preprocessing tokens are removed. The resulting preprocessing token sequence is then rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.
- If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.
- 3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.
- 4 EXAMPLE There are cases where it is not clear whether a replacement is nested or not. For example, given the following macro definitions:

```
#define f(a) a*g
#define g(a) f(a)

the invocation
f(2)(9)

may expand to either
2*f(9)
```

2*9*a

or

Strictly conforming programs are not permitted to depend on such unspecified behavior.

6.10.3.5 Scope of macro definitions

- 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 preprocessing translation unit. Macro definitions have no significance after translation phase 4.
- 2 A preprocessing directive of the form

```
# undef identifier new-line
```

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.

3 EXAMPLE 1 The simplest use of this facility is to define a "manifest constant", as in

```
#define TABSIZE 100
```

int table[TABSIZE];

4 EXAMPLE 2 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. 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.

5 EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x
      #define f(a)
                      f(x * (a))
      #undef x
      #define x
      #define q
                      f
      #define z
                      z[0]
      #define h
                      g(~
      #define m(a)
                      a(w)
      #define w
                      0,1
      #define t(a)
                      а
      #define p()
                      int
      #define q(x)
                      x
      \#define r(x,y) x \#\# y
      #define str(x) # x
      f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
      g(x+(3,4)-w) \mid h = 5 & m
             (f)^m(m);
      p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
      char c[2][6] = { str(hello), str() };
results in
      f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
      f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
      int i[] = { 1, 23, 4, 5, };
      char c[2][6] = \{ \text{ "hello"},
```

6 EXAMPLE 4 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

7

8

9

```
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);
          "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
       #include "vers2.h"
                                (after macro replacement, before file access)
       "hello";
       "hello, world"
Space around the # and ## tokens in the macro definition is optional.
EXAMPLE 5 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, };
EXAMPLE 6 To demonstrate the redefinition rules, the following sequence is valid.
       #define OBJ LIKE
                                 (1-1)
       #define OBJ LIKE
                                 /* white space */ (1-1) /* other */
       #define FUNC_LIKE(a)
                                   (a)
       #define FUNC_LIKE( a )( /* note the white space */ \
                                    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 white space
       #define FUNC_LIKE(b) ( a ) // different parameter usage
       #define FUNC_LIKE(b) ( b ) // different parameter spelling
EXAMPLE 7 Finally, to show the variable argument list macro facilities:
```

```
#define debug(...)
                              fprintf(stderr, __VA_ARGS__)
      #define showlist(...)
                              puts(#__VA ARGS_
      #define report(test, ...) ((test)?puts(#test):\
                  printf(__VA_ARGS___))
      debug("Flag");
      debug("X = %d\n", x);
      showlist(The first, second, and third items.);
      report(x>y, "x is %d but y is %d", x, y);
results in
                       "Flag" );
      fprintf(stderr,
                       "X = %d\n", x);
      fprintf(stderr,
      puts( "The first, second, and third items." );
      ((x>y)?puts("x>y"):
                  printf("x is %d but y is %d", x, y));
```

6.10.4 Line control

Constraints

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

Semantics

- The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token.
- 3 A preprocessing directive of the form
 - # line digit-sequence new-line

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). The digit sequence shall not specify zero, nor a number greater than 2147483647.

- 4 A preprocessing directive of the form
 - # line digit-sequence "s-char-sequence_{opt}" new-line

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

- 5 A preprocessing directive of the form
 - # line pp-tokens new-line

(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). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

6.10.5 Error directive

Semantics

1 A preprocessing directive of the form

```
# error pp-tokens<sub>opt</sub> new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

6.10.6 Pragma directive

Semantics

1 A preprocessing directive of the form

```
# pragma pp-tokens<sub>opt</sub> new-line
```

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)¹⁷⁴⁾ causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored.

If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms¹⁷⁵⁾ whose meanings are described elsewhere:

```
#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
on-off-switch: one of
ON OFF DEFAULT
```

Forward references: the FP_CONTRACT pragma (7.12.2), the FENV_ACCESS pragma (7.6.1), the CX_LIMITED_RANGE pragma (7.3.4).

¹⁷⁴⁾ An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

¹⁷⁵⁾ See "future language directions" (6.11.8).

6.10.7 Null directive

Semantics

1 A preprocessing directive of the form

new-line

has no effect.

6.10.8 Predefined macro names

- 1 The values of the predefined macros listed in the following subclauses¹⁷⁶⁾ (except for __FILE__ and __LINE__) remain constant throughout the translation unit.
- None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.
- The implementation shall not predefine the macro **__cplusplus**, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

- 1 The following macro names shall be defined by the implementation:
 - The date of translation of the preprocessing translation unit: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the asctime function, and the first character of dd is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.
 - **__FILE__** The presumed name of the current source file (a character string literal). 177)
 - __**LINE**__ The presumed line number (within the current source file) of the current source line (an integer constant). 177)
 - **__STDC**__ The integer constant **1**, intended to indicate a conforming implementation.
 - **__STDC_HOSTED**__ The integer constant **1** if the implementation is a hosted implementation or the integer constant **0** if it is not.

¹⁷⁶⁾ See "future language directions" (6.11.9).

¹⁷⁷⁾ The presumed source file name and line number can be changed by the **#line** directive.

STDC	VERSION	The integer constant	201 <i>ymm</i> L . ¹⁷⁸⁾

____TIME__ The time of translation of the preprocessing translation unit: a character string literal of the form "hh:mm:ss" as in the time generated by the asctime function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the **asctime** function (7.27.3.1).

6.10.8.2 Environment macros

- 1 The following macro names are conditionally defined by the implementation:
 - __STDC_ISO_10646__ An integer constant of the form yyyymmL (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type wchar_t, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
 - ___STDC_MB_MIGHT_NEQ_WC__ The integer constant 1, intended to indicate that, in the encoding for wchar_t, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.
 - __STDC_UTF_16__ The integer constant 1, intended to indicate that values of type char16_t are UTF-16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
 - __STDC_UTF_32__ The integer constant 1, intended to indicate that values of type char32_t are UTF-32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

Forward references: common definitions (7.19), unicode utilities (7.28).

¹⁷⁸⁾ This macro was not specified in ISO/IEC 9899:1990 and was specified as **199409L** in ISO/IEC 9899:1990/Amd.1:1995 and as **199901L** in ISO/IEC 9899:1999. The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this International Standard.

__STDC_IEC_559_COMPLEX__.

6.10.8.3 Conditional feature macros

1 The following macro names are conditionally defined by the implementation: __STDC_ANALYZABLE__ The integer constant 1, intended to indicate conformance to the specifications in annex L (Analyzability). **STDC IEC 559** The integer constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic). **STDC IEC 559 COMPLEX** The integer constant 1, intended to indicate adherence to the specifications in annex G (IEC 60559 compatible complex arithmetic). **STDC LIB EXT1** The integer constant **201**ymm**L**, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces). 179) __STDC_NO_ATOMICS__ The integer constant 1, intended to indicate that the implementation does not support atomic types (including the _Atomic type qualifier) and the **<stdatomic.h>** header. **STDC NO COMPLEX** The integer constant 1, intended to indicate that the implementation does not support complex types or the <complex.h> header. **STDC NO THREADS** The integer constant 1, intended to indicate that the implementation does not support the **<threads.h>** header. **STDC NO VLA** The integer constant 1, intended to indicate that the implementation does not support variable length arrays or variably modified types. 2 An implementation that defines __**STDC_NO_COMPLEX**__ shall not define

¹⁷⁹⁾ The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this International Standard.

6.10.9 Pragma operator

Semantics

1 A unary operator expression of the form:

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting any encoding prefix, 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 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.

2 EXAMPLE A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)
LISTING ( ..\listing.dir )
```