6.8 Preprocessing directives

Syntax

preprocessing-file:
 group$_{opt}$

group:
 group-part
 group group-part

group-part:
 pp-tokens$_{opt}$ *new-line*
 if-section
 control-line

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

if-group:
 # if *constant-expression new-line group*$_{opt}$
 # ifdef *identifier new-line group*$_{opt}$
 # ifndef *identifier new-line group*$_{opt}$

elif-groups:
 elif-group
 elif-groups elif-group

elif-group:
 # elif *constant-expression new-line group*$_{opt}$

else-group:
 # else *new-line group*$_{opt}$

endif-line:
 # endif *new-line*

control-line:
 # include *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list*$_{opt}$ **)** *replacement-list new-line*
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens*$_{opt}$ *new-line*
 # pragma *pp-tokens*$_{opt}$ *new-line*
 # *new-line*

lparen:
 the left-parenthesis character without preceding white-space

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

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 begins with a **#** preprocessing token that 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, and is ended by the next new-line character.[82]

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.

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

6.8.1 Conditional inclusion

Constraints

The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[83] and it may contain unary operator expressions of the form

    defined *identifier*

or

    defined (***identifier* **)

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

Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token.

Semantics

Preprocessing directives of the forms

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

82 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 6.8.3.2, for example).

83 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.

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 are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 6.4 using arithmetic that has at least the ranges specified in 5.2.4.2, except that **int** and **unsigned int** act as if they have the same representation as, respectively, **long** and **unsigned long**. 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.[84] Also, whether a single-character character constant may have a negative value is implementation-defined.

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.[85]

Forward references: macro replacement (6.8.3), source file inclusion (6.8.2).

6.8.2 Source file inclusion

Constraints

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

Semantics

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.

84 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)
```

85 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.

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 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.

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.[86] 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.

There shall be an implementation-defined mapping between the delimited sequence and the external source file name. The implementation shall provide unique mappings for sequences consisting of one or more letters (as defined in 5.2.1) followed by a period (.) and a single letter. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to six 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).

Examples

1. The most common uses of **#include** preprocessing directives are as in the following:

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

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 (6.8.3).

86 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.

6.8.3 Macro replacement

Constraints

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 a macro without use of lparen (an *object-like* macro) 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.

An identifier currently defined as a macro using lparen (a *function-like* macro) 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.

The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, and there shall exist a **)** preprocessing token that terminates the invocation.

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

Semantics

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.

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.

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 name[87] 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.

A preprocessing directive of the form

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

defines a function-like macro with arguments, 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.

87 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.

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 (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

6.8.3.1 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 translation unit; no other preprocessing tokens are available.

6.8.3.2 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

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 comprising 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). If the replacement that results is not a valid character string literal, the behavior is undefined. The order of evaluation of **#** and **##** operators is unspecified.

6.8.3.3 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, a parameter is immediately preceded or followed by a **##** preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

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. 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.

6.8.3.4 Rescanning and further replacement

After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned 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. Further, 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.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

6.8.3.5 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 translation unit.

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.

Examples

1. The simplest use of this facility is to define a ''manifest constant,'' as in

```
#define TABSIZE 100

int table[TABSIZE];
```

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.

3. To illustrate the rules for redefinition and reexamination, the sequence

```
#define x    3
#define f(a) f(x * (a))
#undef  x
#define x    2
#define g    f
#define z    z[0]
#define h    g(~
#define m(a) a(w)
#define w    0,1
#define t(a) a

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
      (f)^m(m);
```

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);
```

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

```
#define str(s)      # s
#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  /* from previous #include example */
#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"
```

Space around the **#** and **##** tokens in the macro definition is optional.

5. And finally, to demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FTN_LIKE(a)   ( a )
#define FTN_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 FTN_LIKE(b) ( a )   /* different parameter usage */
#define FTN_LIKE(b) ( b )   /* different parameter spelling */
```

6.8.4 Line control

Constraints

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.

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 32767.

A preprocessing directive of the form

> **`# line`** *digit-sequence* "*s-char-sequence*$_{opt}$" *new-line*

sets the 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*

(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.

6.8.5 Error directive

Semantics

A preprocessing directive of the form

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

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

6.8.6 Pragma directive

Semantics

A preprocessing directive of the form

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

causes the implementation to behave in an implementation-defined manner. Any pragma that is not recognized by the implementation is ignored.

6.8.7 Null directive

Semantics

A preprocessing directive of the form

*new-line*

has no effect.

6.8.8 Predefined macro names

The following macro names shall be defined by the implementation:

`__LINE__`	The line number of the current source line (a decimal constant).
`__FILE__`	The presumed name of the source file (a character string literal).
`__DATE__`	The date of translation of the source file (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.
`__TIME__`	The time of translation of the source file (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.
`__STDC__`	The decimal constant 1, intended to indicate a conforming implementation.

The values of the predefined macros (except for **`__LINE__`** and **`__FILE__`**) 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. All predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

Forward references: the **`asctime`** function (7.12.3.1).