The C Preprocessor

Last revised September 1998
for GCC version 2

Richard M. Stallman

This booklet is eventually intended to form the first chapter of a GNU C Language manual.

1 The C Preprocessor

The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, the C Compatible Compiler Preprocessor. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options '`-trigraphs`', '`-undef`' and '`-pedantic`', but in practice the consequences of having strict ANSI Standard C make it undesirable to do this. See Section 1.9 [Invocation], page 31.

The C preprocessor is designed for C-like languages; you may run into problems if you apply it to other kinds of languages, because it assumes that it is dealing with C. For example, the C preprocessor sometimes outputs extra white space to avoid inadvertent C token concatenation, and this may cause problems with other languages.

1.1 Transformations Made Globally

Most C preprocessor features are inactive unless you give specific directives to request their use. (Preprocessing directives are lines starting with '`#`'; see Section 1.2 [Directives], page 2). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of directives.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see Section 1.4.3 [Predefined], page 10).

The first two transformations are done *before* nearly all other parsing and before preprocessing directives are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into '`#define FOO 1020`'. You can split even an escape sequence with Backslash-Newline. For example, you can split `"foo\bar"` between the '`\`' and the '`b`' to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a '`#include`' directive in which the file name is delimited with '`<`' and '`>`'.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI "trigraph". Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to recognize the trigraph.

  This exception is relevant only if you use the '`-trigraphs`' option to enable trigraph processing. See Section 1.9 [Invocation], page 31.

1.2 Preprocessing Directives

Most preprocessor features are active only if you use preprocessing directives to request their use.

Preprocessing directives are lines in your program that start with '`#`'. The '`#`' is followed by an identifier that is the *directive name*. For example, '`#define`' is the directive that defines a macro. Whitespace is also allowed before and after the '`#`'.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directive names require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, '`#define`' must be followed by a macro name and the intended expansion of the macro. See Section 1.4.1 [Simple Macros], page 7.

A preprocessing directive cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the directive into multiple lines, but the comments are changed to Spaces before the directive is interpreted. The only way a significant Newline can occur in a preprocessing directive is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The '`#`' and the directive name cannot come from a macro expansion. For example, if '`foo`' is defined as a macro expanding to '`define`', that does not make '`#foo`' a valid preprocessing directive.

1.3 Header Files

A header file is a file containing C declarations and macro definitions (see Section 1.4 [Macros], page 7) to be shared between several source files. You request the use of a header file in your program with the C preprocessing directive '`#include`'.

1.3.1 Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with '`.h`'. Avoid unusual characters in header file names, as they reduce portability.

1.3.2 The '`#include`' Directive

Both user and system header files are included using the preprocessing directive '`#include`'. It has three variants:

`#include <`*file*`>`

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option '`-I`' (see Section 1.9 [Invocation], page 31). The option '`-nostdinc`' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of '#include' is slightly special because comments are not recognized within the '<...>'. Thus, in '#include <x/*y>' the '/*' does not start a comment and the directive specifies inclusion of a system header file named 'x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument *file* may not contain a '>' character. It may, however, contain a '<' character.

#include "*file*"

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the '-I-' option is used, the special treatment of the current directory is inhibited.)

The argument *file* may not contain '"' characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, '#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

#include *anything else*

This variant is called a *computed #include*. Any '#include' directive whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded (see Section 1.4 [Macros], page 7). When this is done, the result must fit one of the above two variants—in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-specific configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

1.3.3 How '#include' Works

The '#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' directive. For example, given a header file 'header.h' as follows,

```
char *test ();
```

and a main program called 'program.c' that uses the header file, like this,

```
int x;
#include "header.h"
```

```
main ()
{
  printf (test ());
}
```

the output generated by the C preprocessor for '`program.c`' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

It is possible for a header file to begin or end a syntactic unit such as a function definition, but that would be very confusing, so don't do it.

The line following the '`#include`' directive is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

1.3.4 Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* FILE_FOO_SEEN */
```

The macro `FILE_FOO_SEEN` indicates that the file has been included once already. In a user header file, the macro name should not begin with '`_`'. In a system header file, this name should begin with '`__`' to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

The GNU C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a '`#ifndef`' conditional, then it records that fact. If a subsequent '`#include`' specifies the same file, and the macro in the '#ifndef' is already defined, then the file is entirely skipped, without even reading it.

There is also an explicit directive to tell the preprocessor that it need not include a file more than once. This is called '#pragma once', and was used *in addition to* the '#ifndef' conditional around the contents of the header file. '#pragma once' is now obsolete and should not be used at all.

In the Objective C language, there is a variant of '#include' called '#import' which includes a file, but does so at most once. If you use '#import' *instead of* '#include', then you don't need the conditionals inside the header file to prevent multiple execution of the contents.

'#import' is obsolete because it is not a well designed feature. It requires the users of a header file—the applications programmers—to know that a certain header file should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using '#ifndef' accomplishes this goal.

1.3.5 Inheritance and Header Files

Inheritance is what happens when one object or file derives some of its contents by virtual copying from another object or file. In the case of C header files, inheritance means that one header file includes another header file and then replaces or adds something.

If the inheriting header file and the base header file have different names, then inheritance is straightforward: simply write '#include "*base*"' in the inheriting file.

Sometimes it is necessary to give the inheriting file the same name as the base file. This is less straightforward.

For example, suppose an application program uses the system header 'sys/signal.h', but the version of '/usr/include/sys/signal.h' on a particular system doesn't do what the application program expects. It might be convenient to define a "local" version, perhaps under the name '/usr/local/include/sys/signal.h', to override or add to the one supplied by the system.

You can do this by compiling with the option '-I.', and writing a file 'sys/signal.h' that does what the application program expects. But making this file include the standard 'sys/signal.h' is not so easy—writing '#include <sys/signal.h>' in that file doesn't work, because it includes your own version of the file, not the standard system version. Used in that file itself, this leads to an infinite recursion and a fatal error in compilation.

'#include </usr/include/sys/signal.h>' would find the proper file, but that is not clean, since it makes an assumption about where the system header file is found. This is bad for maintenance, since it means that any change in where the system's header files are kept requires a change somewhere else.

The clean way to solve this problem is to use '#include_next', which means, "Include the *next* file with this name." This directive works like '#include' except in searching for the specified file: it starts searching the list of header file directories *after* the directory in which the current file was found.

Suppose you specify '-I /usr/local/include', and the list of directories to search also includes '/usr/include'; and suppose both directories contain 'sys/signal.h'. Ordinary '#include <sys/signal.h>' finds the file under '/usr/local/include'. If that file contains

'`#include_next <sys/signal.h>`', it starts searching after that directory, and finds the file in '`/usr/include`'.

1.4 Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

1.4.1 Simple Macros

A *simple macro* is a kind of abbreviation. It is a name which stands for a fragment of code. Some people refer to these as *manifest constants*.

Before you can use a macro, you must *define* it explicitly with the '`#define`' directive. '`#define`' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named '`BUFFER_SIZE`' as an abbreviation for the text '`1020`'. If somewhere after this '`#define`' directive there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro '`BUFFER_SIZE`', resulting in

```
foo = (char *) xmalloc (1020);
```

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessing directives. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. This is because it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (But if it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name ‘`TABLESIZE`’ when used in the program would go through two stages of expansion, resulting ultimately in ‘`1020`’.

This is not at all the same as defining ‘`TABLESIZE`’ to be ‘`1020`’. The ‘`#define`’ for ‘`TABLESIZE`’ uses exactly the body you specify—in this case, ‘`BUFSIZE`’—and does not check to see whether it too is the name of a macro. It's only when you *use* ‘`TABLESIZE`’ that the result of its expansion is checked for more macro names. See Section 1.4.8.7 [Cascaded Macros], page 22.

1.4.2 Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept *arguments*. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition. A macro that accepts arguments is called a *function-like macro* because the syntax for using it looks like a function call.

To define a macro that uses arguments, you write a ‘`#define`’ directive with a list of *argument names* in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a “minimum” macro in GNU C. See Section 1.4.8.4 [Side Effects], page 19, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro ‘`min`’ include ‘`min (1, 2)`’ and ‘`min (x + 28, *p)`’.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro ‘`min`’ defined above, ‘`min (1, 2)`’ expands into

```
((1) < (2) ? (1) : (2))
```

where ‘`1`’ has been substituted for ‘`X`’ and ‘`2`’ for ‘`Y`’.

Likewise, ‘`min (x + 28, *p)`’ expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: '`array[x = y`' and '`x + 1]`'. If you want to supply '`array[x = y, x + 1]`' as an argument, you must write it as '`array[(x = y, x + 1)]`', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, '`min (min (a, b), c)`' expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If a macro `foo` takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: '`foo ( )`'. Just '`foo ()`' is providing no arguments, which is an error if `foo` expects an argument. But '`foo0 ()`' is the correct way to call a macro defined to take zero arguments, like this:

```
#define foo0() ...
```

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named '`min`' in the same source file that defines the macro. If you write '`&min`' with no argument list, you refer to the function. If you write '`min (x, bb)`', with an argument list, the macro is expanded. If you write '`(min) (a, bb)`', where the name '`min`' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function '`min`'.

You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines '`FOO`' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines '`BAR`' to take no argument and always expand into '`(x) - 1 / (x)`').

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

1.4.3 Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

1.4.3.1 Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding `__GNUC__` in this table are standardized by ANSI C; the rest are GNU C extensions.

`__FILE__` This macro expands to the name of the current input file, in the form of a C string constant. The precise name returned is the one that was specified in '`#include`' or as the input file name argument.

`__LINE__` This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

This and '`__FILE__`' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A '`#include`' directive changes the expansions of '`__FILE__`' and '`__LINE__`' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the '`#include`' directive, the expansions of '`__FILE__`' and '`__LINE__`' revert to the values they had before the '`#include`' (but '`__LINE__`' is then incremented by one as processing moves to the line after the '`#include`').

The expansions of both '`__FILE__`' and '`__LINE__`' are altered if a '`#line`' directive is used. See Section 1.6 [Combining Sources], page 29.

`__DATE__` This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like '`"Feb 1 1996"`'.

`__TIME__` This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like '`"23:59:01"`'.

`__STDC__` This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

On some hosts, system include files use a different convention, where '`__STDC__`' is normally 0, but is 1 if the user specifies strict conformance to the C Standard. The preprocessor follows the host convention when processing system include files, but when processing user files it follows the usual GNU C convention.

This macro is not defined if the '`-traditional`' option is used.

`__STDC_VERSION__`

This macro expands to the C Standard's version number, a long integer constant of the form '*yyyymm*L' where *yyyy* and *mm* are the year and month of the Standard version. This signifies which version of the C Standard the preprocessor conforms to. Like '`__STDC__`', whether this version number is accurate for the entire implementation depends on what C compiler will operate on the output from the preprocessor.

This macro is not defined if the '`-traditional`' option is used.

`__GNUC__` This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, '`__GNUC__`' is undefined. The value identifies the major version number of GNU CC ('`1`' for GNU CC version 1, which is now obsolete, and '`2`' for version 2).

`__GNUC_MINOR__`

The macro contains the minor version number of the compiler. This can be used to work around differences between different releases of the compiler (for example, if gcc 2.6.3 is known to support a feature, you can test for `__GNUC__ > 2 || (__GNUC__ == 2 && __GNUC_MINOR__ >= 6)`). The last number, '`3`' in the example above, denotes the bugfix level of the compiler; no macro contains this value.

`__GNUG__` The GNU C compiler defines this when the compilation language is C++; use '`__GNUG__`' to distinguish between GNU C and GNU C++.

`__cplusplus`

The draft ANSI standard for C++ used to require predefining this variable. Though it is no longer required, GNU C++ continues to define it, as do other popular C++ compilers. You can use '`__cplusplus`' to test whether a header is compiled by a C compiler or a C++ compiler.

`__STRICT_ANSI__`

GNU C defines this macro if and only if the '`-ansi`' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

`__BASE_FILE__`

This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

`__INCLUDE_LEVEL__`
: This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every '`#include`' directive and decremented at every end of file. For input files specified by command line arguments, the nesting level is zero.

`__VERSION__`
: This macro expands to a string constant which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as '`"2.6.0"`'.

`__OPTIMIZE__`
: GNU CC defines this macro in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. You should not refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

`__CHAR_UNSIGNED__`
: GNU C defines this macro if and only if the data type `char` is unsigned on the target machine. It exists to cause the standard header file '`limits.h`' to work correctly. You should not refer to this macro yourself; instead, refer to the standard macros defined in '`limits.h`'. The preprocessor uses this macro to determine whether or not to sign-extend large character constants written in octal; see Section 1.5.2.1 [The '`#if`' Directive], page 24.

`__REGISTER_PREFIX__`
: This macro expands to a string (not a string constant) describing the prefix applied to CPU registers in assembler code. You can use it to write assembler code that is usable in multiple environments. For example, in the '`m68k-aout`' environment it expands to the null string, but in the '`m68k-coff`' environment it expands to the string '`%`'.

`__USER_LABEL_PREFIX__`
: Similar to `__REGISTER_PREFIX__`, but describes the prefix applied to user generated labels in assembler code. For example, in the '`m68k-aout`' environment it expands to the string '`_`', but in the '`m68k-coff`' environment it expands to the null string. This does not work with the '`-mno-underscores`' option that the i386 OSF/rose and m88k targets provide nor with the '`-mcall*`' options of the rs6000 System V Release 4 target.

1.4.3.2 Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones. You can use '`cpp -dM`' to see the values of predefined macros; see Section 1.9 [Invocation], page 31.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example, `unix` 'unix' is normally predefined on all Unix systems.

`BSD` 'BSD' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax` 'vax' is predefined on Vax computers.

`mc68000` 'mc68000' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k` 'm68k' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use 'mc68000' and some use 'm68k'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020` 'M68020' has been observed to be predefined on some systems that use 68020 CPUs—in addition to 'mc68000' and 'm68k', which are less specific.

`_AM29K`
`_AM29000` Both '_AM29K' and '_AM29000' are predefined for the AMD 29000 CPU family.

`ns32000` 'ns32000' is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun` 'sun' is predefined on all models of Sun computers.

`pyr` 'pyr' is predefined on all models of Pyramid computers.

`sequent` 'sequent' is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option '-ansi' inhibits the definition of these symbols.

This tends to make '-ansi' useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with '-ansi'. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding '__' at the beginning and end. Thus, the symbol `__vax__` would be available on a Vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when `cpp` is itself compiled) by the macro 'CPP_PREDEFINES', which should be a string containing '-D' options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in 'tm.h'.

1.4.4 Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying 'foo (z)' results in '"foo (z)"'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character '#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no '#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Here the actual argument for 'EXP' is substituted once as given, into the 'if' statement, and once as stringified, into the argument to 'fprintf'. The 'do' and 'while (0)' are a kludge to make it possible to write 'WARN_IF (*arg*);', which the resemblance of 'WARN_IF' to a function would make C programmers want to do; see Section 1.4.8.3 [Swallow Semicolon], page 18.

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies the actual value of 'EXP' into a separate string constant, resulting in text like

```
do { if (x == 0) \
        fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) \
        fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying 'p = "foo\n";' results in '"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: '\n' by itself stringifies to '"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

1.4.5 Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator '##' in the macro body. When the macro is called, after actual arguments are substituted, all '##' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the '##'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with '_command'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as '1.5' and 'e3') into a number. Also, multi-character operators such as '+=' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with 'x' on one side and '+' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the '`x`' and '`+`' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating '`/`' and '`*`': the '`/*`' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a '`##`' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

1.4.6 Undefining Macros

To *undefine* a macro means to cancel its definition. This is done with the '`#undef`' directive. '`#undef`' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, '`FOO`' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of '`#undef`' directive will cancel definitions with arguments or definitions that don't expect arguments. The '`#undef`' directive has no effect when used on a name not currently defined as a macro.

1.4.7 Redefining Macros

Redefining a macro means defining (with '`#define`') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see Section 1.3 [Header Files], page 3), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with '`#undef`' before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

1.4.8 Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

1.4.8.1 Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand '`call_with_1 (double)`' into '`(2*(1))`'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
```

This bizarre example expands to '`fprintf (stderr, "%s %d", p, 35)`'!

1.4.8.2 Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many '`int`' objects are needed to hold a certain number of '`char`' objects.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

Unintended grouping can result in another way. Consider '`sizeof ceil_div(1, 2)`'. That has the appearance of a C expression that would compute the size of the type of '`ceil_div (1, 2)`', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the '`sizeof`' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the recommended way to define '`ceil_div`':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

1.4.8.3 Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument '`p`' says where to find it) across whitespace characters:

```
#define SKIP_SPACES(p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
    if (*p++ != ' ') {          \
      p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be laid out if not part of a macro definition.

A call to this macro might be '`SKIP_SPACES (p, lim)`'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in '`SKIP_SPACES (p, lim);`'

But this can cause trouble before '`else`' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements—the compound statement and a null statement—in between the '`if`' condition and the '`else`' makes invalid C code.

The definition of the macro 'SKIP_SPACES' can be altered to solve this problem, using a 'do ... while' statement. Here is how:

```
#define SKIP_SPACES(p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
while (0)
```

Now 'SKIP_SPACES (p, lim);' expands into

```
do {...} while (0);
```

which is one statement.

1.4.8.4 Duplication of Side Effects

Many C programs define a macro 'min', for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where 'x + y' has been substituted for 'X' and 'foo (z)' for 'Y'.

The function 'foo' is used only once in the statement as it appears in the program, but the expression 'foo (z)' has been substituted twice into the macro expansion. As a result, 'foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that 'min' is an *unsafe* macro.

The best solution to this problem is to define 'min' in a way that computes the value of 'foo (z)' only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                     \
({ typeof (X) __x = (X), __y = (Y);   \
   (__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when *using* the macro 'min'. For example, you can calculate the value of 'foo (z)', save it in a variable, and use that variable in 'min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where we assume that 'foo' returns type 'int').

1.4.8.5 Self-Referential Macros

A *self-referential* macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where '`foo`' is also a variable in your program.

Following the ordinary rules, each reference to '`foo`' will expand into '`(4 + foo)`'; then this will be rescanned and will expand into '`(4 + (4 + foo))`'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at '`(4 + foo)`'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of '`foo`' wherever '`foo`' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that '`foo`' is a variable will not expect that it is a macro as well. The reader will come across the identifier '`foo`' in the program and think its value should be that of the variable '`foo`', whereas in fact the value is four greater.

The special rule for self-reference applies also to *indirect* self-reference. This is the case where a macro *x* expands to use a macro '`y`', and the expansion of '`y`' refers to the macro '`x`'. The resulting reference to '`x`' comes indirectly from the expansion of '`x`', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

'`x`' would expand into '`(4 + (2 * x))`'. Clear?

But suppose '`y`' is used elsewhere, not from the definition of '`x`'. Then the use of '`x`' in the expansion of '`y`' is not a self-reference because '`x`' is not "in progress". So it does expand. However, the expansion of '`x`' contains a reference to '`y`', and that is an indirect self-reference now because '`y`' is "in progress". The result is that '`y`' expands to '`(2 * (4 + y))`'.

It is not clear that this behavior would ever be useful, but it is specified by the ANSI C standard, so you may need to understand it.

1.4.8.6 Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see Section 1.4.8.5 [Self-Reference], page 20): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to '"foo"'. Once more, prescan has been prevented from having any noticeable effect.

More precisely, stringification and concatenation use the argument as written, in unprescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.

```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

expands to '"foo" lose(4)'.

You might now ask, "Why mention the prescan, if it makes no difference? And why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.
- Macros whose expansions contain unshielded commas.

We say that *nested* calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if 'f' is a macro that expects one argument, 'f (f (1))' is a nested pair of calls to 'f'. The desired expansion is made by expanding 'f (1)' and substituting that into the definition of 'f'. The prescan causes the expected result to happen. Without the prescan, 'f (1)' itself would be substituted as an actual argument, and the inner use of 'f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. Here is an example:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))

bar(foo)
```

We would like 'bar(foo)' to turn into '(1 + (foo))', which would then turn into '(1 + (a,b))'. But instead, 'bar(foo)' expands into 'lose(a,b)', and you get an error because `lose` requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the '`({...})`' construct which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it—if you use two levels of macros. Let's add a new macro '`xstr`' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into '`"4"`', not '`"foo"`'. The reason for the difference is that the argument of '`xstr`' is expanded at prescan (because '`xstr`' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for '`str`'. '`str`' uses its argument without prescan because it performs stringification; but it cannot prevent or undo the prescanning already done by '`xstr`'.

1.4.8.7 Cascaded Use of Macros

A *cascade* of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining '`TABLESIZE`' to be '`1020`'. The '`#define`' for '`TABLESIZE`' uses exactly the body you specify—in this case, '`BUFSIZE`'—and does not check to see whether it too is the name of a macro.

It's only when you *use* '`TABLESIZE`' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of '`BUFSIZE`' at some point in the source file. '`TABLESIZE`', defined as shown, will always expand using the definition of '`BUFSIZE`' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now '`TABLESIZE`' expands (in two stages) to '`37`'. (The '`#undef`' is to prevent any warning about the nontrivial redefinition of `BUFSIZE`.)

1.4.9 Newlines in Macro Arguments

Traditional macro processing carries forward all newlines in macro arguments into the expansion of the macro. This means that, if some of the arguments are substituted more than once, or not at all, or out of order, newlines can be duplicated, lost, or moved around within the expansion. If the expansion consists of multiple statements, then the effect is to distort the line numbers of some of these statements. The result can be incorrect line numbers, in error messages or displayed in a debugger.

The GNU C preprocessor operating in ANSI C mode adjusts appropriately for multiple use of an argument—the first use expands all the newlines, and subsequent uses of the same argument produce no newlines. But even in this mode, it can produce incorrect line numbering if arguments are used out of order, or not used at all.

Here is an example illustrating this problem:

```
#define ignore_second_arg(a,b,c) a; c

ignore_second_arg (foo (),
                   ignored (),
                   syntax error);
```

The syntax error triggered by the tokens '`syntax error`' results in an error message citing line four, even though the statement text comes from line five.

1.5 Conditionals

In a macro processor, a *conditional* is a directive that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an '`if`' statement in C, but it is important to understand the difference between them. The condition in an '`if`' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

1.5.1 Why Conditionals are Used

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessing conditional, the offending code can be effectively excised from the program when it is not valid.

- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data, or prints the values of those data for debugging, while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessing conditionals.

1.5.2 Syntax of Conditionals

A conditional in the C preprocessor begins with a *conditional directive*: '`#if`', '`#ifdef`' or '`#ifndef`'. See Section 1.5.4 [Conditionals-Macros], page 26, for information on '`#ifdef`' and '`#ifndef`'; only '`#if`' is explained here.

1.5.2.1 The '`#if`' Directive

The '`#if`' directive in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the '`#endif`' is not required, but it is a good practice because it helps people match the '`#endif`' to the corresponding '`#if`'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the '`#endif`' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as `long` or `unsigned long`.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type '`char`' for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats '`char`' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations ('`&&`' and '`||`').
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that '`sizeof`' operators and `enum`-type values are not allowed. `enum`-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The *controlled text* inside of a conditional can include preprocessing directives. Then the directives inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the '`#if`' and '`#endif`' directives must balance.

1.5.2.2 The '`#else`' Directive

The '`#else`' directive can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If *expression* is nonzero, and thus the *text-if-true* is active, then '`#else`' acts like a failing conditional and the *text-if-false* is ignored. Contrariwise, if the '`#if`' conditional fails, the *text-if-false* is considered included.

1.5.2.3 The '`#elif`' Directive

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional directive, '`#elif`', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

'`#elif`' stands for "else if". Like '`#else`', it goes in the middle of a '`#if`'-'`#endif`' pair and subdivides it; it does not require a matching '`#endif`' of its own. Like '`#if`', the '`#elif`' directive includes an expression to be tested.

The text following the '`#elif`' is processed only if the original '`#if`'-condition failed and the '`#elif`' condition succeeds. More than one '`#elif`' can go in the same '`#if`'-'`#endif`' group. Then the text after each '`#elif`' is processed only if the '`#elif`' condition succeeds after the original '`#if`' and any previous '`#elif`' directives within it have failed. '`#else`' is equivalent to '`#elif 1`', and '`#else`' is allowed after any number of '`#elif`' directives, but '`#elif`' may not follow '`#else`'.

1.5.3 Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put '`#if 0`' before it and '`#endif`' after it. This is better than using comment delimiters '`/*`' and '`*/`' since those won't work if the code already contains comments (C comments do not nest).

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced '`#if`' and '`#endif`').

Conversely, do not use '`#if 0`' for comments which are not C code. Use the comment delimiters '`/*`' and '`*/`' instead. The interior of '`#if 0`' must consist of complete tokens; in particular, singlequote characters must balance. But comments often contain unbalanced singlequote characters (known in English as apostrophes). These confuse '`#if 0`'. They do not confuse '`/*`'.

1.5.4 Conditionals and Macros

Conditionals are useful in connection with macros or assertions, because those are the only ways that an expression's value can vary from one compilation to another. A '`#if`' directive whose expression uses no macros or assertions is equivalent to '`#if 1`' or '`#if 0`'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program.

For example, here is a conditional that tests the expression '`BUFSIZE == 1020`', where '`BUFSIZE`' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

(Programmers often wish they could test the size of a variable or data type in '`#if`', but this does not work. The preprocessor does not understand `sizeof`, or typedef names, or even the type keywords such as `int`.)

The special operator '`defined`' is used in '`#if`' expressions to test whether a certain name is defined as a macro. Either '`defined` *name*' or '`defined (`*name*`)`' is an expression whose value is 1 if *name* is defined as macro at the current point in the program, and 0 otherwise. For the '`defined`' operator it makes no difference what the definition of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would succeed if either of the names '`vax`' and '`ns16000`' is defined as a macro. You can test the same condition using assertions (see Section 1.5.5 [Assertions], page 27), like this:

```
#if #cpu (vax) || #cpu (ns16000)
```

If a macro is defined and later undefined with '`#undef`', subsequent use of the '`defined`' operator returns 0, because the name is no longer defined. If the macro is defined again with another '`#define`', '`defined`' will recommence returning 1.

Conditionals that test whether just one name is defined are very common, so there are two special short conditional directives for this case.

`#ifdef` *name*
: is equivalent to '`#if defined (`*name*`)`'.

`#ifndef` *name*
: is equivalent to '`#if ! defined (`*name*`)`'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name '`vax`' is a predefined macro. On other machines, it would not be defined.
- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro '`BUFSIZE`' might be defined in a configuration file for your program that is included as a header file in each source file. You would use '`BUFSIZE`' in a preprocessing conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with '`-D`' and '`-U`' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See Section 1.9 [Invocation], page 31.

1.5.5 Assertions

Assertions are a more systematic alternative to macros in writing conditionals to test what sort of computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with preprocessing directives or command-line options.

The macros traditionally used to describe the type of target are not classified in any way according to which question they answer; they may indicate a hardware architecture, a particular hardware model, an operating system, a particular version of an operating system, or specific configuration options. These are jumbled together in a single namespace. In contrast, each assertion consists of a named question and an answer. The question is usually called the *predicate*. An assertion looks like this:

```
#predicate (answer)
```

You must use a properly formed identifier for *predicate*. The value of *answer* can be any sequence of words; all characters are significant except for leading and trailing whitespace, and differences in internal whitespace sequences are ignored. Thus, '`x + y`' is different from '`x+y`' but equivalent to '`x + y`'. '`)`' is not allowed in an answer.

Here is a conditional to test whether the answer *answer* is asserted for the predicate *predicate*:

```
#if #predicate (answer)
```

There may be more than one answer asserted for a given predicate. If you omit the answer, you can test whether *any* answer is asserted for *predicate*:

```
#if #predicate
```

Most of the time, the assertions you test will be predefined assertions. GNU C provides three predefined predicates: `system`, `cpu`, and `machine`. `system` is for assertions about the type of software, `cpu` describes the type of computer architecture, and `machine` gives more information about the computer. For example, on a GNU system, the following assertions would be true:

```
#system (gnu)
#system (mach)
#system (mach 3)
#system (mach 3.subversion)
#system (hurd)
#system (hurd version)
```

and perhaps others. The alternatives with more or less version information let you ask more or less detailed questions about the type of system software.

On a Unix system, you would find `#system (unix)` and perhaps one of: `#system (aix)`, `#system (bsd)`, `#system (hpux)`, `#system (lynx)`, `#system (mach)`, `#system (posix)`, `#system (svr3)`, `#system (svr4)`, or `#system (xpg4)` with possible version numbers following.

Other values for `system` are `#system (mvs)` and `#system (vms)`.

Portability note: Many Unix C compilers provide only one answer for the `system` assertion: `#system (unix)`, if they support assertions at all. This is less than useful.

An assertion with a multi-word answer is completely different from several assertions with individual single-word answers. For example, the presence of `system (mach 3.0)` does not mean that `system (3.0)` is true. It also does not directly imply `system (mach)`, but in GNU C, that last will normally be asserted as well.

The current list of possible assertion values for `cpu` is: `#cpu (a29k)`, `#cpu (alpha)`, `#cpu (arm)`, `#cpu (clipper)`, `#cpu (convex)`, `#cpu (elxsi)`, `#cpu (tron)`, `#cpu (h8300)`, `#cpu (i370)`, `#cpu (i386)`, `#cpu (i860)`, `#cpu (i960)`, `#cpu (m68k)`, `#cpu (m88k)`, `#cpu (mips)`, `#cpu (ns32k)`, `#cpu (hppa)`, `#cpu (pyr)`, `#cpu (ibm032)`, `#cpu (rs6000)`, `#cpu (sh)`, `#cpu (sparc)`, `#cpu (spur)`, `#cpu (tahoe)`, `#cpu (vax)`, `#cpu (we32000)`.

You can create assertions within a C program using '`#assert`', like this:

```
#assert predicate (answer)
```

(Note the absence of a '`#`' before *predicate*.)

Each time you do this, you assert a new true answer for *predicate*. Asserting one answer does not invalidate previously asserted answers; they all remain true. The only way to remove an assertion is with '`#unassert`'. '`#unassert`' has the same syntax as '`#assert`'. You can also remove all assertions about *predicate* like this:

```
#unassert predicate
```

You can also add or cancel assertions using command options when you run `gcc` or `cpp`. See Section 1.9 [Invocation], page 31.

1.5.6 The '#error' and '#warning' Directives

The directive '`#error`' causes the preprocessor to report a fatal error. The rest of the line that follows '`#error`' is used as the error message. The line must consist of complete tokens.

You would use '#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef __vax__
#error "Won't work on Vaxen.  See comments at get_last_object."
#endif
```

See Section 1.4.3.2 [Nonstandard Predefined], page 12, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with '#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
    || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE should not be divisible by a small prime
#endif
```

The directive '#warning' is like the directive '#error', but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows '#warning' is used as the warning message.

You might use '#warning' in obsolete header files, with a message directing the user to the header file which should be used instead.

1.6 Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use '#include'; both '#include' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a directive by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the `bison` parser generator, which operates on another file that is the true source file. Parts of the output from `bison` are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the `bison` output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic debuggers to know the original source file and line number of each line in the `bison` input.

`bison` arranges this by writing '#line' directives into the output file. '#line' is a directive that specifies the original line number and source file name for subsequent input in the current preprocessor input file. '#line' has three variants:

`#line` *linenum*

Here *linenum* is a decimal integer constant. This specifies that the line number of the following line of input, in its original source file, was *linenum*.

`#line` *linenum filename*

Here *linenum* is a decimal integer constant and *filename* is a string constant. This specifies that the following line of input came originally from source file *filename* and its line number there was *linenum*. Keep in mind that *filename* is not just a file name; it is surrounded by doublequote characters so that it looks like a string constant.

`#line` *anything else*

anything else is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

'`#line`' directives alter the results of the '`__FILE__`' and '`__LINE__`' predefined macros from that point on. See Section 1.4.3.1 [Standard Predefined], page 10.

The output of the preprocessor (which is the input for the rest of the compiler) contains directives that look much like '`#line`' directives. They start with just '`#`' instead of '`#line`', but this is followed by a line number and file name as in '`#line`'. See Section 1.8 [Output], page 30.

1.7 Miscellaneous Preprocessing Directives

This section describes three additional preprocessing directives. They are not very useful, but are mentioned for completeness.

The *null directive* consists of a '`#`' followed by a Newline, with only whitespace (including comments) in between. A null directive is understood as a preprocessing directive but has no effect on the preprocessor output. The primary significance of the existence of the null directive is that an input line consisting of just a '`#`' will produce no output, rather than a line of output containing just a '`#`'. Supposedly some old C programs contain such lines.

The ANSI standard specifies that the effect of the '`#pragma`' directive is implementation-defined. In the GNU C preprocessor, '`#pragma`' directives are not used, except for '`#pragma once`' (see Section 1.3.4 [Once-Only], page 5). However, they are left in the preprocessor output, so they are available to the compilation pass.

The '`#ident`' directive is supported for compatibility with certain other systems. It is followed by a line of text. On some systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this directive has no effect. Typically '`#ident`' is only used in header files supplied with those systems where it is meaningful.

1.8 C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessing directive lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, unless '`-traditional`' is used, spaces may be inserted into the expansions of macro calls to prevent tokens from being concatenated.

Source file name and line number information is conveyed by lines of the form

`#` *linenum filename flags*

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file *filename* at line *linenum*.

After the file name comes zero or more flags, which are '`1`', '`2`', '`3`', or '`4`'. If there are multiple flags, spaces separate them. Here is what the flags mean:

'`1`' This indicates the start of a new file.

'`2`' This indicates returning to a file (after having included another file).

'`3`' This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

'`4`' This indicates that the following text should be treated as C.

1.9 Invoking the C Preprocessor

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful on its own.

The C preprocessor expects two file names as arguments, *infile* and *outfile*. The preprocessor reads *infile* together with any other files it specifies with '`#include`'. All the output generated by the combined input files is written in *outfile*.

Either *infile* or *outfile* may be '`-`', which as *infile* means to read from standard input and as *outfile* means to write to standard output. Also, if *outfile* or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. These options can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

'`-P`' Inhibit generation of '`#`'-lines with line-number information in the output from the preprocessor (see Section 1.8 [Output], page 30). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the '`#`'-lines.

'`-C`' Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

'`-traditional`'

Try to imitate the behavior of old-fashioned C, as opposed to ANSI C.

- Traditional macro expansion pays no attention to singlequote or doublequote characters; macro argument symbols are replaced by the argument values even when they appear within apparent string or character constants.
- Traditionally, it is permissible for a macro expansion to end in the middle of a string or character constant. The constant continues into the text surrounding the macro call.
- However, traditionally the end of the line terminates a string or character constant, with no error.

- In traditional C, a comment is equivalent to no text at all. (In ANSI C, a comment counts as whitespace.)
- Traditional C does not have the concept of a "preprocessing number". It considers '1.0e+4' to be three tokens: '1.0e', '+', and '4'.
- A macro is not suppressed within its own definition, in traditional C. Thus, any macro that is used recursively inevitably causes an error.
- The character '#' has no special meaning within a macro definition in traditional C.
- In traditional C, the text at the end of a macro expansion can run together with the text after the macro call, to produce a single token. (This is impossible in ANSI C.)
- Traditionally, '\' inside a macro argument suppresses the syntactic significance of the following character.

Use the '-traditional' option when preprocessing Fortran code, so that singlequotes and doublequotes within Fortran comment lines (which are generally not recognized as such by the preprocessor) do not cause diagnostics about unterminated character or string constants.

However, this option does not prevent diagnostics about unterminated comments when a C-style comment appears to start, but not end, within Fortran-style commentary.

So, the following Fortran comment lines are accepted with '-traditional':

```
C This isn't an unterminated character constant
C Neither is "20000000000, an octal constant
C in some dialects of Fortran
```

However, this type of comment line will likely produce a diagnostic, or at least unexpected output from the preprocessor, due to the unterminated comment:

```
C Some Fortran compilers accept /* as starting
C an inline comment.
```

Note that g77 automatically supplies the '-traditional' option when it invokes the preprocessor. However, a future version of g77 might use a different, more-Fortran-aware preprocessor in place of cpp.

'-trigraphs'
: Process ANSI standard trigraph sequences. These are three-character sequences, all starting with '??', that are defined by ANSI C to stand for single characters. For example, '??/' stands for '\', so ''??/n'' is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless '-trigraphs' is used, but if you ever notice the difference it will be with relief.

  You don't want to know any more about trigraphs.

'-pedantic'
: Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows '#else' or '#endif'.

'-pedantic-errors'
: Like '-pedantic', except that errors are produced rather than warnings.

'-Wtrigraphs'
: Warn if any trigraphs are encountered. Currently this only works if you have turned trigraphs on with '-trigraphs' or '-ansi'; in the future this restriction will be removed.

'-Wcomment'
: Warn whenever a comment-start sequence '/*' appears in a '/*' comment, or whenever a Backslash-Newline appears in a '//' comment.

'-Wall'
: Requests both '-Wtrigraphs' and '-Wcomment' (but not '-Wtraditional' or '-Wundef').

'-Wtraditional'
: Warn about certain constructs that behave differently in traditional and ANSI C.

'-Wundef'
: Warn if an undefined identifier is evaluated in an '#if' directive.

'-I *directory*'
: Add the directory *directory* to the head of the list of directories to be searched for header files (see Section 1.3.2 [Include Syntax], page 3). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

'-I-'
: Any directories specified with '-I' options before the '-I-' option are searched only for the case of '#include "*file*"'; they are not searched for '#include <*file*>'.

  If additional directories are specified with '-I' options after the '-I-', these directories are searched for all '#include' directives.

  In addition, the '-I-' option inhibits the use of the current directory as the first search directory for '#include "*file*"'. Therefore, the current directory is searched only if it is requested explicitly with '-I.'. Specifying both '-I-' and '-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

'-nostdinc'
: Do not search the standard system directories for header files. Only the directories you have specified with '-I' options (and the current directory, if appropriate) are searched.

'-nostdinc++'
: Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building the C++ library.)

'-remap'
: When searching for a header file in a directory, remap file names if a file named 'header.gcc' exists in that directory. This can be used to work around limitations of file systems with file name restrictions. The 'header.gcc' file should

contain a series of lines with two tokens on each line: the first token is the name to map, and the second token is the actual name to use.

'-D *name*' Predefine *name* as a macro, with definition '1'.

'-D *name*=*definition*'
Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one '-D' for the same *name*, the rightmost definition takes effect.

'-U *name*' Do not predefine *name*. If both '-U' and '-D' are specified for one name, the '-U' beats the '-D' and the name is not predefined.

'-undef' Do not predefine any nonstandard macros.

'-gcc' Define the macros *__GNUC__* and *__GNUC_MINOR__*. These are defined automatically when you use 'gcc -E'; you can turn them off in that case with '-no-gcc'.

'-A *predicate*(*answer*)'
Make an assertion with the predicate *predicate* and answer *answer*. See Section 1.5.5 [Assertions], page 27.

You can use '-A-' to disable all predefined assertions; it also undefines all predefined macros and all macros that preceded it on the command line.

'-dM' Instead of outputting the result of preprocessing, output a list of '#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor; assuming you have no file 'foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

will show the values of any predefined macros.

'-dD' Like '-dM' except in two respects: it does *not* include the predefined macros, and it outputs *both* the '#define' directives and the result of preprocessing. Both kinds of output go to the standard output file.

'-dI' Output '#include' directives in addition to the result of preprocessing.

'-M [-MG]' Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using '\'-newline.

'-MG' says to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to '-M'.

This feature is used in automatic updating of makefiles.

'-MM [-MG]'
: Like '-M' but mention only the files included with '#include "*file*"'. System header files included with '#include <*file*>' are omitted.

'-MD *file*'
: Like '-M' but the dependency information is written to *file*. This is in addition to compiling the file as specified—'-MD' does not inhibit ordinary compilation the way '-M' does.

  When invoking gcc, do not specify the *file* argument. gcc will create file names made by replacing ".c" with ".d" at the end of the input file names.

  In Mach, you can use the utility md to merge multiple dependency files into a single dependency file suitable for using with the 'make' command.

'-MMD *file*'
: Like '-MD' except mention only user header files, not system header files.

'-H'
: Print the name of each header file used, in addition to other normal activities.

'-imacros *file*'
: Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '-imacros *file*' is to make the macros defined in *file* available for use in the main input.

'-include *file*'
: Process *file* as input, and include all the resulting output, before processing the regular input file.

'-idirafter *dir*'
: Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that '-I' adds to).

'-iprefix *prefix*'
: Specify *prefix* as the prefix for subsequent '-iwithprefix' options.

'-iwithprefix *dir*'
: Add a directory to the second include path. The directory's name is made by concatenating *prefix* and *dir*, where *prefix* was specified previously with '-iprefix'.

'-isystem *dir*'
: Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

'-x c'
'-x c++'
'-x objective-c'
'-x assembler-with-cpp'
: Specify the source language: C, C++, Objective-C, or assembly. This has nothing to do with standards conformance or extensions; it merely selects which base syntax to expect. If you give none of these options, cpp will deduce the language from the extension of the source file: '.c', '.cc', '.m', or '.S'. Some

other common extensions for C++ and assembly are also recognized. If cpp does not recognize the extension, it will treat the file as C; this is the most generic mode.

Note: Previous versions of cpp accepted a '`-lang`' option which selected both the language and the standards conformance level. This option has been removed, because it conflicts with the '`-l`' option.

'`-std=`*standard*'
'`-ansi`' Specify the standard to which the code should conform. Currently cpp only knows about the standards for C; other language standards will be added in the future.

standard may be one of:

`iso9899:1990`
The ISO C standard from 1990.

`iso9899:199409`
`c89` The 1990 C standard, as amended in 1994. '`c89`' is the customary shorthand for this version of the standard.

The '`-ansi`' option is equivalent to '`-std=c89`'.

`iso9899:199x`
`c9x` The revised ISO C standard, which is expected to be promulgated some time in 1999. It has not been approved yet, hence the '`x`'.

`gnu89` The 1990 C standard plus GNU extensions. This is the default.

`gnu9x` The 199x C standard plus GNU extensions.

'`-Wp,-lint`'
Look for commands to the program checker `lint` embedded in comments, and emit them preceded by '`#pragma lint`'. For example, the comment '`/* NOTREACHED */`' becomes '`#pragma lint NOTREACHED`'.

Because of the clash with '`-l`', you must use the awkward syntax above. In a future release, this option will be replaced by '`-flint`' or '`-Wlint`'; we are not sure which yet.

'`-$`' Forbid the use of '`$`' in identifiers. The C standard does not permit this, but it is a common extension.

Concept Index

T

U

Index of Directives, Macros and Options

#

-

_

B

D

M

N

P

S

U

V

Table of Contents