Contents

Table of Contents

2

The GNU C Preprocessor

The GNU C preprocessor (**cpp**) is a macro processor the C compiler uses to transform your program before actual compilation. It's called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

The C preprocessor provides the following four facilities:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define and use macros, which are abbreviations for arbitrary fragments of C code. The C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessor commands, 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 their implementation details. This section describes the GNU C preprocessor, which provides a superset of the features of ANSI-standard C.

Note: Mac OS X actually includes two preprocessors: the standard GNU C preprocessor (**cpp**) and the precompilation preprocessor (**cpp-precomp**). The two preprocessors are largely similar, except for some rarely used extensions. The precompilation preprocessor (**cpp-precomp**) is the default preprocessor for C and Objective-C code. The standard preprocessor (**cpp**) is the default preprocessor for C++ and Objective-C++ code. To switch to the standard preprocessor (**cpp**) on platforms on which precompiled headers are available, use the **-traditonal-cpp** flag on **cpp** the command line.

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 C preprocessor is configured to accept these constructs by default. To get ANSI-standard C you would use the options **-trigraphs**, **-undef**, and **-pedantic**, although in practice the consequences of having strict ANSI Standard C may make it undesirable to do this. See the section "Invoking the C Preprocessor" for more information.

Global Transformations

Most C preprocessor features are inactive unless you give specific commands to request their use. But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands:

- C, C++, and Objective C comments are replaced with single spaces.
- Backslash-newline sequences are deleted. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see the section "Predefined Macros").

The first two transformations are done *before* nearly all other parsing and before preprocessor commands 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 to **#define FOO 1020**. You can even split an escape sequence with backslash-newline. For example, you can split "foo\bar" between the backslash 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 doesn't allow newlines in string constants, so this isn't considered a problem.)

There are a few exceptions to all three transformations:

- C comments and predefined macro names aren't recognized inside an **#include** command 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 rather than an exception.)

- 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.

Preprocessor Commands

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

Preprocessor commands are lines in your program that start with **#**. The **#** is followed by an identifier that's the command name. For example, **#define** is the command that defines a macro. White-space characters are allowed before and after the **#**.

The set of valid command names is fixed. Programs can't define new preprocessor commands.

Some command names require arguments; these make up the rest of the command line and must be separated from the command name by one or more white-space characters. For example, **#define** must be followed by a macro name and the intended expansion of the macro.

A preprocessor command normally can't be more than one line. It may be split cosmetically with backslash-newline, but that has no effect on its meaning. Comments containing newlines can also divide the command into multiple lines, but the comments are changed to spaces before the command is interpreted. The only way a significant newline can occur in a preprocessor command 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. This compiler does accept them, however..

The **#** and the command name can't come from a macro expansion. For example, if **foo** is defined as a macro expanding to **define**, that doesn't make **#foo** a valid preprocessor command.

Header Files

Header files can contain C declarations and macro definitions that are to be shared by more than one source file. You request the inclusion of a header file in a source file by using the C preprocessor command **#include** (or more typically in the Mac OS X environment, the Objective-C preprocessor command **#import**).

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 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's 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 recompiled.

By convention, names of header files end with the extension ".h".

The #include Command

Both user and system header files are included using the preprocessor command **#include**. The **#include** command directs the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor will contain 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** command. Included files can themselves contain **#include** commands to include other files.

Included files are not limited to declarations and macro definitions, although those are 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.

The line following the **#include** command is always treated as a separate line by the C preprocessor, even if the included file lacks a final newline.

Note: The Objective-C language equivalent of **#include** is **#import**; the only difference is that **#import** doesn't include a file more than once, no matter how many **#import** commands try to include it. You should feel free to use **#import** in your code, but be aware that it isn't defined as part of ANSI-standard C.

The **#include** command has three variants:

#include <>

```
#include <file>
```

This variant is used for system or framework header files. It searches for a file named *file* in a list of directories specified by you, and then, if it isn't found, in a standard list of system directories. You specify directories to search for header files with the command option **-I** (see the section "Invoking the C Preprocessor"). The option **-nostdinc** inhibits searching the standard system directories; in this case only the directories you specify are searched.

For frameworks the semantics of the text between the angle brackets is different. The word preceding the slash indicates a framework. Thus the line:

```
<AppKit/AppKit.h>
```

causes the search for the header file **AppKit.h** to occur in the Application Kit framework (**/System/Library/Frameworks/AppKit.framework**). The **PrivateHeaders** subdirectory is searched first, and then the **Headers** directory, thus allowing a private header file to override a public one. The flags **-F**, **-I**, and **-L** affect search path for frameworks (see "Invoking the Preprocessor," below); The linker's **-framework** flag, however, has no effect.

The parsing of this form of **#include** is slightly special because comments are not recognized within the **<*file*>** argument. (The **<** and **>** are similar to string delimiters instead of operators.) Thus, in **#include <x/*y>** the **/*** doesn't start a comment and the command specifies inclusion of a system header file named **x/*y**. (Of course, a header file with such a name is unlikely to exist on Mac OS X or a UNIX system, where shell wildcard features would make it hard to manipulate.)

The *file* argument may not contain a **>** character, although it may contain a **<** character. Whitespace characters in the *file* argument may or may not be ignored, so do not use them.

#include ""

```
   #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 tried first because it's presumed to be the location of the files of the program being compiled. (If the **-I-** option is used, the special treatment of the current directory is inhibited.)

The *file* argument may not contain **"** characters. If backslashes occur within *file*, they might be 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 file name containing three backslashes. It isn't clear why this behavior is ever useful, but the ANSI standard specifies it.

#include *anything else*

```
   #include anything else
```

This variant is called a computed **#include**. Any **#include** command whose argument doesn't fit the above two forms is a computed **#include**. The text *anything else* is checked for macro calls, which are expanded. When this is done, the result must fit one of the above two variants.

This feature allows you to define a macro that controls the file name to be used at a later point in the program. One application of this is to allow a site-configuration file for your program to specify the names of the system header 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.

Multiple Inclusion of Header Files

Very often one header file includes another, which can result in a certain header file being included more than once. This may lead to errors if the header file defines structure types or typedefs, and in any event is wasteful. For these reasons, you should try to avoid multiple inclusion of a header file.

The standard way to prevent multiple inclusion of a file 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; its name begins with __ to avoid conflicts with user programs, and it contains the name of the file and some additional text to avoid conflicts with other header files.

Alternatively (if compatibility with non–Mac OS X platforms isn't an issue), you can ensure that each file is included only once simply by using the Objective-C **#import** command instead of the **#include** command.

Precompiled Header Files

A precompiled header is a C header file that has been preprocessed and parsed, thereby improving compile time and reducing symbol table size. The macros and external declarations from the original header are sorted to enable fast lookup. A new implementation of the C preprocessor can use precompiled headers in place of standard headers.

In most cases, the use of precompiled headers is transparent. Precompiled headers are simple enough to use that most projects require no conversion at all, or can be converted in a day or less.

Note that the following information on precompiled headers applies only to to this compiler. This feature may not be avaiable, or may take another form, in other compilers.

Using Precompiled System Header Files

The precompiled version of a header file has a ".p" extension, rather than the standard ".h" extension. You should *not* refer to **AppKit.p** in your source files; just use **AppKit.h** and the preprocessor will use the precompiled form if it's available and appropriate.

When the preprocessor encounters an include directive, it automatically looks for a precompiled version of the header. If one is found, it checks whether the context is equivalent to the context in which the precompiled header was built—if it is, the precompiled header is used. However, if any of the following problems occur, the non-precompiled form is included instead:

- A header which was included by the precompiled header could not be found in the filesystem to verify its modification time, or the modification time did not match. In practice, this never occurs for precompiled headers that are part of the release, and occurs only rarely when programmers build their own precompiled headers.
- A macro was defined when the precompiled header was built, but is not defined in the current context. This is only a problem if the macro was actually referenced somewhere in the precompiled header.
- A macro was undefined when the precompiled header was built, but is defined in the current context. This is only a problem if there might have been an invocation of the macro in the precompiled header.

Compile-time warnings (described at the end of this file) indicate the nature of any problems that occur. However, you may suppress these warning messages with **-Wno-precomp**. The intent of these messages is to point out problems that, if corrected, would improve compilation speed.

If you're developing a small project, you don't need to bother building your own precompiled headers—just use the precompiled system headers **AppKit.p**, **Foundation.p**, **mach.p** and so on. If these system precompiled header files don't exist on Mac OS X, you can create them by running the **fixPrecomps** utility. Also, it's easy to create your own precompiled headers if you wish to do so, however, as described in the next section.

Creating Your Own Precompiled Header Files

You create a precompiled header by passing the **-precomp** switch to **cc**. Depending on the context(s) in which the header is used, **-D** switches should also be passed to **cc**, as explained below.

```
% cc -precomp foo.h -o foo.p
```

We say a header is "context dependent" if the definitions in the header may change depending on the context in which it is included. Most uses of conditional compilation and macro expansions cause context dependence. For instance, the following header is context dependent:

```
#ifdef DEBUG
int a;
#else
int b;
#endif
```

The context at any point is determined by the macros that are defined there. A precompiled header must be created in a context equivalent to that where it is used. By passing switches to the preprocessor, any set of macros can be predefined, creating a context in which the precompiled header is built. This is done by passing a **-D** switch for each macro in the context.

A precompiled header built from system headers typically requires no **-D** switches, because programmers usually include system headers in a context-independent way. For instance, the public Application Kit headers contain almost no preprocessor conditionals; clients cannot change declarations in headers by defining macros. So the command to build a precompiled header from **AppKit.h** is:

```
% cc -precomp AppKit.h -o AppKit.p -arch i386 -arch ppc
```

(The architectures affected are usually specified using the **-arch** switch: “fat” compilation requires “fat” precompilation.) But if you must use a header **bar.h** in a context where FOO is defined, you should build the precompiled header as follows:

```
% cc -precomp -DFOO bar.h -o bar.p
```

You should also pass any preprocessor switches, such as **-I**, that you use in your project.

By making precompiled headers bigger (that is, containing more headers), a given C file may include fewer precompiled headers, and will generally compile faster. However, the bigger a precompiled header is, the more likely that name conflicts will occur.

For example, if you were to combine all the headers for a project, including system headers, into a single precompiled header, it is possible that there would be a name conflict. There may be a macro defined that happens to match one of your local identifiers, or there may be a public struct declared that happens to match one of your private struct names. Such conflicts manifest themselves as preprocessing errors, syntax errors, or semantic errors. The conflicts may be resolved by renaming identifiers, or removing a conflicting header from the precompiled header.

Another disadvantage to big precompiled headers is file dependencies. If all of the C files in a project depend on a single precompiled header which in turn depends on all headers in the project, then changing a header requires recompilation of the entire project. A better approach is to build a precompiled header containing all the system headers used by a project, and perhaps also a separate precompiled header for the local headers in the project. We recommend that during development, while local headers are changing, precompiled headers be used only for system files. When local headers have stabilized, they may be combined into a precompiled header.

A precompiled header is dependent on all the files it includes. A make dependency rule can be constructed similar to the way rules are constructed for source files. The following rule builds a precompiled header from a header:

```
.h.p:
    cc -precomp $(CFLAGS) $*.h -o $*.p
```

A precompiled header records absolute path names for all the headers that went into it. These paths are then checked when the precompiled header is used. Therefore a precompiled header should be built in the same directory in which it is to be used, and all the headers that went into the precompiled header must not be moved or modified.

Troubleshooting

To use precompiled headers you must have the **cpp-precomp** preprocessor and parser, which has several incompatibilities with the standard GNU C preprocessor and parser. For example, preprocessing errors and syntax errors are in a slightly different format.

Only rarely will you have trouble building a precompiled header. The most common problem you might encounter is that the header doesn't parse; this is often because the header does not include other headers it depends on, so that there are undefined types. Another typical problem is conflicting definitions, which can be solved by renaming identifiers or removing a header from the precompiled header.

The following list describes the compile-time warnings that may occur when using a precompiled header:

- **could not use precompiled header 'header.p'**

  The precompiled header could not be used for one of the reasons below.

- **macro 'macro' undefined**

  The macro was defined when the precompiled header was built, but is not defined in the current context.

- **macro 'macro' previously defined on command line for precomp. Not defined.**

  The macro was undefined when the precompiled header was built, but is defined in the current context. This error can often be avoided by importing precompiled headers in the source file before any other headers.

- **macro 'macro' defined by 'header.p' conflicts with precomp**

A previously included precompiled header defines a macro differently than does the current precompiled header being processed.

- **macro 'macro' defined on command line conflicts with precomp**

  Similar to the previous warning, except that the earlier definition of the macro occurred on the command line (with the **-D** flag).

- **macro 'macro' redefined, locations of the conflict are:**
  header1.h:23
  header2.h:47 (within the precompiled header)

  The macro has been defined in two different ways in two different precompiled headers

- **#ifdef 'SYM' not defined when precompiled**

  A symbol was defined for the inclusion of this precompiled header, but was not when the header was precompiled. Since this symbol is used in an #ifdef, the precompiled header does not contain all the source code desired by the including context.

- **'header.h' has different date than in precomp**

  The modification time of the header on the disk does not match the modification time of the header when the precompiled header was built.

- **could not find 'header.h'**

  The header which was included by the precompiled header could not be found on the disk to verify its modification time.

- **could not use precomp 'header.p' (incorrect version)**

  It was discovered that the version of the referenced precompiled header is incompatible with the compiler, possibly signifying a corrupt or obsolete **header.p.**

Macros

A macro is an abbreviation you define once and then use later. This section describes some important features associated with macros in the C preprocessor.

Simple Macros

A simple macro is a kind of abbreviation—it's a name that stands for a fragment of code. Simple macros are sometimes referred to as *manifest constants*.

Before you can use a macro, you must define it explicitly with the **#define** command. **#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**. With this definition in effect, the C preprocessor would expand the following statement

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

to

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

The definition must be a single line; however, it may not end in the middle of a multiline string constant or character constant.

For readability, uppercase is used for macro names by convention. Programs are easier to read when it's possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line (although you can always split a long macro definition cosmetically with backslash-newline). There's one exception: Newlines can be included in the macro definition if they're within a string or character constant. It isn't 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, and the body need not resemble valid C code. (Of course, you might get error messages from the C compiler when you use the macro.) However, tokens must be valid C tokens. For example, the symbol **1A** would cause an error in both the compiler and the preprocessor.

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 macros continues. Therefore, the macro body can contain other macros. For example, after the following definitions

```
#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 isn't the same as defining **TABLESIZE** to be **1020**. The **#define** for **TABLESIZE** uses exactly the body you specify—in this case, **BUFSIZE**—and doesn't 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 the section "Cascaded Use of Macros."

Macros that Take Arguments

A simple macro always stands for exactly the same text, each time it's 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.

To define a macro that takes arguments, you use the **#define** command with a list of parameters in parentheses after the name of the macro. The parameters may be any valid C identifiers separated by commas at the top level (that is, commas that aren't within parentheses) and, optionally, by white-space characters. The left parenthesis must follow the macro name immediately, with no space in between.

For example, here's a macro that computes the minimum of two numeric values:

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

Note that this isn't the best way to define a "minimum" macro in GNU C (see the section "Duplication of Side Effects" for more information).

To use a macro that takes arguments, you write the name of the macro followed by a list of arguments in parentheses, separated by commas. The number of arguments you give must match the number of parameters in the macro definition. The following examples show the use of the macro **min**:

```
min (1, 2)
min (x + 28, *p)
```

The expansion text of the macro depends on the arguments you use. Each of the macro's parameters is replaced, throughout the macro definition, with the corresponding argument. Using the same macro **min** defined above, **min (1, 2)** expands to

```
((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 arguments must balance; a comma within parentheses doesn't end an argument. However, there's no requirement for brackets or braces to balance; thus, if you want to supply

```
array[x = y, x + 1]
```

as an argument, you would write it as

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

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

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

Line breaks shown here for clarity wouldn't actually be generated.

If a macro takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses. For example

```
foo ( )
```

is acceptable, but

```
foo ()
```

generates an error if **foo** expects an argument.

The correct way to call a macro defined to take zero arguments is:

```
#define foo()
  . . .
foo()
```

If you use the macro name followed by something other than a left parenthesis (after ignoring any spaces, tabs, and comments that follow), it isn't considered a macro invocation, and the preprocessor doesn't change what you've written. Therefore, it's 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 argument list follows) or the variable or function (if an argument list doesn't 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. 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** isn't followed by a left parenthesis, the macro isn't expanded; rather, the function **min** is called.

A name can't be defined 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 the rest of the name is taken to be the expansion. The reason for this is that it's 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 (which defines **FOO** to take an argument and expand into minus the reciprocal of that argument)

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

or this (which defines **FOO** to take no argument and always expand into **(x) - 1 / (x)**):

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

It matters only in the macro definition whether there's a space before the left parenthesis; when you use the macro, it doesn't matter if there are spaces there or not.

Predefined Macros

Several standard macros are predefined, some by ANSI C and some as extensions. Their names all start and end with double underscores.

ANSI C standard macros

The following predefined macros are part of the ANSI C standard:

__FILE__

```
__FILE__
```

This macro expands to the name of the current input file, in the form of a C string constant.

__LINE__

```
__LINE__
```

This macro expands to the current input line number, in the form of a decimal integer constant. (Note that although this is considered a predefined macro, 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__);
```

An **#include** command 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** command, 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** command is used. See the section "Combining Source Files."

__DATE__

```
__DATE__
```

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains 15 characters and looks like “Tue Jun 02 1992”.

__TIME__

```
    __TIME__
```

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains 12 characters and looks like “23:59:01 EDT”.

__STDC__

```
    __STDC__
```

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

GNU C Macros

The following predefined macros are GNU C extensions to the ANSI C standard:

__GNUC__

```
    __GNUC__
```

This macro is defined if and only if this is GNU C. Moreover, it’s defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, **__GNUC__** is undefined.

__STRICT_ANSI__

```
    __STRICT_ANSI__
```

This macro is defined 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 traditional UNIX constructs that are incompatible with ANSI C.

__GNUG__

```
    __GNUG__
```

The GNU C compiler defines this when the compilation language is C++; use **__GNUG__** or **__cplusplus** to distinguish between GNU C and GNU C++ code.

__cplusplus

```
    __cplusplus
```

The draft standard for C++ requires the predefinition of this variable. GNU C++ to define it as **1**, to indicate that this compiler does not yet fully support the standard C++ language. You can use **__cplusplus** to test whether a header is compiled by a C compiler or a C++ compiler.

__VERSION__

```
    __VERSION__
```

This macro expands to a string describing the version number of the compiler. The string is normally a sequence of decimal numbers separated by periods, such as **"1.18"**. The main use of this macro is to incorporate the version number into a string constant.

__OPTIMIZE__

```
    __OPTIMIZE__
```

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It's unwise to refer to or test the definition of this macro unless you make sure that programs will execute with the same effect regardless.

__CHAR_UNSIGNED__

```
    __CHAR_UNSIGNED__
```

This macro is defined if and only if the data type **char** is unsigned on the target machine. Its purpose is to cause the standard header file **limit.h** to work correctly. It's bad practice to refer to this macro yourself; instead, refer to the standard macros defined in **limit.h**.

Mac OS X macros

The following macros are defined in Mac OS X:

__OBJC__

```
    __OBJC__
```

This macro is defined when you compile Objective-C ".m" files or Objective-C++ ".M" files, or when you override the file extension with **-ObjC** or **-ObjC++**.

__ASSEMBLER__

```
    __ASSEMBLER__
```

This macro is defined when compiling ".s" files.

__NATURAL_ALIGNMENT__

```
    __NATURAL_ALIGNMENT__
```

This macro is defined on systems that use natural alignment. When using natural alignment, an **int** is aligned on **sizeof(int)** boundary, a **short int** is aligned on **sizeof(short)** boundary, and so on. It's defined by default when you're compiling code for the PowerPC, SPARC, and HPPA. It's not defined when you use the **-malign-mac68k** compiler switch.

The following code snippet guarantees that the struct layout for **foo** will not change regardless of whether or not **__NATURAL_ALIGNMENT__** is defined.

```
struct foo {
  short count;
#ifdef __NATURAL_ALIGNMENT__
  /* compiler will pad to align "flags"
     on the sizeof(long) boundary */
#else
  unsigned char pad[ sizeof(long) - sizeof(short) ];
#endif
  long  flags;
}
```

__STRICT_BSD__

```
__STRICT_BSD__
```

This macro is defined if and only if the **-bsd** switch was specified when GNU C was invoked.

__MACH__

```
__MACH__
```

This macro is defined if Mach system calls are supported.

Platform-Dependant 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 section lists some that are useful on Mac OS X computers.

__APPLE__

```
__APPLE__
```

This macro is defined on any Apple platform, including Mac OS X.

Use the **__APPLE__** macro instead of the following, which may not be supported in future versions: **NeXT**, **__NeXT**, **__NeXT__**, and **_NEXT_SOURCE**.

__APPLE_CC__

```
__APPLE_CC__
```

This macro is set to an integer that represents the version number of the compiler. This lets you distinguish, for example, between compilers based on the same version of GCC, but with different bug fixes or features. Larger values denote later compilers.

Use the **__APPLE_CC__** macro instead of the following, which may not be supported in future versions: **NX_COMPILER_RELEASE_3_0**, **NX_COMPILER_RELEASE_3_1**, **NX_COMPILER_RELEASE_3_2**, **NX_COMPILER_RELEASE_3_3**, **NX_CURRENT_COMPILER_RELEASE**, **NS_TARGET**, **NS_TARGET_MAJOR**, and **NS_TARGET_MINOR**.

__hppa__

```
__hppa__
```

This macro is defined when you're compiling code to run on a HPPA-based workstastion .

__i386__

```
__i386__
```

This macro is defined when you're compiling code to run on any chip in the Intel 80x86 family, including all versions of the Pentium.

Use the **__i386__** macro instead of **i386** and **__i386**, which may not be supported in future versions.

__ppc__

```
__ppc__
```

This macro is defined when you're compiling code to run on any chip in PowerPC family.

Use the **__ppc__** macro instead of **ppc** and **__ppc**, which may not be supported in future versions.

__sparc__

```
__sparc__
```

This macro is defined when you're compiling code to run on a SPARC-based workstastion .

__unix__

```
__unix__
```

This macro is defined when you're compiling code to run on a UNIX system. It is predefined if you are compiling code to run on a Solaris and HP-UX system.

Note that it will not be defined in future versions of Mac OS X. To test for a Mac OS X system, use **__APPLE__**.

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 argument when it's 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's 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 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)** make it possible to write **WARN_IF (ARG)**; safely (see the section "Swallowing the Semicolon").

The stringification feature is limited to transforming one macro argument into one string constant: There's 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:

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

Stringification in C involves more than putting double quotes around the fragment; it's necessary to put backslashes in front of all double quotes, 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 aren't inside string or character constants aren't duplicated: **\n** by itself stringifies to **"\n"**.

White-space characters (including comments) in the text being stringified are handled according to the following rules:

- All leading and trailing white-space characters are ignored.
- Any sequence of white-space characters in the middle of the text is converted to a single space in the stringified result.

It's often useful to define, for example:

```
STR(X) #X
```

so that when you use the macro instead of **#X** directly, **X** is re-scanned one more time for macro expansion.

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 argument to the macro can be concatenated with another 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 invoked, arguments are substituted. Then all **##** operators are deleted, along with any white-space characters next to them (including white-space characters that are part of an 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 that 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":

```
#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's also possible to concatenate two numbers (or a number and a name, such as **1.5** and **e3**) into a number. Also, multicharacter operators such as **+=** can be formed by concatenation. In some cases it's 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 isn't meaningful because those two characters can't fit together in any lexical unit of C. Although the ANSI standard says that such an attempt at concatenation is undefined, the GNU C preprocessor handles it as follows: it puts the **x** and **+** side by side with no particular special results.

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. You can freely use comments next to a **##** in a macro definition, or in arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

Undefining Macros

To undefine a macro means to cancel its definition. This is done with the **#undef** command. **#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's 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** must 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** command will cancel definitions with arguments or definitions that don't expect arguments. The **#undef** command has no effect when used on a name not currently defined as a macro.

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 the section "Header Files"), so they're 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's 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 previously. Remember, comments count as whitespace.

Pitfalls and Subtleties of Macros

This section describes some special rules that apply to macros and macro expansion, and points out certain cases in which the rules have counterintuitive consequences that you must watch out for.

Improperly Nested Constructs

Recall that when a macro is invoked 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 macros.

It's possible to piece together a macro invocation coming partially from the macro body and partially from the 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 don't have to have balanced parentheses. By writing an unbalanced left parenthesis in a macro body, it's possible to create a macro invocation that begins inside the macro body but ends outside 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)
```

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 has parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. This section discusses why it's best to write macros that way.

Suppose you define a macro

```
#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's used as follows:

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

This expands into

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

which doesn't do what's intended. The operator-precedence rules of C make this equivalent to:

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

But what we want is:

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

Defining the macro as follows provides the desired result:

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

However, unintended grouping can happen in another way. Consider **sizeof ceil_div(1, 2)**. This 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's what it expands to:

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

This would take the size of an integer and divide it by 2. 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's the recommended way to define **ceil_div**:

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

Swallowing the Semicolon

Often it's desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, which advances a pointer across space 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 appear if not part of a macro definition.

An invocation of this macro might be **SKIP_SPACES (p, lim)**. Strictly speaking, the invocation 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:

```
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:

```
#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 one statement:

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

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. Therefore **min** is an “unsafe” macro.

One way to solve 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 don't 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);
}
```

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 isn't considered a macro invocation. It's passed into the preprocessor output unchanged.

Consider the following example (assume that **foo** is also a variable in your program):

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

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's a bad idea to take advantage of this feature. A person reading the program who sees that **foo** is a variable won't expect that it's 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 4 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 **y**'s expansion refers to the macro **x**. The resulting reference to **x** comes indirectly from the expansion of **x**, so it's a self-reference and isn't further expanded. Thus, after

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

x would expand into **(4 + (2 * x))**.

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

Separate Expansion of Macro Arguments

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

What really happens is more subtle: First each argument text is scanned separately for macros. 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 arguments are scanned twice to expand macros in them.

Most of the time, this has no effect. If the argument contained any macros, they're expanded during the first scan. The result therefore contains no macros, so the second scan doesn't change it. If the argument were substituted as given, with no prescan, the single remaining scan would find the same macros and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an argument of another macro (see the section "Self-Referential Macros" above); the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this isn't what happens. The self-references that don't expand in the first scan are marked so that they won't expand in the second scan either.

The prescan isn't done when an argument is stringified or concatenated. (More precisely, stringification and concatenation use the argument as written, in unprescanned form. The same argument would be used in prescanned form if it's substituted elsewhere without stringification or concatenation.) Thus,

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

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

The prescan does make a difference in three special cases:

- Nested invocations of a macro
- Macros that invoke other macros that stringify or concatenate
- Macros whose expansions contain unshielded commas

Nested invocations of a macro occur when a macro's argument contains an invocation of that very macro. For example, if **f** is a macro that expects one argument, **f (f (1))** is a nested pair of invocations of **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 argument, and the inner use of **f** would appear during the main scan as an indirect self-reference and wouldn't be expanded. Here, the prescan cancels an undesirable side effect of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. For 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 aren't 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's also one case where prescan is useful. It's 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 to **"4"**, not **"foo"**. The reason for the difference is that the argument of **xstr** is expanded at prescan (because **xstr** doesn't specify stringification or concatenation of the argument). The result of prescan then forms the argument for **str**. **str** uses its argument without prescan because it performs stringification; but it can't prevent or undo the prescanning already done by **xstr**.

Cascaded Use of Macros

A cascade of macros occurs when one macro's body contains a reference to another macro (a very common practice). For example:

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

This isn't 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 doesn't 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's currently in effect:

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

Now **TABLESIZE** expands in two stages to **37**.

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 are out of order, newlines can be duplicated, lost, or moved around within the expansion. If the expansion consists of multiple statements, then the the line numbers of some of these statements can become distorted. The result can be incorrect line numbers in error messages or as displayed by a debugger.

The C preprocessor operating in ANSI C mode adjusts itself for multiple uses 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 are 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.

Inability to Define a Macro that Produces a # Character

You can't use the C preprocessor to define macros that produce **#** characters. For instance, the following has unexpected results:

```
#define linkmacro(numBytes) link #numBytes,a6
```

Note that you can use the **#** character inside a string or character constant, as shown here:

```
#define PrintSharp() printf("#")
```

Macro Arguments inside String Constants

The C preprocessor doesn't substitute macro arguments that appear inside string constants. For example, the following macro will produce the output **"a"** no matter what the argument **a** is:

```
#define foo(a) "a"
```

The **-traditional** option directs **cc** to handle such cases (among others) in the traditional non-ANSI way.

Conditionals

In a macro processor, a conditional is a command that allows 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 an **if** statement in C, but it's 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's operating on. The condition in a preprocessor conditional command 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.

There are three reasons to use a conditional:

- A program may need to use different code depending on the target machine or operating system. 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 don't exist on the other system. When this happens, it isn't 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 preprocessor conditional, the offending code can be effectively excised from the program when it isn't 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 while the other doesn't.

- A conditional whose condition is always false is a good way to exclude code from the program but keep it for future reference.

Most programs using only Mac OS X API won't need to use preprocessor conditionals.

Syntax of Conditionals

A conditional in the C preprocessor begins with a conditional command: **#if**, **#ifdef**, or **#ifndef**. These and a few related commands are described in the following sections.

The #if Command

The **#if** command in its simplest form consists of

```
#if expression
conditional-text
#endif   /* expression */
```

The comment following the **#endif** isn't required, but it makes the code easier to read. Such comments should always be used, except in short conditionals that aren't nested. (Although you can put anything at all after the **#endif** and it will be ignored by the C preprocessor, only comments are acceptable in ANSI Standard C.)

expression is a C expression of type **int**, 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 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.
- Character constants. The C preprocessor uses the C data type **char** for these character constants.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, **&&**, and **||**.
- Identifiers that aren't macros, which are all treated as 0.
- Macro invocation. All macros in the expression are expanded before actual computation of the expression's value begins.

sizeof operators and **enum**-type values aren't allowed. **enum**-type values, like all other identifiers that aren't taken as macro invocations and expanded, are treated as 0.

The controlled text inside a conditional can include preprocessor commands. Then the commands 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** commands must balance.

The #else Command

The **#else** command can be added to a conditional to provide alternative text to be used if the condition is false:

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

If *expression* is nonzero, *text-if-true* is included; then **#else** acts like a failing conditional and *text-if-false* is ignored. If *expression* is 0, the **#if** conditional fails and *text-if-false* is included.

The #elif Command

A common use of nested conditionals is to check for more than two possible alternatives:

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

The conditional command **#elif** (which stands for "else if") can be used to abbreviate this as follows:

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

Like **#else**, **#elif** goes in the middle of a **#if-#endif** pair and subdivides it; it doesn't require a matching **#endif** of its own. Like **#if**, the **#elif** command 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** commands within it have failed. **#else** is allowed after any number of **#elif** commands, but **#elif** may not follow a **#else**.

Keeping Deleted Code for Future Reference

If you replace or delete part of the program but want to keep the old code around as a comment for future reference, you can simply put **#if 0** before it and **#endif** after it.

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

Conditionals and Macros

Conditionals are useful in macros or assertions, because those are the only ways that an expression's value can vary from one compilation to another. An **#if** command 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's 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 expressions, 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's a definition. Thus, for example,

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

will include the following code if either **vax** or **ns16000** is defined as a macro.

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 again return 1.

Conditionals that test just the definedness of one name are very common, so there are two special short conditional commands 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 any of the following reasons:

- Some macros are predefined on each kind of machine. For example, on a Mac OS X computer the name **__APPLE__** is a nonstandard predefined macro. On other machines, it isn't defined.

- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It's useful to test these macros with conditionals to avoid using a system feature on a machine where it isn't implemented.

- Macros are a common way for you 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's included as a header file in each source file. You would use **BUFSIZE** in a preprocessor 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. You can also use macros to specify different build types of the same program. For example, you could use **-DDEBUG** and **-DPROFILE** for debugging and profililing builds, respectively. See the section "Invoking the C Preprocessor."

The #error and #warning Commands

The **#error** command causes the preprocessor to report a fatal error. The rest of the line that follows **#error** is used as the error message.

You would use **#error** inside a conditional that detects a combination of parameters that you know the program doesn't support.

For example, if you know that the program won't run properly on a VAX, you might write

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

Similarly, 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 shouldn't be divisible by a small prime
#endif
```

The **#warning** command is like the **#error** command, 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.

Pragmas

The **#pragma** command is specified in the ANSI standard to have an arbitrary implementation-defined effect. For example, a **#pragma** might be used to indicate to the translator the best way to generate code, optimize, or diagnose errors. It may also pass information to the translator about the environment, or add debugging information.

The effect of anything specified in a **#pragma** is currently limited to the outermost declaration (that is, a function or a global data declaration).

The following pragmas are passed on by the C preprocessor to the compiler itself:

Pragma	Description
#pragma CC_OPT_ON	Force optimization on.
#pragma CC_OPT_OFF	Force optimization off.
#pragma CC_OPT_RESTORE	Restore optimization to what was specified on the command line (on if **-O** was specified, off if not).
#pragma CC_WRITABLE_STRINGS	Place strings in the data segment.
#pragma CC_NON_WRITABLE_STRINGS	Place strings in the text segment.

All other **#pragma** commands are ignored by the C preprocessor.

Combining Source Files

One of the jobs of the C preprocessor is to tell the C compiler the source file and line number that each line of C code came from.

C code can come from multiple source files if you use **#include** or **#import**. If you include header files, or if you use conditionals or macros, the line number of a line in the preprocessor output may be different from the line number of the same line in the original source file. Normally you would want both the C compiler (in error messages) and the GDB debugger to use the line numbers of your source file.

The C preprocessor offers a **#line** command by which you can control this feature explicitly. **#line** specifies the original line number and source file name for subsequent input in the current preprocessor input file. **#line** has three variants:

Command	Description
#line *linenum*	*linenum* is a decimal integer constant. This resets the current line number in the source file to *linenum*.
#line *linenum* **"***file***"**	*linenum* is a decimal integer constant and **"**file**"** is a string constant. This resets the line number to *linenum* and changes the name of the file referred to by *file*.
#line *macros*	*macros* should be one or more macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

#line commands alter the results of the **__FILE__** and **__LINE__** predefined macros from that point on. See the section "Predefined Macros."

The output of the preprocessor (which is the input for the rest of the compiler) contains commands that look much like **#line** commands. They start with just **#** instead of **#line**, but this is followed by a line number and file name as in **#line**.

C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessor command lines have been replaced with blank lines and all comments with spaces. White-space characters within a line aren't altered;

however, a space is inserted after the expansions of most macros. Also, pragmas are passed through verbatim.

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

```
# linenum file {digit}
```

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 *file* at line *linenum*.

After the file name comes zero or more numeric flags: 1, 2 or 3, separated by spaces if multiple flags:

Flag	Description
1.	The start of a new file
2.	Return to a file (after having included another file).
3.	Text that follows comes from a system header file (so certain warnings should be suppressed).

Invoking the C Preprocessor

Usually you won't have to invoke the C preprocessor explicitly, because the C compiler does so automatically. However, there may be times when you want to use the preprocessor by itself by invoking the **cpp** command.

The **cpp** and **cpp-precomp** commands expect two file names as arguments, *infile* and *outfile*. The preprocessor reads *infile* together with any other files that *infile* specifies by means of **#include** or **#import**. 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 the standard input and as *outfile* means to write to the 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's a list of command options accepted by the C preprocessor. Most of them can also be given when compiling a C program; they're passed along automatically to the preprocessor when it's invoked by the compiler.

-P

```
    -P
```

Inhibit generation of **#** lines with line-number information in the output from the preprocessor (see the section “C Preprocessor Output”). This might be useful when running the preprocessor on something that isn’t C code and that will be sent to a program which might be confused by the **#** lines.

-C

```
    -C
```

Don’t discard comments: Pass them through to the output file. Comments appearing in arguments of a macro invocation will be copied to the output before the expansion of the macro.

-traditional

```
    -traditional
```

Try to imitate the behavior of old-fashioned C, as opposed to ANSI C. Traditional C preprocessing has these characteristics:

- Traditional macro expansion pays no attention to single-quote or double-quote characters; macro argument symbols are replaced by the argument values even when they appear within apparent string or character constants.
- Traditionally, a macro expansion may end in the middle of a string or character constant. The constant continues into the text surrounding the macro call.
- 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.
- In traditional C a macro is not suppressed within its own definition. 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.

- Traditionally, \ inside a macro argument suppresses the syntactic significance of the following character.

-trigraphs

```
    -trigraphs
```

Process ANSI standard trigraph sequences.

-pedantic

```
    -pedantic
```

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

-I

```
    -Idir
```

Add the directory *dir* to the end of the list of user-supplied directories to be searched for header files (see the section "The **#include** Command"). 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 later.

-I-

```
    -I-
```

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

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

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's requested explicitly with a **-I.** option. Specifying both **-I-** and **-I.** allows you to control precisely which directories are searched before the current one and which are searched after.

-F

```
    -Fdir
```

Add *dir* to the end of the list of directories in which to search for frameworks. Directories specified with **-F** are searched before the standard framework directories.

-nostdinc

```
    -nostdinc
```

Don't search the standard system directories for header files. Only the directories you specify with **-I** options (and the current directory, if appropriate) are searched.

-D

```
    -Dname
```

Predefine *name* as a macro, with definition **1**.

```
    -Dname=definition
```

Predefine *name* as a macro, with definition *definition*.

-U*name*

```
    -Uname
```

Don't predefine *name*. If both **-U** and **-D** are specified for one name, the name won't be predefined.

-undef

```
    -undef
```

Don't predefine any nonstandard macros.

-d

```
    -d
```

Produce a list of **#define** commands for all the macros defined during the execution of the preprocessor, instead of producing the normal preprocessing output.

-M

```
    -M
```

Produce a rule suitable for **make** describing the dependencies of the main source file, instead of outputting the result of preprocessing. The preprocessor produces 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 backslash-newline.

This feature is used in automatic updating of makefiles.

-MD

```
-MD
```

This is similar to **-M**, but the dependency information is written to files with names made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified; **-MD** doesn't inhibit ordinary compilation the way **-M** does.

-MM

```
-MM
```

This is similar to **-M**, but mentions only the files included with **#include "***file***"**. System header files included with **#include <***file***>** are omitted.

-MMD

```
-MMD
```

This is similar to **-MM**, but mentions only user header files, not system header files.

-H

```
-H
```

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

-i

```
-ifile
```

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 **-i***file* is to make the macros defined in *file* available for use in the main input.

Index