

Tutorial 1

The C Preprocessor

THINGS TO LOOK FOR...

- The major directives in the preprocessor language.
- How to write and work with simple and parameterized macros.
- How to control the code that is sent to the compiler for compilation.
- How to utilize conditional compilation.
- How to include external files in a build.
- How to identify the origin of a compile error in the original source.
- The purpose and use of the pragma directive.

1.0 INTRODUCTION

In this tutorial we will begin with a brief introduction to the C preprocessor. The preprocessor is simply a program that reads a C source file with embedded *preprocessor* directives and writes out a modified version of the source that can then be read and translated to object code by the compiler.

We will begin with the preprocessor language then examine and illustrate each of the major directives. These include working with simple and parameterized macros, conditional compilation, file inclusion, and several other useful tools.

1.1 The Preprocessor Language

The preprocessor language comprises a set of approximately 12-15 directives given in Table 1.0. Each is marked with the special character '#'. We note that these directives are not part of the specification of the C language.

1.1.1 Preprocessor Actions

When the preprocessor runs, it examines each line in the original source file. Lines beginning with the symbol '#' are interpreted as preprocessor directives and are expanded

or transformed into C source code as specified by that directive. Any other lines are viewed as C source text and are sent to the output without modification. At the end of the day, the output of the preprocessor must yield a C program.

1.1.2 Lexical Conventions

According to the current ANSI/ISO C standard, lines marked as preprocessor directives may have white space preceding or following the '#' on the same source line. Lines with only the # character, according to the ANSI/ISO C standard, define a null directive and are treated as blank lines.

Note:

Earlier preprocessors required that the # symbol begin in column 1 of the source line and it could not be followed by white space.

The remainder of the line following the directive may contain arguments that are subject to macro replacement. When the macro body is inserted into the source code, the arguments are replaced by the specified local variables. If no arguments are required, the remainder of the line should be empty. White space and comments are allowed.

<i>Directive</i>	<i>Definition</i>
#define	Define a preprocessor macro
#undef	Undefine or remove a preprocessor macro
#include	Insert contents of another source file
#if	Conditionally include contents of another source file
#ifdef	Conditionally include contents of another source file if macro name is defined
#ifndef	Conditionally include contents of another source file if macro name is not defined
#elif	Conditionally include contents of another source file if macro name is defined and previous #if, #ifdef, #ifndef, or #elif failed
#else	Alternative action if preceding #if, #ifdef, #ifndef, or #elif directive fails
#endif	Closes #if, #ifdef, #ifndef, or #elif construct
#line	Return line number for compiler message
defined	Return 1 if names is defined as preprocessor macro and 0 otherwise
# operator	Replaces the macro parameter with string constant containing parameter's value
## operator	Create single token from two adjacent tokens
#pragma	Specify proprietary information to the compiler
#error	Return a compile time error with associated message

1.2 Preprocessor Directives

Let's now look at each of the preprocessor directives in some detail. We'll begin with the preprocessor macro directive, `#define`.

1.2.1 Macro Expansion - Substitution

The preprocessor directive `#define` defines what is called a *macro*. A macro comprises a *name* and a *collection of text* or tokens associated with that name.

`#define` and `#undef`

The directive `#define` has several versions; the associated directive `#undef` undoes what `#define` has set up – seems strange, but, it works very well.

The syntax for these directives is given as,

Syntax

```
#define name text
#define name (arg1, arg2,... argn,) text
#undef name
```

The first form of the directive causes *name* to be defined as a macro. When, or if, *name* is subsequently encountered in the source file, all (unquoted) occurrences are replaced with *text*. Such replacement is called *macro expansion* or *substitution*. The argument *name* must be an identifier as defined by the C language. The expression given by *text* is called the *body* of the macro.

1.2.1.1 Simple Macros

The most common use of a basic macro in C is as a *symbolic constant*. We use symbolic constants rather than hard coding the variable values using what are called *magic numbers* in our code. This way, if the constant is changed, the change can be made in one place rather than trying to find all the places where the value might have been used.

Example 1.0

In Figure 1.0 we give several examples of how we might use symbolic constants.

```
#define MAXSIZE 2048
#define PI 3.14
#define TWOPI 6.28
#define TWOPI (3.14*2.0)
#define TWOPI (PI + PI)

int myArray [MAXSIZE];
int circumference = 0;
int radius = 0;
int area = 0;
circumference = TWOPI * radius // later in the program
area = PI * pow(r,2); // later in the program
```

Figure 1.0
Defining and Using Symbolic Constants

#define
macro
name
collection of
text
#define
#undef

name
text
macro expansion
substitution
body

symbolic
constant
magic
numbers

1.2.1.2 Macros with Parameters

The second form of the `#define` directive declares a formal parameter list immediately following the macro name. There can be no separating white space between the name and the argument list. If there is, the preprocessor assumes that the argument list is the macro's definition.

Example 1.1

Figure 1.1 gives an example of incorrectly defining a simple parameterized macro.

```
#define sum (x,y) ((x) + (y)) // The macro is incorrectly specified
#define sum(x,y) ((x) + (y)) // The macro is correctly specified
```

The white space between `sum` and `(x,y)` in the first version of the macro will cause the preprocessor to interpret `sum` as the macro name and `(x,y)` as the macro.

Figure 1.1

Using Parameterized Macros

The parameter list may be empty; however, if arguments are given, they must be identifiers. No two arguments can have the same name. Although they may be specified, there is no formal requirement that the arguments be used in the macro body.

We invoke a macro by writing its name followed by an opening left parenthesis, then a comma separated list of arguments (one for each parameter), and a closing right parenthesis. If no formal parameters are given, the definition must include an empty argument list. White space may appear between the name and the opening left parenthesis of the definition.

The formal argument list may contain properly balanced parenthesis, commas (if they are within a set of parenthesis), braces and subscripting brackets (which cannot contain commas and do not have to balance).

Example 1.2

We now use the macro in several places in the code fragment in Figure 1.2.

```
#define sum(x,y) ((x) + (y))

x = sum(2*a, b) / sum (c,d); // computes the quotient of the sums
                             // of two numbers
x = sum(2 * g(a,b), h(a,b)) / sum (c,d); // computes the quotient of the sums
                                         // of two numbers
                                         // the arguments are the return values
                                         // from two function calls.
```

Figure 1.2

Using Parameterized Macros

Example 1.3

The code fragment in Figure 1.3 give another example of how we might use a parameterized macro.

```
#define getSerial() getc(serialIn)

while ((c = getSerial()) != EOF)      // read a character from a serial port and
                                     // test for end of file
```

Figure 1.3

Using Parameterized Macros

Example 1.4

In Figure 1.4 we define a macro that takes an arbitrary statement as its argument.

```
#define assign(anyStatement) anyStatement

assign( {a = 1 ; b = 2;} )
assign ( {c = 0; d = 1; e = 2; } )
```

Figure 1.4

Using Parameterized Macros

The code as written is correct. The expression *anyStatement* acts like a variable that is instantiated when the macro is processed. For each instantiation of the macro above, *anyStatement* takes on the value of the expression in the curly brackets.

Note

The use of white space around the curly brackets.

A macro can be defined that looks like a function as we see in Figure 1.5. This is done when we want the C code for the function body to be placed in line, replacing the function invocation. The advantage of doing so is a possible increase in execution speed since the (time burden) cost of a function call is avoided. The penalty is increased code size.

```
#define max(A,B) ((A) > (B) ? (A) : (B))

// After substitution these become
max (3, 4); // ? 3 > 4 ? 3 : 4; if 3 is greater than 4, return 3 otherwise return 4
max (6, 5); // ? 6 > 5 ? 6 : 5; if 6 is greater than 5, return 6 otherwise return 5
```

Figure 1.5

Using Parameterized Macros

There are some potential problems with such a macro. Consider the following line of code,

```
max(i++, j++);
```

The statement appears to be a simple use of `max()`. However, watch what happens. When the macro is expanded, from the definition,

1. `((A) > (B))` replaced by `((i++) > (j++))`
2. `(A) : (B)` replaced by `((i++) : (j++))`

Based upon the values of A and B, potentially each variable is incremented twice.

`#undef`

```
Syntax
#undef name
```

The `#undef` macro is a companion to `#define` and is used to make the variable *name* no longer defined (great English, that). That is, it causes the preprocessor to forget the macro definition of *name*. Once *name* is undefined, it can be given a new definition using `#define` or it can be `#defined` once again as in the original macro. It is not an error to `#undef` a name that is not defined. In addition, macro expansion is not performed within the `#undef` directive.

Preprocessor lines are recognized and interpreted *before* macro expansion. If a macro *before* expands into something that looks like a preprocessor directive, the resulting, apparent directive will not be recognized as we see in Figure 1.6 in Example 1.5.

Example 1.5

```
#define STRLIB #include<string.h>
STRLIB
```

Figure 1.6

Using Parameterized Macros

The fragment is interpreted and expanded according to the following steps,

1. The line beginning with `#define` is recognized as a preprocessor macro.
2. The `#define` is processed.
3. The STRLIB substitution is executed – `#include<string.h>` replaces the character string 'STRLIB'.
4. The token sequence `#include<string.h>`, unprocessed, is passed to compiler as code.

There are times when a preprocessor directive exceeds the standard line length. When such a situation occurs, the preprocessor can be directed to continue interpretation and processing onto the next line. We use the *line continuation character* `'\'`, to so direct the preprocessor.

*line
continuation
character*

Example 1.6

In Figure 1.7, we use the line continuation character to tell the preprocessor that the code fragments on the second and third lines are part of the same expression.

```
#define DOLLAR $
#define BACKSLASH \
#define MODULUS |
```

Figure 1.7
Using Line Continuation

With the line continuation character, the directives in the example result in two lines not three as might be expected.

1.3 Conditional Compilation

There are several very handy directives that can be used to control how the program is assembled and compiled. That is, to stipulate which lines of source code are passed through to the output file (to be compiled) or ignored based upon a computed condition. These directives are particularly important if we want to build for different targets or when we are enabling or disabling debug code, for example. The syntax for each is given as,

```
Syntax
#if, #ifdef, #ifndef constant-expression
#else, #elif constant-expression
#endif
```

#if, #ifdef, #ifndef constant-expression

The statement *constant-expression* must evaluate to a constant arithmetic value; it may include macro substitution. For the case of the *#if* directive, if *constant-expression* is non-zero, subsequent lines of source code are passed on to the preprocessor output until a *#else*, *#elif*, or *#endif* is encountered. If *constant-expression* is a macro name, and if the macro name is defined, the source code is included. Finally, if *constant-expression* is a macro name, and if the macro name is not defined, the source code is excluded.

#else, #elif constant-expression

#else

Like the familiar else, if previous conditions fail, the source lines following the *#else* directive are passed on to the compiler.

#elif

The *#elif* directive is equivalent to the else if construct in C. Like the *#if* directive, *constant-expression* is evaluated with the same consequences.

#endif

The *#endif* directive closes the *#if* sequence. It acts like the closing curly bracket in the C code block construct.

Assume that our cross compiler can generate code for several related models of microprocessor. Let the variable TARGET identify which of those systems that we are working with. Based upon the selected target, we want to include a different header file containing parameters specific to each target. In Figure 1.8, we select between two different header files based upon the model for which we are compiling the code. We do this during the preprocessor phase of the compilation.

```

#ifndef TARGET
    #if MODEL == 'M0'
        #include model0.h
    #else
        #include model1.h
    #endif
#endif

```

Figure 1.8

Using Conditional Directives

Example 1.7

When we are developing an application, we will often add code to the source to help with debugging. Once we are satisfied with the functionality of the code, we don't want the memory burden of code that we no longer need and we also don't want to have the debug code routinely sending out debug messages – it upsets customers or other users. Looking ahead, we also don't want to remove it because as new features are added or the system is upgraded, the debug code may be useful again. The preprocessor can help us here. We can use the conditional directives to control whether or not our debug code is included in the current or final build of the application.

We can write,

<pre> #define DEBUG // DEBUG defined // debug code will be included #ifdef DEBUG my debugging code0 #endif // my program to include // debugging code0 #ifdef DEBUG my debugging code1 #endif // my program to include // debugging code1 </pre>	<pre> // #define DEBUG // DEBUG not defined // debug code will not be included #ifdef DEBUG my debugging code0 #endif // my program will not include // debugging code0 #ifdef DEBUG my debugging code1 #endif // my program will not include // debugging code1 </pre>
--	---

Figure 1.9

Using Conditional Directives

1.4 File Inclusion

File inclusion, probably the simplest of the preprocessor directives, has several different forms. We'll start with two of them.

Syntax
`#include <fileName>`
`#include "fileName"`

Either form replaces the current line with the entire contents of the named file. If the complete path is not given, the search for the file is determined by the form used. The angle brackets tell the preprocessor to look for the file in a standard location set when the compiler was installed. The double quotes tell the preprocessor to look for the file in the current directory.

The included file may contain `#include` directives as well. The permissible number of such inclusions is implementation dependent; ANSI/ISO C requires support for a minimum of 8. The preprocessor emits an error if the included file cannot be found.

In addition to the two more common forms, a third form of the `#include` directive is supported.

Syntax
`#include preprocessor tokens`

The tokens undergo normal macro expansion and the result of the expansion must match one of the previous two forms.

Example 1.8

If the program begins with the two lines in the code fragment in Figure 1.10 and if the cross compiler is on a PC,

```
#define french "g:\mysystem\include\french.h"
#include french
```

Figure 1.10
Using File Inclusion Directives

the preprocessor will look in the directory `g:\mysystem\include` for the header file `french.h`.

A similar directory path construct can be used for cross compilers running under LINUX™ or UNIX™

In the text, we discussed how to utilize the `#define` and `#include` preprocessor directives to manage larger and more sophisticated multiple file projects.

1.5 Miscellaneous Directives

The preprocessor includes several directives that provide support for compiler vendors or, once again, to aid in the debugging process. These include,

#error, #line, and #pragma

#error

```
Syntax
#error errorMessage
```

The *error* directive is used to write out a compile time error message. That message is subject to macro expansion. It is typically used in conditional statements to warn of inconsistencies, constraint violations, or incomplete information.

Example 1.9

In the code fragment in Figure 1.11, the error directive is used to flag problems during the preprocessor phase of the compilation.

```
#if defined (DEBUG) && defined(NDEBUG)
#error "Can't enable and disable debugging"
#endif
```

Figure 1.11
Using the error Directive

#line

```
Syntax
#line line-number "fileName"
#line line-number
```

When we build a program from multiple files, if an error occurs during compilation, it's sometimes useful to be able to annotate it with line numbers from the original file instead of normal sequential numbering. The information provided by *#line* directive is used to instantiate the values for the `__LINE__` and `__FILE__` preprocessor macros. The former hold the line number of a designated source program and must be a decimal integer constant. The latter is the name of the current source file and must be a string constant. The double underscore before and after the two macro names is part of the names.

Example 1.10

The line directive is used in the code fragment in Figure 1.12 to annotate the compilation process.

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char* myString = "Hello";

    #line 123 "myFile"
    printf ("This line is %d from %s\n", __LINE__, __FILE__);

    printf ("The string length is %d\n", strlen(myString));
    return;
}
/*
   When executed the program will print
   This line is 123 from myFile
   The string length is 5
*/
```

Figure 1.12
Using the line Directive

#pragma

The *pragma* directive is used to add preprocessor functionality that is not part of the ANSI/ISO C standard or to provide vendor defined information to the compiler. *pragma*

Syntax

#pragma tokens

The directive is particularly useful in the embedded world, because the standard C language does not include facilities for managing vendor specific hardware or features such as interrupts, timers, or input/output port management. The *pragma* directive allows such features and capabilities to augment the base language.

No restrictions are placed on permissible tokens. Compilers should ignore what they do not understand. The arguments to the directive are subject to macro expansion.

Example 1.11

Syntax
`#pragma inline`

The directive tells the compiler to expand the function body at the point of the function invocation thereby eliminating the overhead of the function call. We would do this when the specifications stipulate very tight time constraints. Observe the similarity to a macro.

Following the preprocessing step in the build of an embedded software program is the management of the variables and functions used in the program to get the real work done. We cover these in the text.

In our work with the variables and the functions in the program, interest centers on issues such as how to refer them, how much memory is needed to store them, how are they found in memory, what is their lifetime, and who can read, use, or change them. These issues are particularly important in embedded applications.

1.6 Summary

In this tutorial we began with an introduction to the preprocessor language then examined and illustrated each of the major preprocessor directives. These included simple and parameterized macros, conditional compilation, file inclusion, and several useful directives.

References

Austell-Wolfson, B., Otieno, R., *Complete Book of C Programming*, Prentice-Hall, Inc., 2000.

Antonakos, J., Mansfield Jr., K., *Structured C for Engineering and Technology*, 4th ed., Prentice-Hall, Inc., 2001.

Deitel, H., Deitel, P., *C How to Program*, 2nd ed., Prentice-Hall, Inc., 1994.

Hanly, J., Koffman, E., *Problem Solving and Program Design in C*, Addison-Wesley, Inc., 1996.

Harbison, S., Steele Jr., G., *C A Reference Manual*, 5th ed., Prentice-Hall, Inc., 2002.

Kernighan, B., Ritchie, D., *The C Programming Language*, 2nd ed., Prentice-Hall, Inc., 1988.

Miller, L., Quilici, A., *The Joy of C*, John Wiley & Sons, 1997.

Press, W., Teukolsky, S., Vetterling, W., Flannery, B., *Numerical Recipes in C The Art of Scientific Computing*, 2nd ed., Cambridge University Press, 1992.

Waite, M., Prata, S., *C Primer Plus*, Sams Publishing, 1993.