Sams Teach Yourself C in 24 Hours

Hour 23 - The C Preprocessor

Intelligence is the faculty of making artificial objects, especially tools to make tools.

—H. Bergson

In Hour 2, "Writing Your First C Program," you learned how to use the #include preprocessor directive to include C header files. Since then, the #include directive has been used in every program in this book. In this lesson you'll learn more about the C preprocessor and making macro definitions with the preprocessor directives. The following topics are discussed in this hour:

- What the C preprocessor can do
- Macro definitions and macro substitutions
- The #define and #undef directives
- How to define function-like macros with #define
- The #ifdef, #ifndef, and #endif directives
- The #if, #elif, and #else directives
- How to nest #if and #elif directives

What Is the C Preprocessor?

If there is a constant appearing in several places in your program, it's a good idea to associate a symbolic name to the constant, and then use the symbolic name to replace the constant throughout the program. There are two advantages to doing so. First, your program will be more readable. Second, it's easier to maintain your program. For instance, if the value of the constant needs to be changed, find the statement that associates the constant with the symbolic name and replace the constant with the new one. Without using the symbolic name, you have to look everywhere in your program to replace the constant. Sounds great, but can we do this in C?

Well, C has a special program called the C preprocessor that allows you to define and associate symbolic names with constants. In fact, the C preprocessor uses the terminology macro names and macro body to refer to the symbolic names and the constants. The C preprocessor runs before the compiler. During preprocessing, the operation to replace a macro name with its associated macro body is called macro substitution or macro expansion.

You can put a macro definition anywhere in your program. However, a macro name has to be defined before it can be used in your program.

In addition, the C preprocessor gives you the ability to include other source files. For instance, we've been using the preprocessor directive #include to include C header files, such as stdio.h, stdlib.h, and string.h, in the programs throughout this book. Also, the C preprocessor enables you to compile different sections of your program under specified conditions.

The C Preprocessor Versus the Compiler

One important thing you need to remember is that the C preprocessor is not part of the C compiler.

The C preprocessor uses a different syntax. All directives in the C preprocessor begin with a pound sign (#). In other words, the pound sign denotes the beginning of a preprocessor directive, and it must be the first nonspace character on the line.

The C preprocessor is line oriented. Each macro statement ends with a newline character, not a semicolon. (Only C statements end with semicolons.) One of the most common mistakes made by the programmer is to place a semicolon at the end of a macro statement. Fortunately, many C compilers can catch such errors.

The following sections describe some of the most frequently used directives, such as

TIP

> Macro names, especially those that will be substituted with constants, are normally represented with uppercase letters so that they can be distinguished from other variable names in the program.

The #define and #undef Directives

The #define directive is the most common preprocessor directive, which tells the preprocessor to replace every occurrence of a particular character string (that is, a macro name) with a specified value (that is, a macro body).

The syntax for the #define directive is

```
#define macro_name  macro_body
```

Here macro_name is an identifier that can contain letters, numerals, or underscores. macro_body may be a string or a data item, which is used to substitute each macro_name found in the program.

As mentioned earlier, the operation to replace occurrences of macro_name with the value specified by macro_body is known as macro substitution or macro expansion.

The value of the macro body specified by a #define directive can be any character string or number. For example, the following definition associates STATE_NAME with the string "Texas" (including the quotation marks):

```
#define STATE_NAME   "Texas"
```

Then, during preprocessing, all occurrences of STATE_NAME will be replaced by "Texas".

Likewise, the following statement tells the C preprocessor to replace SUM with the string (12 + 8):

```
#define SUM  (12 + 8)
```

On the other hand, you can use the #undef directive to remove the definition of a macro name that has been previously defined.

The syntax for the #undef directive is

```
#undef macro_name
```

Here macro_name is an identifier that has been previously defined by a #define directive.

The #undef directive "undefines" a macro name. For instance, the following segment of code:

```
#define STATE_NAME "Texas"
   printf("I am moving out of %s.\n", STATE_NAME);
#undef STATE_NAME
```

defines the macro name STATE_NAME first, and uses the macro name in the printf() function; then it removes the macro name.

Defining Function-Like Macros with #define

You can specify one or more arguments to a macro name defined by the #define directive, so that the macro name can be treated like a simple function that accepts arguments.

For instance, the following macro name, MULTIPLY, takes two arguments:

```
#define MULTIPLY(val1, val2)  ((val1) * (val2))
```

When the following statement:

```
result = MULTIPLY(2, 3) + 10;
```

is preprocessed, the preprocessor substitutes the expression 2 for val1 and 3 for val2, and then produces the following equivalent:

```
result = ((2) * (3)) + 10;
```

The program in Listing 23.1 is an example of using the #define directive to perform macro substitution.

TYPE
Listing 23.1. Using the #define directive.

```
1:  /* 23L01.c:  Using #define */
2:  #include <stdio.h>
3:
4:  #define METHOD       "ABS"
5:  #define ABS(val)     ((val) < 0 ? -(val) : (val))
6:  #define MAX_LEN      8
7:  #define NEGATIVE_NUM -10
8:
9:  main(void)
10: {
11:    char *str = METHOD;
12:    int array[MAX_LEN];
13:    int i;
14:
15:    printf("The orignal values in array:\n");
16:    for (i=0; i<MAX_LEN; i++){
17:       array[i] = (i + 1) * NEGATIVE_NUM;
18:       printf("array[%d]: %d\n", i, array[i]);
19:    }
20:
21:    printf("\nApplying the %s macro:\n", str);
22:    for (i=0; i<MAX_LEN; i++){
23:       printf("ABS(%d): %3d\n", array[i], ABS(array[i]));
24:    }
25:
26:    return 0;
27: }
```

OUTPUT

The following output appears on the screen after you run the executable 23L01.exe of the program in Listing 23.1:

```
C:\app>23L01
The orignal values in array:
array[0]: -10
array[1]: -20
array[2]: -30
array[3]: -40
array[4]: -50
array[5]: -60
array[6]: -70
array[7]: -80

Applying the ABS macro:
ABS(-10):  10
ABS(-20):  20
ABS(-30):  30
ABS(-40):  40
ABS(-50):  50
ABS(-60):  60
```

```
ABS(-70):  70
ABS(-80):  80
C:\app>
```

ANALYSIS
The purpose of the program in Listing 23.1 is to define different macro names, including a function-like macro, and use them in the program.

In lines 4_7, four macro names, METHOD, ABS, MAX_LEN, and NEGATIVE_NUM are defined with the #define directive. Among them, ABS can accept one argument. The definition of ABS in line 5 checks the value of the argument and returns the absolute value of the argument. Note that the conditional operator ?: is used to find the absolute value for the incoming argument. (The ?: operator was introduced in Hour 8, "More Operators.")

Then, inside the main() function, the char pointer str is defined and assigned with METHOD in line 11. As you can see, METHOD is associated with the string "ABS". In line 12, an int array called array is defined with the element number specified by MAX_LEN.

In lines 16_19, each element of array is initialized with the value represented by the (i + 1) * NEGATIVE_NUM expression that produces a series of negative integer numbers.

The for loop in lines 22_24 applies the function-like macro ABS to each element of array and obtains the absolute value for each element. Also, all absolute values are printed on the screen. The output from the program in Listing 23.1 proves that each macro defined in the program works very well.

Nested Macro Definitions
A previously defined macro can be used as the value in another #define statement. The following is an example:

```
#define ONE     1
#define TWO     (ONE + ONE)
#define THREE   (ONE + TWO)
result = TWO * THREE;
```

Here the macro ONE is defined to be equivalent to the value 1, and TWO is defined to be equivalent to (ONE + ONE), where ONE is defined in the previous macro definition. Likewise, THREE is defined to be equivalent to (ONE + TWO), where both ONE and TWO are previously defined.

Therefore, the assignment statement following the macro definitions is equivalent to the following statement:

```
result = (1 + 1) * (1 + (1 + 1));
```

WARNING

When you are using the #define directive with a macro body that is an expression, you need to enclose the macro body in parentheses. For example, if the macro definition is

```
#define SUM  12 + 8

result = SUM * 10;
```

becomes this:

```
result = 12 + 8 * 10;
```

which assigns 92 to result.
However, if you enclose the macro body in parentheses like this:

```
#define SUM  (12 + 8)

result = (12 + 8) * 10;
```

and produces the result 200, which is likely what you want.

Compiling Your Code Under Conditions

You can select portions of your C program that you want to compile by using a set of preprocessor directives. This is useful, especially when you're testing a piece of new code or debugging a portion of code.

The #ifdef and #endif Directives

The #ifdef and #endif directives control whether a given group of statements is to be included as part of your program.

The general form to use the #ifdef and #endif directives is

```
#ifdef macro_name
   statement1
   statement2
   . . .
   statementN
#endif
```

Here macro_name is any character string that can be defined by a #define directive. statement1, statement2, and statementN are statements that are included in the program if macro_name has been defined. If macro_name has not been defined, statement1, statement2, and statementN are skipped.

Because the statements under the control of the #ifdef directive are not enclosed in braces, the #endif directive must be used to mark the end of the #ifdef block.

For instance, the #ifdef directive in the following code segment:

```
. . .
#ifdef DEBUG
   printf("The contents of the string pointed to by str: %s\n", str);
#endif
. . .
```

indicates that if the macro name DEBUG is defined, the printf() function in the statement following the #ifdef directive is included in the program. The compiler will compile the statement so that the contents of a string pointed to by str can be printed out after the statement is executed.

The #ifndef Directive

The #ifndef directive enables you to define code that is to be executed when a particular macro name is not defined.

The general format to use #ifndef is the same as for #ifdef:

```
#ifndef macro_name
   statement1
   statement2
   . . .
   statementN
#endif
```

Here macro_name, statement1, statement2, and statementN have the same meanings as those in the form of #ifdef introduced in the previous section. Again, the #endif directive is needed to mark the end of the #ifndef block.

Listing 23.2 contains a program that demonstrates how to use the #ifdef, #ifndef, and #endif directives together.

TYPE
Listing 23.2. Using the #ifdef, #ifndef, and #endif directives.

```
1:  /* 23L02.c:  Using #ifdef, #ifndef, and #endif */
2:  #include <stdio.h>
3:
4:  #define UPPER_CASE  0
5:  #define NO_ERROR    0
6:
7:  main(void)
8:  {
9:     #ifdef UPPER_CASE
10:      printf("THIS LINE IS PRINTED OUT,\n");
11:      printf("BECAUSE UPPER_CASE IS DEFINED.\n");
12:    #endif
13:    #ifndef LOWER_CASE
14:      printf("\nThis line is printed out,\n");
15:      printf("because LOWER_CASE is not defined.\n");
16:    #endif
17:
18:    return NO_ERROR;
19: }
```

OUTPUT

The following output is shown on the screen after the executable 23L02.exe is created and run:

```
C:\app>23L02
THIS LINE IS PRINTED OUT,
BECAUSE UPPER_CASE IS DEFINED.

This line is printed out,
because LOWER_CASE is not defined.
C:\app>
```

ANALYSIS

The purpose of the program in Listing 23.2 is to use #ifdef and #ifndef directives to control whether a piece of message needs to be printed out.

Two macro names, UPPER_CASE and NO_ERROR, are defined in lines 4 and 5.

The #ifdef directive in line 9 checks whether the UPPER_CASE macro name has been defined. Because the macro name has been defined in line 4, the two statements in lines 10 and 11 are executed before the #endif directive in line 12 marks the end of the #ifdef block.

In line 13, the #ifndef directive tells the preprocessor to include the two statements in lines 14 and 15 in the program if the LOWER_CASE macro name has not been defined. As you can see, LOWER_CASE is not defined in the program at all. Therefore, the two statement in lines 14 and 15 are counted as part of the program.

The output from running the program in Listing 23.2 shows that the printf() functions in lines 10, 11, 14, and 15 are executed accordingly, under the control of the #ifdef and #ifndef directives.

The #if, #elif, and #else Directives

The #if directive specifies that certain statements are to be included only if the value represented by the conditional expression is nonzero. The conditional expression can be an arithmetic expression.

The general form to use the #if directive is

```
#if expression
   statement1
   statement2
```

```
   . . .
   statementN
#endif
```

Here expression is the conditional expression to be evaluated. statement1, statement2, and statementN represent the code to be included if expression is nonzero.

Note that the #endif directive is included at the end of the definition to mark the end of the #if block, as it does for an #ifdef or #ifndef block.

In addition, the #else directive provides an alternative to choose. The following general form uses the #else directive to put statement_1, statement_2, and statement_N into the program if expression is zero:

```
#if expression
   statement1
   statement2
   . . .
   statementN
#else
   statement_1
   statement_2
   . . .
   statement_N
#endif
```

Again, the #endif directive is used to mark the end of the #if block.

Also, a macro definition can be used as part of the conditional expression evaluated by the #if directive. If the macro is defined, it has a nonzero value in the expression; otherwise, it has the value 0.

For example, look at the following portion of code:

```
#ifdef DEBUG
   printf("The value of the debug version: %d\n", debug);
#else
   printf("The value of the release version: %d\n", release);
#endif
```

If DEBUG has been defined by a #define directive, the value of the debug version is printed out by the printf() function in the following statement:

```
printf("The value of the debug version: %d\n", debug);
```

Otherwise, if DEBUG has not been defined, the following statement is executed:

```
printf("The value of the release version: %d\n", release);
```

Now consider another example:

```
#if 1
 printf("The line is always printed out.\n");
#endif
```

The printf() function is always executed because the expression 1 evaluated by the #if directive never returns 0.

In the following example:

```
#if MACRO_NAME1 || MACRO_NAME2
   printf("MACRO_NAME1 or MACRO_NAME2 is defined.\n");
#else
   printf("MACRO_NAME1 and MACRO_NAME2 are not defined.\n");
#endif
```

the logical operator || is used, along with MACRO_NAME1 and MACRO_NAME2 in the expression evaluated by the #if directive. If one of the macro names, MACRO_NAME1 or MACRO_NAME2, has been defined, the expression returns a nonzero value; otherwise, the expression returns 0.

The C preprocessor has another directive, #elif, which stands for "else if." You can use #if and #elif together to build an if-else-if chain for multiple conditional compilation.

The program shown in Listing 23.3 is an example of using the #if, #elif, and #else directives.

TYPE
Listing 23.3. Using the #if, #elif, and #else directives.

```
1:  /* 23L03.c:  Using #if, #elif, and #else */
2:  #include <stdio.h>
3:
4:  #define C_LANG    `C'
5:  #define B_LANG    `B'
6:  #define NO_ERROR  0
7:
8:  main(void)
9:  {
10:    #if C_LANG == `C' && B_LANG == `B'
11:      #undef C_LANG
12:      #define C_LANG "I know the C language.\n"
13:      #undef B_LANG
14:      #define B_LANG "I know BASIC.\n"
15:      printf("%s%s", C_LANG, B_LANG);
16:    #elif C_LANG == `C'
```

```
17:      #undef C_LANG
18:      #define C_LANG "I only know C language.\n"
19:      printf("%s", C_LANG);
20:    #elif B_LANG == `B'
21:      #undef B_LANG
22:      #define B_LANG "I only know BASIC.\n"
23:      printf("%s", B_LANG);
24:    #else
25:      printf("I don't know C or BASIC.\n");
26:    #endif
27:
28:    return NO_ERROR;
29: }
```

OUTPUT

After the executable 23L03.exe is created and run, the following output is displayed on the screen:

```
C:\app>23L03
I know C language.
I know BASIC.
C:\app>
```

ANALYSIS

The purpose of the program in Listing 23.3 is to use the #if, #elif, and #else directives to select portions of code that are going to be compiled.

Inside the main() function, the #if directive in line 10 evaluates the conditional expression C_LANG == `C' && B_LANG == `B'. If the expression returns nonzero, then statements in lines 11_15 are selected to be compiled.

In line 11 the #undef directive is used to remove the C_LANG macro name. Line 12 then redefines C_LANG with the string "I know the C language.\n". Likewise, line 13 removes the B_LANG macro name and line 14 redefines B_LANG with another character string. The printf() function in line 15 prints the two newly assigned strings associated with C_LANG and B_LANG.

The #elif directive in line 16 starts to evaluate the expression C_LANG == `C' if the expression in line 10 fails to return a nonzero value. If the C_LANG == `C' expression returns nonzero, the statements in lines 17_19 are compiled.

If, however, the expression in line 16 also fails to return a nonzero value, the B_LANG == `B' expression in line 20 is evaluated by another #elif directive. The statements in lines 21_23 are skipped, and the statement in line 25 is compiled finally if the B_LANG == `B' expression returns 0.

In line 26 the #endif directive marks the end of the #if block that starts on line 10.

From the program in Listing 23.3 you can tell that C_LANG and B_LANG have been properly defined in lines 4 and 5. Therefore, the statements in lines 11_15 are selected as part of the program and compiled by the C compiled. The two character strings assigned to C_LANG and B_LANG during the redefinition are printed out after the program in Listing 23.3 is executed.

You're advised to change the value of the macros C_LANG and B_LANG to test the other executions in the program.

Nested Conditional Compilation

According to the ANSI C standard, the #if and #elif directives can be nested at least eight levels.

For example, the #if directive is nested in the following code segment:

```
#if MACRO_NAME1
   #if MACRO_NAME2
      #if MACRO_NAME3
         printf("MACRO_NAME1, MACRO_NAME2, and MACRO_NAME3\n");
      #else
         printf("MACRO_NAME1 and MACRO_NAME2\n");
      #endif
   #else
      printf("MACRO_NAME1\n");
   #endif
#else
   printf("No macro name defined.\n");
#endif
```

Here the #if directive is nested to three levels. Note that each #else or #endif is associated with the nearest #if.

Now let's have a look at another example in Listing 23.4, in which the #if directives are nested.

TYPE
Listing 23.4. Nesting the #if directive.

```
1:  /* 23L04.c: Nesting #if   */
2:  #include <stdio.h>
3:
4:  /* macro definitions */
5:  #define ZERO      0
6:  #define ONE       1
7:  #define TWO       (ONE + ONE)
8:  #define THREE     (ONE + TWO)
9:  #define TEST_1    ONE
10: #define TEST_2    TWO
11: #define TEST_3    THREE
12: #define MAX_NUM   THREE
13: #define NO_ERROR  ZERO
14: /* function declaration */
```

```
15: void StrPrint(char **ptr_s, int max);
16: /* the main() function */
17: main(void)
18: {
19:    char *str[MAX_NUM] = {"The choice of a point of view",
20:                          "is the initial act of culture.",
21:                          "--- by O. Gasset"};
22:
23:    #if TEST_1 == 1
24:       #if TEST_2 == 2
25:          #if TEST_3 == 3
26:             StrPrint(str, MAX_NUM);
27:          #else
28:             StrPrint(str, MAX_NUM - ONE);
29:          #endif
30:       #else
31:         StrPrint(str, MAX_NUM - TWO);
32:       #endif
33:    #else
34:       printf("No TEST macro has been set.\n");
35:    #endif
36:
37:    return NO_ERROR;
38: }
39: /* function definition */
40: void StrPrint(char **ptr_s, int max)
41: {
42:    int i;
43:
44:    for (i=0; i<max; i++)
45:       printf("Content: %s\n",
46:              ptr_s[i]);
47: }
```

OUTPUT

The following output is shown on the screen after the executable 23L04.exe is created and run on my machine:

```
C:\app>23L04
Content: The choice of a point of view
Content: is the initial act of culture.
Content: --- by O. Gasset
C:\app>
```

ANALYSIS

The purpose of the program in Listing 23.4 is to print the content of character strings controlled by the nested #if directives.

At the beginning of the program, nine macro names are defined in lines 5_13. The prototype of a function, StrPrint(), is given in line 15. Lines 19_21 define and initialize an array of char pointers called str.

The #if directives in lines 23_25 evaluate macro names, TEST_1, TEST_2, and TEST_3,

respectively. If the three macro names all return nonzero values, then in line 26, StrPrint() is invoked to print the content of all character strings pointed to by the pointers in the str array.

If, however, only TEST_1 and TEST_2 are nonzero, the statement in line 28 prints out the content of the MAX_NUM-ONE strings. Likewise, if only TEST_1 returns a nonzero value, the StrPrint() function is called in line 31 to print out the content of the MAX_NUM-TWO strings.

The last case is that TEST_1, TEST_2, and TEST_3 all return zero. Then the printf() function in line 34 is executed to display the message No TEST macro has been set. onscreen.

As you can tell from the program in Listing 23.4, TEST_1, TEST_2, and TEST_3 are all defined with nonzero constants; the content of all character strings referenced by the pointers of the str array are printed out as the output from the program.

You're advised to change the value of the macros TEST_1, TEST_2, and TEST_3 to test the other executions in the program.

Summary

In this lesson you've learned the following:

- The C preprocessor runs before the compiler. During preprocessing, all occurrences of a macro name are replaced by the macro body associated with the macro name.
- The C preprocessor also enables you to include additional source files to the program or compile sections of C code conditionally.
- The C preprocessor is not part of the C compiler.
- A macro statement ends with a newline character, not a semicolon.
- The #define directive tells the preprocessor to replace every occurrence of a macro name defined by the directive with a macro body that is associated with the macro name.
- The #undef directive is used to remove the definition of a macro name that has been previously defined.
- You can specify one or more arguments to a macro name defined by the #define directive.
- The #ifdef directive enables you to define code that is to be included when a particular macro name is defined.

- The #ifndef directive is a mirror directive to the #ifdef directive. The former enables you to define code that is to be included when a particular macro name is not defined.
- The #endif is used to mark the end of an #ifdef, an #ifndef, or an #if block.
- The #if, #elif, and #else directives enable you to select portions of code to compile.

In the next lesson you'll see a summary of what you've learned and what you can do after studying this book.

Q&A

Q Is the C preprocessor part of the C compiler?

A No. The C preprocessor is not part of the C compiler. With its own line-oriented grammar and syntax, the C preprocessor runs before the compiler in order to handle named constants, macros, and inclusion of files.

Q How do you remove a macro name?

A By putting a macro name after the #undef directive, the macro name can be removed. According to the ANSI C standard, a macro name has to be removed before it can be redefined.

Q Why do you need the #endif directive?

A The #endif directive is used with an #if, #ifdef, or #ifndef directive because statements under the control of a conditional preprocessor directive are not enclosed in braces ({ and }). Therefore, #endif must be employed to mark the end of the block of statements.

Q Can the conditional expression following the #if directive be an arithmetic expression?

A Yes. The conditional expression evaluated by the #if directive can be an arithmetic expression. If the expression returns a nonzero value, the code between the #if directive and the next nearest directive are included for compilation. Otherwise, the code is skipped.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. What's wrong with the following macro definition?

```
#define  ONE   1;
```

2. What is the final value assigned to result after the assignment statement is executed?

```
#define ONE       1
#define NINE      9
#define EXPRESS   ONE + NINE
result = EXPRESS * NINE;
```

3. What message will be printed out from the following code segment?

```
#define MACRO_NAME  0
#if MACRO_NAME
   printf("Under #if.\n");
#else
   printf("Under #else.\n");
#endif
```

4. What message will be printed out from the following code segment?

```
#define MACRO_NAME  0
#ifdef MACRO_NAME
   printf("Under #ifdef.\n");
#endif
#ifndef MACRO_NAME
   printf("Under #ifndef.\n");
#endif
```

Exercises

1. In Hour 18, "More Data Types and Functions," you learned how to define enum data. Rewrite the program in Listing 18.1 with the #define directive.
2. Define a macro name that can multiply two arguments. Write a program to calculate the multiplication of 2 and 3 with the help of the macro. Print out the result of the program.
3. Rewrite the program in Listing 23.2 with the #if, #elif, and #else directives.
4. Rewrite the program in Listing 23.3 with nested #if directives.