Chapter 6: The C Preprocessor

AIDS TO CLEAR PROGRAMMING

The preprocessor is a program that is executed just prior to the execution of the compiler. It's operation is transparent to you but it does a very important job. It removes all comments from the source and performs a lot of textual substitution based on your code, passing the result to the compiler for the actual compilation of your code.

Load and display the file named define.c for your first look at some defines and macros. Notice lines 4 through 7 of the program, each starting with **#define.** This is the way all defines and macros are defined. Before the actual compilation starts, the compiler goes through a preprocessor pass to resolve all of the defines. In the present case, it will find every place in the program where the combination START is found and it will replace it with the 0 since that is the definition. The compiler itself will never see the word START, so as far as the compiler is concerned, the zeros were always there. Note that if the string is found in a string constant or in a comment, it will not be changed.

It should be clear to you that putting the word START in your program instead of the numeral 0 is only a convenience to you and actually acts like a comment since the word START helps you to understand what the zero is used for.

In the case of a very small program, such as that before you, it doesn't really matter what you use. If, however, you had a 2000 line program before you with 27 references to START, it would be a completely different matter. If you wanted to change all of the STARTs in the program to a new number, it would be simple to change the one **#define** statement to the new value. If this technique were not used, it would be difficult to find and change all of the references to it manually, and possibly disastrous if you missed one or two of the references.

In the same manner, the preprocessor will find all occurrences of the word ENDING and change them to 9, then the compiler will operate on the changed file with no knowledge that ENDING ever existed.

It is a fairly common practice in C programming to use all capital letters for a symbolic constant such as START and ENDING and use all lower case letters for variable names. You can use any method you choose since it is mostly a matter of personal taste.

IS THIS REALLY USEFUL ?

When we get to the chapters discussing input and output, we will need an indicator to tell us when we reach the end-of-file of an input file. Since different compilers use different numerical values for this, although most use a minus 1, we will write the program with a **#define** to define the EOF used by our particular compiler. If, at some later date, we change to a new compiler, it is a simple matter to change this one **#define** to fix the entire program. In essentially all C compilers, the EOF is defined in the **stdio.h** file. You can observe this for yourself by examining the contents of the **stdio.h** file that was supplied with your compiler.

WHAT IS A MACRO ?

A macro is nothing more than another define, but since it is capable of at least appearing to perform some logical decisions or some math functions, it has a unique name. Consider line 6 of the program on your screen for an example of a macro. In this case, anytime the preprocessor finds the word MAX followed by a group in parentheses, it expects to find two terms in the parentheses and will do a replacement of the terms into the second part of the definition. Thus the first term will replace every **A** in the second part of the definition and the second term will replace every **B** in the second part of the definition. When line 15 of the program is reached, **index** will be substituted for every **A**, and **count** will be substituted for every **B**. Once again, it must be stated that string constants and comments will not be affected. Remembering the cryptic construct we studied a couple of chapters ago will reveal that **mx** will receive the maximum value of **index** or **count**. In like manner, the MIN macro will result in **mn** receiving the minimum value of **index** or **count**. These two particular macros are very common in C programs.

When defining a macro, it is imperative that there is no space between the macro name and the opening parenthesis. If there is a space, the compiler cannot determine that it is a macro, but will handle it like a simple substitution define statement.

The results of the macro usage are then printed out in line 17. There are a lot of seemingly extra parentheses in the macro definition but they are not extra, they are essential. We will discuss the extra parentheses in our next example program. Be sure to compile and execute define.c before going on to the next example program.

LET'S LOOK AT A WRONG MACRO

Load the file named macro.c and display it on your screen for a better look at a macro and its use. Line 4 defines a macro named WRONG that appears to evaluate the cube of **A**, and indeed it does in some cases, but it fails miserably in others. The second macro named CUBE actually does get the cube in all cases.

Consider the program itself where the CUBE of **i+offset** is calculated. If **i** is 1, which it is the first time through, then we will be looking for the cube of 1+5 = 6, which will result in 216. When using CUBE, we group the values like this, (1+5)*(1+5)*(1+5) = 6*6*6 = 216. However, when we use WRONG, we group them as 1+5*1+5*1+5 = 1+5+5+5 = 16 which is a wrong answer. The parentheses are therefore required to properly group the variables together. It should be clear to you that either CUBE or WRONG would arrive at a correct answer for a single term replacement such as we did in the last program. The correct values of the cube and the square of the numbers are printed out as well as the wrong values for your inspection.

In line 7 we define the macro ADD_WRONG according to the above rules but we still have a problem when we try to use the macro in lines 25 and 26. In line 26 when we say we want the program to calculate 5*ADD_WRONG(i) with i = 1, we get the result 5*1 + 1 which evaluates to 5 + 1 or 6, and this is most assuredly not what we had in mind. We really wanted the result to be 5*(1 + 1) = 5*2 = 10 which is the answer we get when we use the macro named ADD_RIGHT, because of the extra parentheses around the entire expression in the definition given in line 8. A little time spent studying the program and the result will be worth your effort in understanding how to use macros.

In order to prevent the above problems, most experienced C programmers include parentheses around each variable in a macro and additional parentheses around the entire expression. This will allow any macro to work correctly.

The remainder of the program is simple and will be left to your inspection and understanding.

CONDITIONAL COMPILATION - PART 1

The example program named ifdef.c is our first illustration of a conditional compilation. **OPTION_1** is defined in line 4, and is considered defined for the entire program. Therefore when the preprocessor gets to line 6, it keeps the text between lines 6 and 8 in the program and passes it to the compiler. If **OPTION_1** was not defined when we reach line 6, the preprocessor would throw away line 7 and the compiler would never see it. Likewise line 17 is conditionally compiled based on whether **OPTION_1** is defined or not. This is a very useful construct, but not the way we are using it here. Generally it is used to include a feature if we are using a certain processor, a certain operating system, or even a special piece of hardware.

You should compile and execute the program as is, then comment out line 4 so that **OPTION_1** will not be defined, and recompile and execute the program. You will see that the extra line will not be printed because it will be thrown away by the preprocessor. Keep in mind that the preprocessor does only textual substitution or text removal and you will be able to use it effectively.

Line 23 illustrates an undefine command to the preprocessor. This removes the fact that **OPTION_1** was defined and from this point on, the program acts as though it were never defined. Of course, it does no good here since the program is completed and there are no executable statements following the undefine, but it does illustrate the undefine statement.

You should move the undefine to line 5, recompile and execute the program, and you will see that it acts as though **OPTION_1** was never defined.

CONDITIONAL COMPILATION - PART 2

The next example program illustrates the preprocessor directive which includes code if a symbol in not defined. The ifndef directive reads literally, "if not defined", and with that much definition, its operation should be intuitive. This program will be a real exercise in logic for the diligent student, but should be understandable with a little effort. The symbol **OPTION_1** is reversed from the last program and the symbol **PRINT_DATA** is used to enable printing if it is not defined. If it is not defined, there will be some printout. This example program, much like the last one, is rather silly but illustrates the use of preprocessor directives. The next program is a little more practical.

CONDITIONAL COMPILATION - PART 3

The program named debugex.c is a good illustration of a very practical use of the preprocessor. In this program we define a symbol named **MY_DEBUG** at the beginning of the program. When we reach the code in the main program we see why it is defined. Apparently we do not have enough information to complete this code, so we sort of slopped it in until we have a chance to talk to Bill and Linda about how to do these calculations. In the meantime, we wish to continue work on other parts of the program, so we use the preprocessor to temporarily throw away this uncompilable code for us. Because of the obnoxious message we put into line 14, it will be impossible for us to forget about the bad state of affairs we left the code in, so we are forced to come back later and clean it up.

In this case, we are only concerned with a few lines of code, but it could be a large block of code we are working with. We could also be using this technique to handle several large blocks of code, some of which are in other modules, until Bill returns to explain the analysis and we can complete the undefined blocks.

MULTIPLE FILE PROGRAMS

For very small programs, it is expedient to include all of the code in a single file, and compile that

one file for the final resulting code. It is not generally acceptable to do this because all but the most trivial programs are too big to place in a single file because the file gets to be very cumbersome to work with. It is not at all unusual for a C program to be made up of over a thousand source files. It is, of course, necessary for these files to communicate and work together as one large program.

Even though it is best not to use global variables, a variable that is defined outside of any function, it is sometimes expedient to use a few. Sometimes these variables need to be referenced by two or more different files, and C provides a way to do this. Consider the following three file portions.

```
  FILE1.C                FILE2.C                FILE3.C
int index;             extern int index;      extern int index;
extern int count;      int count;
                       static int value;      int value;
                       int main();
static void one();     void two();            void three();
```

The variable named **index** declared in FILE1.C is available to any other file for use because it is declared globally. The other two files make use of the same variable by declaring it as an **extern** variable. In essence, they are telling the compiler, "I wish to use the variable named **index** which is defined somewhere else". Anytime **index** is referred to in either of the other two files, the variable of that name is used from FILE1.C, and it can be read, or modified by any of the three files. This provides an easy way to pass data from any file to any other file, but it could lead to problems. It would be very easy for any of these files to modify **index** in some way not meant to and corrupt the data. It could be very difficult to determine which file corrupted the value of **index**.

The variable named **count** is defined in FILE2.C and referred to in the same manner defined above within FILE1.C, but is not available for use in FILE3.C because it is not declared in it. A static variable, such as **value** in FILE2.C cannot be referenced in any other file but is hidden in the declaring file by definition. A completely separate variable named **value** is declared in FILE3.C that has nothing to do with the same named variable in FILE2.C. In this case, FILE1.C could declare value as an external variable and refer to that variable in FILE3.C if desired.

The **main()** entry point can only be called by the operating system to get the program started, but the functions **two()** and **three()** can be called from anywhere within the three files because they are global functions. The function **one()** however, because it is declared static, can only be called from within the file in which it is declared. It cannot be called from within FILE2.C or FILE3.C. It is sometimes expedient to "hide" a function within a file, and it is often referred to as a local function as opposed to being a global function.

WHAT IS AN ENUMERATION VARIABLE ?

Load and display the program named enum.c for an example of how to use the enum type variable. Line 6 contains the first **enum** type variable named **result** which is a variable that can take on any of the values contained within the braces. Actually the variable **result** is an **int** type variable and can be assigned any of the values defined for an **int** type variable. The names within the parentheses are **int** type constants and can be used anywhere it is legal to use an **int** type constant. The constant **WIN** is assigned the value of 0, **TIE** the value 1, **BYE** the value 2, etc.

In use, the variable named **result** is used just like any **int** variable would be used as can be seen by its use in the program. The **enum** type of variable is intended to be used by you, the programmer, as a coding aid since you can use a constant named **MON** for control structures rather than the meaningless (at least to you) value of 1. Notice that **days** is assigned the values of days of the week in the remainder of the program. If you were to use a switch statement, it would be much more meaningful to use the labels **SUN**, **MON**, etc, rather than the more awkward 0, 1, 2, etc.

All caps are used for the enumeration values in this program as a matter of personal taste because they are all constants. There is no universal standard on this matter and each programmer is free to do as he wishes. All caps for these values tends to be standard practice however.

WHAT IS A PRAGMA ?

A pragma is an instruction to your compiler to perform some particular action at compile time. Although pragmas vary from compiler to compiler and are not standardized, they perform some useful functions. Your compiler probably supports some way for you to select the optimization method by inserting a pragma into the source code. If your compiler provides a source listing file, you probably have pragmas to format the output listing to your personal preference. Check your documentation for the pragmas that are provided by your compiler.

Programming Exercise:

---

1. Write a program to count from 7 to -5 by counting down. Use **#define** statements to define the limits. (Hint, you will need to use a decrementing variable in the third part of the for loop control.
2. Add some printf statements in macro.c to see the result of the erroneous and correct addition macros.

Index Previous Next . . . . . *C Tutorial*

The Webwizard