14 The Preprocessor

14.1 INTRODUCTION

C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor*. The C preprocessor provides several tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives **#define** and **#include** to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 14.1.

Table 14.1 *Preprocessor Directives*

Directive	*Function*
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

14.2 MACRO SUBSTITUTION

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a *macro definition* (or simply a macro) takes the following general form:

#define *identifier string*

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string. The keyword **#define** is written just as shown (starting from the first column) followed by the *identifier* and a *string*, with at least one blank space between them. Note that the definition is not terminated by a semicolon. The *string* may be any text, while the *identifier* must be a valid C name.

There are different forms of macro substitution. The most common forms are:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants are:

```
#define    COUNT      100
#define    FALSE      0
#define    SUBJECTS   6
#define    PI         3.1415926
#define    CAPITAL    "DELHI"
```

Notice that we have written all macros (identifiers) in capitals. It is a convention to write all macros in capitals to identify them as symbolic constants. A definition, such as

```
#define  M  5
```

will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

```
total = M * value;
printf("M = %d\n", M);
```

These two lines would be changed during preprocessing as follows:

```
total = 5 * value;
printf("M = %d\n", 5);
```

Notice that the string **"M=%d\n"** is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define     AREA        5 * 12.46
#define     SIZE        sizeof(int) * 4
#define     TWO-PI      2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation

```
ratio = D/A;
```

where D and A are macros defined as follows:

```
#define  D    45 – 22
#define  A    78 + 32
```

The result of the preprocessor's substitution for D and A is:

```
ratio = 45–22/78+32;
```

This is certainly different from the expected expression

(45 – 22)/(78+32)

Correct results can be obtained by using parentheses around the strings as:

```
#define  D    (45 – 22)
#define  A    (78 + 32)
```

It is a wise practice to use parentheses for expressions used in macro definitions.

As mentioned earlier, the preprocessor performs a literal text substitution, whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement. This also suggests that we can use a macro to define almost anything. For example, we can use the definitions

```
#define     TEST        if (x > y)
#define     AND
#define     PRINT       printf("Very Good. \n");
```

to build a statement as follows:

TEST AND PRINT

The preprocessor would translate this line to

```
if(x>y) printf("Very Good.\n");
```

Some tokens of C syntax are confusing or are error-prone. For example, a common programming mistake is to use the token = in place of the token == in logical expressions. Similar is the case with the token &&.

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define     EQUALS      ==
#define     AND         &&
#define     OR          | |
#define     NOT_EQUAL   !=
#define     START       main() {
#define     END         }
#define     MOD         %
```

```
#define     BLANK_LINE      printf("\n");
#define     INCREMENT       ++
```

An example of the use of syntactic replacement is:

```
START
... ...
... ...
if(total EQUALS 240 AND average EQUALS 60)
INCREMENT count;
... ...
... ...
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

```
#define     identifier(f1, f2, . . . . . fn)     string
```

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2, ...,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define     CUBE(x)     (x*x*x)
```

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Consider the following statement:

```
volume = CUBE(a+b);
```

This would expand to:

```
volume = (a+b * a+b * a+b);
```

which would obviously not produce the correct results. This is because the preprocessor performs a blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*.

Example:

```
#define     CUBE(x)     ( (x) * (x) *(x) )
```

This would result in correct expansion of **CUBE(a+b)** as:

```
volume = ( (a+b) * (a+b) * (a+b));
```

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string*.

Some commonly used definitions are:

```
#define    MAX(a,b)      (((a) > (b)) ? (a) : (b))
#define    MIN(a,b)      (((a) < (b)) ? (a) : (b))
#define    ABS(x)        (((x) > 0) ? (x) : (–(x)))
#define    STREQ(s1,s2)  (strcmp((s1,) (s2)) == 0)
#define    STRGT(s1,s2)  (strcmp((s1,) (s2)) > 0)
```

The argument supplied to a macro can be any series of characters. For example, the definition

```
#define PRINT(variable, format)   printf("variable = %format \n", variable)
```

can be called-in by

```
PRINT(price x quantity, f);
```

The preprocessor will expand this as

```
printf( "price x quantity = %f\n", price x quantity);
```

Note that the actual parameters are substituted for formal parameters in a macro call, although they are within a string. This definition can be used for printing integers and character strings as well.

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define    M          5
#define    N          M+1
#define    SQUARE(x)  ( (x) * (x) )
#define    CUBE(x)    (SQUARE (x) * (x))
#define    SIXTH(x)   (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

```
((SQUARE(x) * (x)) * (SQUARE(x) * (x)))
```

Since **SQUARE (x)** is still a macro, it is further expanded into

```
( ( ((x)*(x)) * (x) ) * ( ((x) * (x)) * (x)) )
```

which is finally evaluated as x^6.

Macros can also be used as parameters of other macros. For example, given the definitions of M and N, we can define the following macro to give the maximum of these two:

```
#define MAX(M,N) (( (M) > (N) ) ? (M) : (N))
```

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define    HALF(x)    ( (x)/2.0)
#define    Y          HALF(HALF(x))
```

Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum of the three values x,y, and z:

MAX (x, MAX(y,z))

Undefining a Macro

A defined macro can be undefined, using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

14.5 FILE INCLUSION

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

#include "filename"

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let us assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include   <stdio.h>
#include   "SYNTAX.C"
#include   "STAT.C"
#include   "TEST.C"
#define    M       100
main ()
{
      ----------
      ----------
      ----------
}
```

14.4 COMPILER CONTROL DIRECTIVES

While developing large programs, you may face one or more of the following situations:

1. You have included a file containing some macro definitions. It is not known whether a particular macro (say, TEST) has been defined in that header file. However, you want to be certain that Test is defined (or not defined).
2. Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.
3. You are developing a program (say, for sales analysis) for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.
4. Suppose you are in the process of testing your program, which is rather a large one. You would like to have print calls inserted in certain places to display intermediate results and messages in order to trace the flow of execution and errors, if any. Such statements are called 'debugging' statements. You want these statements to be a part of the program and to become 'active' only when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately, the C preprocessor offers a feature known as *conditional compilation*, which can be used to 'switch' on or off a particular line or group of lines in a program.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include   "DEFINE.H"
#ifndef    TEST
#define    TEST 1
#endif
... ...
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined*, then all the lines between the **#ifndef** and the corresponding **#endif** directive are left 'active' in the program. That is, the preprocessor directive

 # define TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore the directive **#define TEST** is ignored. Remember, you cannot simply write

 # define TEST 1

because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
... ... ...
#ifdef   TEST
#undef    TEST
#endif
... ...
... ...
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again we cannot simply say

```
#undef  TEST
```

because, if TEST is not defined, the directive is erroneous.

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
... ...
... ...
main()
{
   ... ...
   ... ...
#ifdef IBM_PC
{
   ... ...
   ... ...          code for IBM_PC
   ... ...
}
#else
{
   ... ...
   ... ...          code for HP machine
   ... ...
}
#endif
   ... ..
   ... ..
}
```

If we want the program to run on IBM PC, we include the directive

```
#define IBM_PC
```

in the program; otherwise we don't. Note that the compiler control directives are inside the function. Care must be taken to put the # character at column one.

The compiler complies the code for IBM PC if **IBM-PC** is defined, or the code for the HP machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef   ABC
     group-A lines
#else
     group-B lines
#endif
```

Group-A lines are included if the customer **ABC** is defined. Otherwise, group-B lines are included.

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces wrong results.

The process of error detection and isolation begins with the testing of the program with a known set of test data. The program is divided down and **printf** statements are placed in different parts to see intermediate results. Such statements are called debugging statements and are not required once the errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive using control directives as:

```
... ...
... ...
#ifdef TEST
{
        printf("Array elements\n");
        for (i = 0; i< m; i++)
             printf("x[%d] = %d\n", i, x[i]);
}
#endif
... ..
... ..
#ifdef TEST
        printf(....);
#endif
... ...
```

The statements between the directives **#ifdef** and **#endif** are included only if the macro **TEST** is defined. Once everything is OK, delete or undefine the **TEST**. This makes the **#ifdef TEST** conditions false and therefore all the debugging statements are left out.

The C preprocessor also supports a more general form of test condition - **#if** directive. This takes the following form:

```
#if constant expression
{
        statement-1;
        statement-2;
        ... ...
        ... ...
}
#endif
```

The *constant-expression* may be any logical expression such as:

```
TEST <= 3
(LEVEL == 1 || LEVEL == 2)
MACHINE == 'A'
```

If the result of the constant-expression is nonzero (true), then all the statements between the **#if** and **#endif** are included for processing; otherwise they are skipped. The names **TEST, LEVEL**, etc. may be defined as macros.

14.5 ANSI ADDITIONS

ANSI committee has added some more preprocessor directives to the existing list given in Table 14.1. They are:

#elif	Provides alternative test facility
#pragma	Specifies certain instructions
#error	Stops compilation when an error occurs

The ANSI standard also includes two new preprocessor operations:

#	Stringizing operator
##	Token-pasting operator

elif Directive

The #elif enables us to establish an "if..else..if.." sequence for testing multiple conditions. The general form of use of **#elif** is:

```
#if   expression 1
      statement sequence 1
#elif expression 2
      statement sequence 2
      .....
      .....
        #elif  expression N
            statement sequence N
#endif
```

For example:

```
#if   MACHINE == HCL
      #define FILE "hcl.h"
```

```
#elif   MACHINE == WIPRO
            #define FILE "wipro.h"

        #elif   MACHINE == DCM
                    #define FILE "dcm.h"
#endif
#include FILE
```

pragma Directive

he **#pragma** is an implementation oriented directive that allows us to specify various in-ructions to be given to the compiler. It takes the following form:

#pragma *name*

here, *name* is the name of the **pragma** we want. For example, under Microsoft C,

#pragma loop_opt(on)

causes loop optimization to be performed. It is ignored, if the compiler does not recognize it.

#error Directive

The **#error** directive is used to produce diagnostic messages during debugging. The general form is

#error *error message*

When the **#error** directive is encountered, it displays the error message and terminates processing. Example.

```
#if !defined(FILE_G)
#error NO GRAPHICS FACILITY
#endif
```

Note that we have used a special processor operator **defined** along with **#if**. **defined** is a new addition and takes a *name* surrounded by parentheses. If a compiler does not support this, we can replace it as follows:

#if !defined	by	#ifndef
#if defined	by	#ifdef

Stringizing Operator

ANSI C provides an operator # called *stringizing operator* to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string. Consider the example below:

```
#define sum(xy) printf(#xy " = %f\n", xy)
main()
```

```
{
   ... ...
   ... ...
   sum(a+b);
   ... ...
}
```

The preprocessor will convert the line

```
sum(a+b);
```

into

```
printf("a+b" "=%f\n", a+b);
```

which is equivalent to

```
printf("a+b =%f\n", a+b);
```

Note that the ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator

The token pasting operator ## defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
   main()
   {
         ... ..
         ... ..
         printf("%f", combine(total, sales));
         ... ..
         ... ..
   }
```

The preprocessor transforms the statement

```
printf("%f", combine(total, sales));
```

into the statement

```
printf("%f", totalsales);
```

Consider another macro definition:

```
#define print(i) printf("a" #i "=%f", a##i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf("a5 = %f", a5)
```