20

C PREPROCESSOR

20.1 INTRODUCTION

We have already seen many features provided by C language. Yet another unique feature of the C language is the preprocessor. The C preprocessor provides several tools that are not available in other high–level languages. The programmer can use these tools to make his program more efficient in all respect.

20.2 OBJECTIVES

After going through this lesson you will be able to

- explain preprocessor working
- explain the # define Directive
- define constants
- explain macros
- write the various directions such as # undef, #include, #fdef, #ifdef, #ifndef, # else, #if

20.3 HOW THE PREPROCESSOR WORKS

When you issue the command to compile a C program, the program is run automatically through the preprocessor. The preprocessor is

a program that modifies the C source program according to directives supplied in the program. An original source program usually is stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. Some compilers enable the programmer to run only the preprocessor on the source program and to view the results of the preprocessor stage. All preprocessor directives begin with the number or sharp sign (#). They must start in the first column, and no space is required between the number sign and the directive. The directive is terminated not by a semicolon, but by the end of the line on which it appears.

The C preprocessor is a collection of special statements called directives, that are executed at the beginning of the compilation process. The #include and # define statements are preprocessor directives.

20.4 THE # DEFINE DIRECTIVE

The #define directive is used to define a symbol to the preprocessor and assign it a value. The symbol is meaningful to the preprocessor only in the lines of code following the definition. For example, if the directive #define NULL 0 is included in the program, then in all lines following the definition, the symbol NULL is replaced by the symbol. If the symbol NULL is written in the program before the definition is encountered, however, it is not replaced.

The # define directive is followed by one or more spaces or tabs and the symbol to be defined. The syntax of a preprocessor symbol is the same as that for a C variable or function name. It cannot be a C keyword or a variable name used in the program; if it is so a syntax error is detected by the compiler. For example, suppose a program contains the directive

```
#define dumb 52
```

which in turn is followed by the declaration

```
int dumb;
```

This would be translated by the preprocessor into

```
int 52;
```

which would be rejected by the compiler. The preprocessor does not check for normal C syntax, except for identifying quotation marks. It merely substitutes symbols where it finds them. The sym-

bol being defined is followed by one or more spaces or tabs and a value for the symbol. The value can be omitted, in which case the symbol is defined but not given a value. If this symbol is used later in the program, it is deleted without being replaced with anything.

If a # define directive does not fit on a single line, it can be continued on subsequent lines. All lines of the directives except the last must end with a backslash(\ ) character. A directive can be split only at a point where a space is legal.

20.5 CONSTANTS

A common use for defined symbols is the implementation of named constants. The following are examples of constants in C:

23, 'a', "hello"

Any of these can be assigned to a defined preprocessor symbol:

```
#define INTEGER 23
#define     CHARACTER 'a'
```

A defined symbol can specify only a complete constant. For example, if a program contains the definition

```
#define NULL 0
```

the number 120 cannot be represented by 12 NULL. A defined symbol can be recognized only if it is delimited by white space, punctuation or operators.

20.6 MACROS

When a constant is associated with a defined symbol, the characters making up the constants are assigned , not the numeric value of the constant. The definition

```
#define EOF-1
```

really assigns the string "-1" to EOF. The preprocessor replaces the symbol EOF by the characters "-1" without any consideration of the meaning of the two characters.

When a preprocessor symbol is defined as a segment of text, it is more generally called a macro. Macros are very useful in making C code more readable and compact. For example, some programmers include the following definitions in all their programs:

```
# define and &&
# define or ¦¦ ||
```

These definitions enable the programmer to write more readable code, such as the following:

```
If (a<b or c>d and e<f)
```

A comment can be included at the end of a macro or constant definition:

```
# define same 1 /* Nothing */
```

A comment can be inserted at any point in a C program where white space is permitted without adverse effect to the program.

Macros also can be used as abbreviation for lengthy and frequently used statements. The one restriction on macros is that, even though their definitions can occupy more than one line, they cannot include a newline. That is, when a macro is expanded all the resulting text is placed on the same line.

The preprocessor permits macros to have arguments, just as functions do; the difference is that the arguments are strings rather than values. Consider the following definition.

```
#define Decrement(x) if (x>0)x - = 1
```

when this macro is expanded, the string passed to the argument x replaces all occurrences of x in the symbol's value. That is, the statement

```
Decrement(a);
```

is expanded to

```
    if(a>0) a- = 1;
```

and the statement

```
Decrement(b);
```

becomes

```
    if (b>0) b - =1;
```

Notice that the macro definition itself does not include a semicolon. When the macro is invoked later, it is followed by an explicitly specified semicolon:

```
Decrement(a);
```

The argument names used in a macro are local to the macro. Thus there is no conflict between a macro's formal parameter names and identifiers used in the program itself. Any symbols used in the macro definition which are not argument name or names of other macros are left unchanged, and are assumed to be variable names or other C symbols.

The argument supplied to a macro can be any series of characters.

The comma is the delimiter that separates arguments in a macro.

A macro can be defined in terms of another macro, as in the following example of nested macros:

```
#defined control "%d\n"
#define printint(x) printf(control,x)
#define test(x) if (x>0) printint(x)
```

If a program contains the statements

```
Test(w);
```

Where w is an integer variable, the statement goes through the following conversion steps:

i) `if(w>0) printint(w);`

ii) `if (w>0) printf(CONTROL,w);`

iii) `if(w>0) printf ("%d\n,"w);`

INTEXT QUESTIONS

1. What, in general terms, is the role played by the C preprocessor ?
2. Which symbol always precedes preprocessor directives ?
3. Where on the line must all preprocessor directives begin ?
4. Where can a preprocessor directive be written ?
5. Which symbol is used to separate the arguments of a macro ?

20.7 USE OF DIRECTIVES: # undef, # include

(a) The # undef Directive:-

If a preprocessor symbol has already been defined, it must be undefined before being redefined. This is accomplished by the #undef directive, which specifies the name of the symbol to be undefined. It is not necessary to perform the readefinition at the beginning of a program. A symbol can be redefined in the middle of a program so that it has one value in the first part and another value at the end.

A symbol need not be redefined after it is undefined.

(b) The #include Directive:-

Often a programmer accumulates a collection of useful constants and macro definitions that are used in almost every program. It is desirable to be able to store these definitions in a file that can be inserted automatically into every program. This task is performed by the #include directive.

A file is inserted at a point where the #include directive is encountered. Such files are called header files, and by convention their names end with the character.h (as in stdio.h).

20.8 CONDITIONAL COMPILATION: #ifdef, #ifndef and #endif

Removing statements by hand would be quite tedious and could also lead to error. For this reason, the preprocessor provides directives for selectively removing section of code. This process is known as conditional compilation.

#define RECORD–FILE

If the # ifdef directive tests whether a particular symbol has been defined before the #ifdef is encountered. It does not matter what value has been assigned to the symbol. In fact, a symbol can be defined to the preprocessor without a value.

If the #ifdef directive is encountered after this definition it produce a true result. If, however, the directive #undef RECORD–FILE is encountered before the directive #ifdef RECORD–FILE then the preprocessor considers the symbol RECORD–FILE to be undefined and the #ifdef directive returns a false value.

If an #ifdef returns a true value, all the lines between the #ifdef and the corresponding #endif directive are left in the program. If those lines contains preprocessor directives, the directives are processed. In this way, conditional compilation directives can be nested. If the #ifdef evaluates as false, the associated lines are ignored, including any preprocessor directives that are included.

The statements need not all be grouped in one place. The #ifdef and #fendif directives can be used as many times as required. The preprocessor also provides the directive #ifndef, which produces a true result if a symbol is not defined. This makes it possible to use a single symbol to switch between two versions.

Conditional compilation can be used to select preprocessor directives as well as C code. For example suppose a header file is included in a program. and a certain preprocessor symbol (say, FLAG) may or may not be defined in that header file. If the programmer wants FLAG never to be defined, then the #include directive can be followed by

```
#ifdef FLAG
#undef FLAG
#endif
```

This ensures that, even if the symbol is defined in the header file, its definition is removed. It is not sufficient merely to work on

```
#undef FLAG
```

because if FLAG is not defined, the directive is erroneous. If FLAG should always be defined, then we would write

```
#ifndef FLAG
 #define FLAG
 #endif
```

We could, of course, give FLAG a value. We cannot simply write

```
#define FLAG
```

since if FLAG is already defined, an error probably will result.

20.9 THE #ELSE DIRECTIVE

This directive functions in much the same way as the else clause of an if statement.

All lines between an #ifdef or an #ifndef directive and the corresponding # else clause are included if the #ifdef or #ifndef is true. Otherwise, the lines between the #else and the corresponding #endif are included.

The # else directive and the lines that follow it can be omitted, but never the #endif directive. No other text can be included on the same line as the #else or #endif directive.

20.10 THE # IF DIRECTIVE

The #if directive tests an expression. This expression can be of any form used in a C program, with virtually any operators, except that it can include only integer constant values No variables, or function calls are permitted, nor are floating point, character or string constants.

The #if directive is true, if the expression evaluates to true (non–zero). Any undefined preprocessor symbol used in the #if expression is treated as if it has the value ϕ. Using a symbol that is defined with no value does not work with all preprocessors, and an attempt to do so might result in an error.

INTEXT QUESTIONS

6. What is the characteristics of the #else and the #endif directives?

7. How is the #ifndef directive used?

8. State the reason for using the #ifdef directive.

20.11 WHAT YOU HAVE LEARNT

In this lesson, you have learnt about preprocessors. The preprocessor is a program which modifies the C source program according to instructions provided. Remember that all preprocessor directives begin with the symbol #. There are different preprocessor directives. You also learnt about Macros. This lesson also discussed about conditional compilation.

20.12 TERMINAL QUESTIONS

1. How can a #define directive be continued to a new line ?

2. Where can a preprocessor directive be split ?

3. What is a macro, and what is it used for ?
4. What can a macro argument consist of ?
5. What role is played by the #undef directive?
6. How is the # include directive used ?
7. What is conditional compilation ?
8. How does the #if directive operate ?

20.13 KEY TO INTEXT QUESTIONS

1. It provides for the implementation of macros and conditional compilation.
2. The # sign.
3. The first column.
4. Anywhere in a program, so long as it starts in the first column of a line and is not on the same line as another directive or C statement.
5. The comma.
6. They only allow that text can be included on the lines they occupy.
7. It passes lines of code to the compiler only if the specificed preprocessor symbol is not defined.
8. The #ifdef directive is used in order to pass certain lines of code of the compiter, only if the specified preprocessor symbol is defined before the #ifdef directive is encounted.