**BCA-103**



**Preprocessors**

***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 that is known as preprocessor command lines or directives.*** Preprocessor directives are placed in the source program before themain line. **Before the source code passes through the compiler, it is examined by the** **preprocessor for any preprocessor directives. It there is any preprocessor directive, appropriate actions (as per the directives) are taken and then source program is handed over to the compiler**

The preprocessors begin with the symbol # in column one and do not require semi colon at the end. We have used the directive #include up to a limited extent

A set of commonly used preprocessor directives and their functions is given in following table

**Directive** **Function**

**#define** Defines a macro substitution

**#undef** Undefines a macro

**#include** Specifies a file to be included

**#ifdef** Test for macro definition

**#endif** Specifies the end of #if

**#ifndef** Test whether a macro is not defined

**#if** Test a compiler time condition

**#else** Specifies alternatives when #if test fails

These directives can be divided into four categories:

1. Macro substitution directives
2. File inclusion directive
3. Conditional compilation directives
4. Miscellaneous Directives

**Macro Substitution Directives:**

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 #defiine statement. It is usually known as macro definition (or simply macro) for example:

**#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 stirng.

**BCA-103**

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” |

It is convention to write all macros in capitals to identify them as symbolic constants. A macro definition can include more than a simple constant value. It can include expression as well following are valid definitions:

|  |  |  |  |
| --- | --- | --- | --- |
| **#define** | AREA | (5 \* 12.46) | |
| **#define** | TWO-PI | (2.0 \* 3.1415926) | |
| If you have created a definition such as | |  |  |
|  | #define | M | 5 |

Above macro will replace all occurrences of M with 5, starting from the line of definition to the end of the program. Consider the following example:

total = M \* value;

printf(“M = %d \n” , M);

The two lines would be changed during preprocessing as follows:

total = 5 \* value;

printf(“M = %d\n”, 5);

Macro definition can also include expressions as well for example:

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)

**Pre-processor**

**BCA-103**



**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, f3,… … … …fn) string

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

A simple example of a macro with argument is:

#define CUBE(x) ((x)\*(x)\*(x))

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));

**Nesting of Macros**

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

|  |  |  |
| --- | --- | --- |
| **#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 expends each #define macro, until no more macros appear in the text.

**File Inclusion**

This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

**#include "filename"**

**#include <filename>**

and it simply causes the entire contents of filename to be inserted into the source code at that point in the program The meaning of each of these forms is given below:

**#include "goto.c"**

This command would look for the file goto.c in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.

**#include <goto.c>**

This command would look for the file goto.c in the specified list of directories only.

**Conditional Compilation**

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

* #if
* #ifdef
* #ifndef

Optionally, an alternative block of text can be set aside with one of two directives:

* #else
* #elif

The end of the block or alternative block is marked by the #endif directive.

If the condition checked by #if , #ifdef , or #ifndef is true (nonzero), then all lines between the matching #else (or #elif ) and an #endif directive, if present, are ignored.

If the condition is false (0), then the lines between the #if , #ifdef , or #ifndef and an #else , #elif , or #endif directive are ignored.

## The #if Directive

The #if directive has the following syntax:

#if constant-expression newline

This directive checks whether the constant-expression is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++), decrement (- -), sizeof , pointer (\*), address (&), and cast operators.

## The #ifdef Directive

The #ifdef directive has the following syntax:

#ifdef identifier newline

This directive checks whether the identifier is currently defined. Identifiers can be defined by a #define directive or on the command line. If such identifiers have not been subsequently undefined, they are considered currently defined.

## The #ifndef Directive

The #ifndef directive has the following syntax:

#ifndef identifier newline

This directive checks to see if the identifier is not currently defined.

## The #else Directive

The #else directive has the following syntax:

#else newline

This directive delimits alternative source text to be compiled if the condition tested for in the corresponding #if , #ifdef , or #ifndef directive is false. An #else directive is optional.

## The #elif Directive

The #elif directive has the following syntax:

#elif constant-expression newline

The #elif directive performs a task similar to the combined use of the else-if statements in C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding #if , #ifdef , #ifndef , or another #elif directive is false and if the additional constant expression presented in the #elif line is true. An #elif directive is optional.

## The #endif Directive

The #endif directive has the following syntax:

#endif newline

This directive ends the scope of the #if , #ifdef , #ifndef , #else , or #elif directive.

The number of necessary #endif directives changes according to whether the elif or #else directive is used. Consider the following equivalent examples:

#if true #if true

. .

. .

. .

#elif true .

. #else

. #if false

. .

#endif .

.

#endif

#endif

**Miscellaneous Directives**

There are two more preprocessor directives available, though they are not very commonly used. They are:

(a) #undef

(b) #pragma

***#undef* Directive**

On some occasions it may be desirable to cause a defined name to become ‘undefined’. This can be accomplished by means of the **#undef** directive. In order to undefine a macro that has been earlier **#define**d, the directive,

#undef macro template

can be used. Thus the statement,

#undef PENTIUM

would cause the definition of PENTIUM to be removed from the system. All subsequent **#ifdef PENTIUM** statements would evaluate to false. In practice seldom are you required to undefine a macro, but for some reason if you are required to, then you know that there is something to fall back upon.

***#pragma* Directive**

This directive is another special-purpose directive that you can use to turn on or off certain features. Pragmas vary from one compiler to another.The ‘#pragma’ directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The forms of this directive (commonly known as *pragmas*) specified by C standard are prefixed with STDC. A C compiler is free to attach any meaning it likes to other pragmas.