Introduction to 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 what is known as *preprocessor command lines* or *directives*.

Preprocessor directives are placed in the source program before calling the *function main( )*.

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.

Table 11.1: Set of Commonly used Preprocessor Directives and their Functions

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

Macro Substitution (#define)

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.

#define: 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, the preprocessor replaces every occurrence of the *identifier* in the source code by the string.
- The string may be any text, while the identifier must be a valid C name.

Simple Macro Substitution

Simple string replacement is commonly used to define constants.

e.g.: #define TRUE 1

```
#define     CAPITAL    "DELHI"
```

A macro definition can include more than a simple constant value. It can include expressions as well.

e.g.:
```
#define     AREA 5 * 12.46
#define     SIZE  sizeof (int) * 4
```

Macros with Arguments

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

The identifiers f1, f2, ... fn are formal macro arguments that are analogous to the formal arguments in a function definition.

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.

e.g.: #define CUBE(x) (x*x*x)

If the statement volume = CUBE (side); appears later in the program

then the preprocessor would expand this statement to volume = (side * side * side);

> Parentheses should be used for each occurrence of a formal argument in the string. Also the whole string should also be enclosed within parentheses.

e.g.:
```
#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))
```

Undefining a Macro (#undef)

A defined macro can be undefined, using the statement *#undef identifier*.

This is useful when there is a need to restrict the definition only to a particular part of the program.

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 mark. In this case, the file is searched only in the standard directories, and this type of declaration is used only in case of standard header files. 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.

Conditional Compilation Directives (#if, #else, #elif, #endif, #ifdef, #ifndef)

#ifdef and #endif

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. This is achieved by inserting the preprocessing commands *#ifdef* and *#endif*, which have the general form.

```
#ifdef macroname
        statement1;
        statement2;
        statement3;
#endif
```

If macroname has been *#defined*, the block of code will be processed as usual, otherwise not.

e.g.:
```
main()
{
        #ifdef OKAY
                statement1;
                statement2;
        #endif
                statement3;
}
```

Here, statements 1 and 2 would get compiled only if the macro OKAY has been defined and the definition of the macro has been purposely omitted. Later on if these statements are to be compiled the only requirement is the deletion of *#ifdef* and *#endif*.

#ifndef (If Not Defined)

It works exactly opposite to *#ifdef*.

e.g.:
```
main( )
{
        #ifndef PCAT code suitable for a PC/XT
        #else        code suitable for a PC/AT
        #endif       code common to both the computers
}
```

Thus the program is portable, i.e., it is possible to work on two totally different computers.

#if and #elif

The *#if* directive can be used to test whether an expression evaluates to a non-zero value or not. If the result of the expression is non-zero, then subsequent lines upto a *#else*, *#elif* or *#endif* are compiled, otherwise they are skipped.

The conditional compilation directives can be nested as shown below:

```
#if ADAPTER == MA
        code for monochrome adapter
#else
        #if ADAPTER == CGA
                code for color graphics adapter
        #else
                #if ADAPTER == EGA
                        code for enhanced graphics adapter
                #else
                        #if ADAPTER == VGA
                                code for video graphics array
                        #else
                                code for super video graphics array
                        #endif
                #endif
        #endif
#endif
```

The above program can be made more compact by using another conditional compilation directive called #elif. The same program using this directive can be rewritten as shown below:

```
#if ADAPTER == MA
        code for monochrome adapter
#elif ADAPTER == CGA
        code for color graphics adapter
#elif ADAPTER == EGA
        code for enhanced graphics adapter
#elif ADAPTER == VGA
        code for video graphics array
#else
        code for super video graphics array
#endif
```