Preprocessor directives

The pre-processor is a program that processes the source code before it passes through the compiler. It operates under the control of pre-processor directives which is placed in the source code before the main( ).

The pre-processor directives are always preceded by a hash sign ( # ) directive. The words after # is called command. No semicolon ( ; ) can be placed at the end of a pre-processor directive.

The pre-processor directives are only one line long because as soon as a newline character is found, the pre-processor directive is considered to end. To extend a pre-processor directive to multiple lines, place a backslash ( \ ) at the last character of the line. It means that the line continued in the line following it.

**Types of pre-processor directives:**

Broadly there are two types of directives;

Unconditional Conditional

|  |  |
| --- | --- |
| Define | if |
| Line | else |
| Undef | elif |
| include | ifdef |
| Error | ifndef |
| pragma | endif |

**#define :**

The define pre-processor macros we use #define. The #define statement is also known as *macro definition* or simply macro.

Syntax: **#define identifier string**

The pre-processor replaces every occurrence of the identifier in the source code by a string. The string may be any text, a statement, or anything. However the identifier must be a valid C name.

EX: #define PI 3.141

A macro definition can also include expression. However, when using expressions for replacement, make sure the order of evaluation is correct.

Ex: #define rows 3

#define cols 3

#define size (rows \* cols)

Ex: use of #define for literal text substitution

#include<stdio.h>

#define INPUT printf("Enter a number:");scanf("%d",&num)

#define EQUALS ==

#define STARTS main(){

#define PRINT1 printf("Great")

#define PRINT2 printf("Try again")

#define END return 0;}

STARTS

int num;

INPUT;

if(num EQUALS 100)

PRINT1;

else

PRINT2;

END

Macros can stimulate functions and known as functions like macro. The name of the macro will then be used to replace the function call. The functions like macro includes a list of parameters.

Syntax: #define identifier(arg1,arg2,arg3……,argn) string

An identifier is followed by a parameter list in parentheses and the replacement string.

During invocation if the pre-processor encounters a function-like macro arguments substitution takes place.

Ex:

#define MUL(a,b) (a\*b)

int a=2,b=3,c;

c = MUL(a,b) ; // c= a\*b

Note- while using function-like macros, you must use parentheses to ensure correct evaluation of replacement text.

Ex:

#define SQUARE(x) (x\*x)

int a=2,b=3,c; c = SQUARE(a); //c= a\*a=2\*2

if we write here

c=SQUARE(a+b);

c= 2+3\*2+3; // c= 2+3\*2+3=2+6+3=11

**#undef** :

It is used to undefine the earlier defined macros. It simply ends the presence or validity of a specific macro.

Ex: #define CLOSE 0 // validity of CLOSE starts

…………

…………

#undef CLOSE // validity of CLOSE ends

…………

…………

**#include** :

An external file containing functions, variable or macro definitions can be included as a part of our program.

#include directive is used to inform the pre-processor to treat the contains of a specified file. It can be used in two forms.

#include<filename> and #include “filename”

The first type is used for system header files. But the second one is used for header files of programmers own program. We can include files other than header files. To include a .c file we must use the second type and inside the double quote use the file path location.

**Conditional directives :**

A conditional directive is used to instruct the pre-processor to select whether or not to include a chunk of code in the final token stream passed to the compiler.

**#ifdef :** It is used to check the existence of macro definitions. If the macro has been defined by a #define statement, then the code immediately following the command will be complied.

Syntax:

#ifdef MACRO

Controlled text

#endif

Here the controlled text will be included to output of pre-processor if and only if MACRO is defined i.e. #ifdef succeeds. Even if the controlled #ifdef directive fails the controlled text inside it is still run. So the programmer must ensure that the controlled text is lexically valid.

The **#endif** directive must be used to mark the end of the #ifdef block.

Ex :

#ifdef MAX

int STACK[MAX];

#endif

Here the stake array will not be created if MAX had not been initially defined.

**#ifndef :** It is opposite of #ifdef. If the macro has not been defined by the #define statement, then the code immediately following the command will be compile.

Ex:

#include<stdio.h>

#define country "India"

int main()

{

#ifdef country

printf("%s is a great country",country);

#endif

#ifndef country

printf("I love my Nation");

#endif

return 0;

}

The remaining three directives **#if**, **#elif** and **#else** are acts similar to the conditional statements i.e. **if**, **elseif** and **else** respectively. If the condition of these conditional directives are true then the pre-processor allows these statements for execution.

**##** **- Directive**:

It acts like an concatenation or merging operator. The ## directive is used with the #define macro.

## concatenates what before the ## with what after it.

EX:

#include<stdio.h>

#define action(a,b) a##b+a\*b

int main()

{

printf("%d",action(3,4));

return 0;

}