C Preprocessor Directives

The # include… or #define … etc of a C program are called preprocessor directives and are preprocessed by the preprocessor before actual compilation begins. The end of these lines is identified by the newline character '\n', no semicolon ';' is needed to terminate these lines.

Preprocessor directives are mostly used in defining macros, evaluating conditional statements, source file inclusion, pragma directives, line control, error detection, etc.

List of Preprocessor Directives in C

The following table lists all the preprocessor directives available in the C programming language:

Preprocessor Directives	Description
#define	Used to define a macro.
#undef	Used to undefine a macro.
#include	Used to include a file in the source code program.
#ifdef	Used to include a section of code if a certain macro is defined by #define.
#ifndef	Used to include a section of code if a certain macro is not defined by #define.
#if	Check for the specified condition.
#else	Alternate code that executes when #if fails.
#endif	Used to mark the end of #if, #ifdef, and #ifndef.
#error	Used to generate a compilation error message.
#line	Used to modify line number and filename information.
#pragma once	To make sure the header is included only once.

Preprocessor Directives	Description
#pragma message	Used for displaying a message during compilation.

Types of Preprocessor Directives in C

In C, preprocessor directives are categorized based on their functionalities following are the types of preprocessor directives:

1. Macro Definition
2. File Inclusion directive
3. Conditional Compilation
4. Line control
5. Error directive
6. Pragma directive

1. #define – Macro Directive

In C, macro definition directives uses the **#define** preprocessor directive to define the macros and symbolic constants. We use #define directive to define macro. Macro are basically the symbolic names that represents lines of code or some values. This directive is used to create constants or to define short, reusable codes.

Syntax

```
#define token value
```

Example

```
// C program to illustrate the use of #define directive
#include <stdio.h>
// Defining a macro for PI
#define PI 3.14159

int main()
{
   double radius = 8.0;
   double area = PI * radius * radius;    // Using the PI macro to calculate
   printf("Area of the circle is: %f\n", area);
   return 0;
}
```

Output

```
Area of the circle is: 201.061760
```

2. #include – File Inclusion Directive

#include is one of the file inclusion directive in C. #include preprocessor directive is used to include the content of one file to another file i.e. source code during the preprocessing stage. This is done to easily organize the code and increase the reusability of code.

Syntax

```
#include <file_name>
or
#include "filename"
```

Here, file inclusion with **double quotes (” ”)** tells the compiler to search for the header file in the directory of source file**.**

Example

The below example demonstrates the use of file inclusion directive #include.

1

```
// C program to demonstrate the use of file inclusion
#include <stdio.h>
#include "ce.h"

extern int var;

void test(void)
{
            var = 25;
            int a = 35;
                printf("\nVar in test : %d\n Content of a in fn is %d",var, a);
}


int main(void)
{

   register int r;
  printf("var is %d",var);
  test();
  printf("\nVar in main after call to test : %d, \nAfter update in main : %d \n",var, var+20)

  r = 300;
  auto int a = r;
  printf(" \n Register content is %d\n content of 'a' is %d\n", r,a);

  return 0;

}
```

```
[t66001@sflinuxonline 02.12.2024-13:33:36 - /~]$ cat ce.h
//#include<stdio.h>

int var = 56;
int ch = 34;
```

Output

```
var is 56
Var in test : 25
 Content of a in fn is 35
Var in main after call to test : 25,
After update in main : 45

 Register content is 300
 content of 'a' is 300
```

3. #if, #ifdef, #else, #elif, #endif – Conditional Compilation

Conditional Compilation directives help to compile a specific portion of the program or let us skip compilation of some specific part of the program based on some conditions.

#ifdef: This directive is the simplest conditional directive. This block is called a conditional group. The controlled text will get included in the preprocessor output if the macroname is defined. The controlled text inside a conditional will embrace preprocessing directives. They are executed only if the conditional succeeds. You can nest these in multiple layers, but they must be completely nested. In other words, '#endif' always matches the nearest '#ifdef' (or '#ifndef', or '#if'). Also, you can't begin a conditional group in one file and finish it in another.

Syntax

```
#ifdef MACRO
    controlled text
#endif
```

#ifndef: In #ifdef directive if the macroname is defined, then the block of statements after the #ifdef directive will be executed normally but in case it is not defined, the compiler will simply skip this block of statements. The #ifndef directive is simply the opposite of #ifdef directive. In case of #ifndef , the block of statements between #ifndef and #endif will get executed only if the macro or the identifier with #ifndef is not defined.

Syntax

```
ifndef macro_name
    statement1;
```

```
    statement2;
    statement3;
    .
    .
    .
    statementN;
endif
```

***Note**: If the macro with name as 'macroname' is not defined using the #define directive then only the block of statements will execute.*

#if, #else and #elif: All these directives works together and control compilation of portions of the program using some conditions. If the condition with the #if directive results in a non zero value, then the group of line immediately after the #if directive will be executed otherwise if the condition with the #elif directive evaluates to a non zero value, then the group of line immediately after the #elif directive will be executed else the lines after #else directive will be executed.

Syntax

```
#if macro_condition
   statements
#elif macro_condition
   statements
#else
   statements
#endif
```

Example

The below example demonstrates the use of conditional directives.

```
// C program to demonstrate the use of conditional
// directives.

#include <stdio.h>

#define gfg 7

#if gfg > 200
#undef gfg
#define gfg 200
#elif gfg < 50
#undef gfg
#define gfg 50
#else
#undef gfg
#define gfg 100
#endif
```

```
18
void printValue(int value) { printf("%d", value); }
20
21
int main()
22
{
23
   printValue(gfg); // gfg = 50
24
   return 0;
25
}
```

Output

```
50
```

***Note**: the entire structure of #if, #elif and #else chained directives ends with #endif.*

4. #line – Line Control

Whenever we compile a program, there are chances of occurrence of some error in the program. Whenever compiler identifies error in the program it provides us with the filename in which error is found along with the list of lines and with the exact line numbers where the error is. This makes easy for us to find and rectify error. However we can control what information should the compiler provide during errors in compilation using the #line directive.

Syntax

```
#line number "filename"
```

number – line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on. **“filename”** – optional parameter that allows to redefine the file name that will be shown.

```
1
Example :
// C program to demonstrate the use of #line
#include <stdio.h>
// Macro to print the current line number
6
#define PrintLineNum                              \
7
   printf("Line number is %d in file named %s\n",        \
8
          __LINE__, __FILE__)
int main()
11
```

```
{
    // Print the original line number
    PrintLineNum;

// Using #line to change line number and file name
// temporarily
#line 20 "main.c"
    PrintLineNum;

// revert to the original line number and file name
#line 30 "index.c"
    PrintLineNum;

    return 0;
}
```

Output

```
Line number is 13 in file named ./Solution.c
Line number is 20 in file named main.c
Line number is 30 in file named index.c
```

5. #error – Error Directive

The #error directive aborts the compilation process when it is found in the program during compilation and produces an error which is optional and can be specified as a parameter.

Syntax

```
#error optional_error
```

Here, **optional_error** is any error specified by the user which will be shown when this directive is found in the program.

Example

The below example demonstrate the use of error directive to display custom error message.

```
// C program to demonstrate the use of error directive to
```

```
// display custom error message.

#include <stdio.h>

#ifndef GeeksforGeeks
#error GeeksforGeeks not found!
#endif

int main()
{
    printf("Hello, GeeksforGeeks!\n");
    return 0;
}
```

Output

```
error: #error GeeksforGeeks not found !
```

// Courtesy: https://www.geeksforgeeks.org/cpp-preprocessor-directives-set-2/?ref=lbp