**Macros and Preprocessors in C**

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

| **Sr.No** | **Directives & Descriptions** |
| --- | --- |
| 1 | **#define** Substitutes a preprocessor macro. |
| 2 | **#include** Inserts a particular header from another file. |
| 3 | **#undef** Undefines a preprocessor macro. |
| 4 | **#ifdef** Returns true if this macro is defined. |
| 5 | **#ifndef** Returns true if this macro is not defined. |
| 6 | **#if** Tests if a compile time condition is true. |
| 7 | **#else** The alternative for #if. |
| 8 | **#elif** #else and #if in one statement. |
| 9 | **#endif** Ends preprocessor conditional. |
| 10 | **#error** Prints error message on stderr. |
| 11 | **#pragma** Issues special commands to the compiler, using a standardized method. |

The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

**The following are some interesting facts about preprocessors in C.**

**1)** When we use ***include***directive,  the contents of included header file (after preprocessing) are copied to the current file.   
Angular brackets **<** and **>** instruct the preprocessor to look in the standard folder where all header files are held.  Double quotes **“** and **“** instruct the preprocessor to look into the current folder (current directory).

**2)**When we use***define***for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program *max*is defined as 100.

|  |
| --- |
| #include<stdio.h>  #define max 100  int main()  {      printf("max is %d", max);      return 0;  } |

**Output:**

max is 100

**3)** The macros can take function like arguments, the arguments are not checked for data type. For example, the following macro INCREMENT(x) can be used for x of any data type.

|  |
| --- |
| #include <stdio.h>  #define INCREMENT(x) ++x  int main()  {      char \*ptr = "GeeksQuiz";      int x = 10;      printf("%s  ", INCREMENT(ptr));      printf("%d", INCREMENT(x));      return 0;  } |

**Output:**

eeksQuiz 11

**4)** The macro arguments are not evaluated before macro expansion. For example, consider the following program

|  |
| --- |
| #include <stdio.h>  #define MULTIPLY(a, b) a\*b  int main()  {      // The macro is expanded as 2 + 3 \* 3 + 5, not as 5\*8      printf("%d", MULTIPLY(2+3, 3+5));      return 0;  }  // Output: 16 |

The previous problem can be solved using following program

|  |
| --- |
| #include <stdio.h>  //here, instead of writing a\*a we write (a)\*(b)  #define MULTIPLY(a, b) (a)\*(b)  int main()  {      // The macro is expanded as (2 + 3) \* (3 + 5), as 5\*8      printf("%d", MULTIPLY(2+3, 3+5));      return 0;  }  // This code is contributed by Santanu |

**Output:**

40

**5)** The tokens passed to macros can be concatenated using operator **##** called Token-Pasting operator.

|  |
| --- |
| #include <stdio.h>  #define merge(a, b) a##b  int main()  {      printf("%d ", merge(12, 34));  } |

**Output:**

1234

**6)** A token passed to macro can be converted to a string literal by using # before it.

|  |
| --- |
| #include <stdio.h>  #define get(a) #a  int main()  {      // GeeksQuiz is changed to "GeeksQuiz"      printf("%s", get(GeeksQuiz));  } |

**Output:**

GeeksQuiz

**7)** The macros can be written in multiple lines using ‘\’. The last line doesn’t need to have ‘\’.

|  |
| --- |
| #include <stdio.h>  #define PRINT(i, limit) while (i < limit) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }  int main()  {      int i = 0;      PRINT(i, 3); i, limit) while (0 < 3) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }  PRINT(2,3); 2, 3) while (i < limit) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }  PRINT(4,5) 4, 5) while (i < limit) \                          { \                              printf("GeeksQuiz "); \                              i++; \                          }      return 0;  } |

**Output:**

GeeksQuiz GeeksQuiz GeeksQuiz

**8)**The macros with arguments should be avoided as they cause problems sometimes. And Inline functions should be preferred as there is type checking parameter evaluation in inline functions.

For example consider the following program. From first look the output seems to be 1, but it produces 36 as output.

|  |
| --- |
| #include <stdio.h>   #define square(x) x\*x  int main()  {      // Expanded as 36/6\*6      int x = 36/square(6); 36/6\*6 6\*6 36      printf("%d", x);      return 0;  } |

**Output:**

36

If we use inline functions, we get the expected output. Also, the program given in point 4 above can be corrected using inline functions.

|  |
| --- |
| #include <stdio.h>    static inline int square(int x) { return x\*x; }  int main()  {  int x = 36/square(6); square of x is replaced by 36/(x\*x) 36/(6\*6) 36/36=1  printf("%d", x);  return 0;  } |

**Output:**

1

**9)** Pre-processors also support if-else directives which are typically used for conditional compilation.

|  |
| --- |
| int main()  {  #if VERBOSE >= 2    printf("Trace Message");  #endif  } |

**Output:**

No Output

**10)** A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like ***defined***, ***ifdef***and ***ifndef***are used.

**11)** There are some standard macros which can be used to print program file (\_\_FILE\_\_), Date of compilation (\_\_DATE\_\_), Time of compilation (\_\_TIME\_\_) and Line Number in C code (\_\_LINE\_\_)

|  |
| --- |
| #include <stdio.h>    int main()  {     printf("Current File :%s\n", \_\_FILE\_\_ );     printf("Current Date :%s\n", \_\_DATE\_\_ );     printf("Current Time :%s\n", \_\_TIME\_\_ );     printf("Line Number :%d\n", \_\_LINE\_\_ );     return 0;  } |

**Output:**

Current File :/usr/share/IDE\_PROGRAMS/C/other/081c548d50135ed88cfa0296159b05ca/081c548d50135ed88cfa0296159b05ca.c

Current Date :Sep 4 2019

Current Time :10:17:43

Line Number :8

**12)** We can remove already defined macros using :   
**#undef MACRO\_NAME**

|  |
| --- |
| #include <stdio.h>  #define LIMIT 100  int main()  {     printf("%d",LIMIT);     //removing defined macro LIMIT     #undef LIMIT     //Next line causes error as LIMIT is not defined     printf("%d",LIMIT);     return 0;  } |

Following program is executed correctly as we have declared LIMIT as an integer variable after removing previously defined macro LIMIT

|  |
| --- |
| #include <stdio.h>  #define LIMIT 1000  int main()  {     printf("%d",LIMIT);     //removing defined macro LIMIT     #undef LIMIT     //Declare LIMIT as integer again     int LIMIT=1001;     printf("\n%d",LIMIT);     return 0;  } |

**Output:**

1000

1001

Another interesting fact about macro using (**#undef**)

|  |
| --- |
| #include <stdio.h>  //div function prototype  float div(float, float);  #define div(x, y) x/y    int main()  {  //use of macro div  //Note: %0.2f for taking two decimal value after point  printf("%0.2f",div(10.0,5.0));  //removing defined macro div  #undef div  //function div is called as macro definition is removed  printf("\n%0.2f",div(10.0,5.0));  return 0;  }    //div function definition  float div(float x, float y){  return y/x;  } |

**Output:**

2.00

0.50