The C Preprocessor

Introduction: The C preprocessor is a program that processes the source program before it is passed to the compiler. The 'C' program written by the programmer is known as the 'source code'. The preprocessor works on this 'source code' and creates the 'expanded source code' which is sent to the compiler for compilation.

Features of C Preprocessor:-The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begins with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition.

C has the following preprocessor directives:

1) Macro expansion.
2) File inclusion.
3) Conditional Compilation.
4) Miscellaneous directives.

1) Macro expansion- Have a look at the following program.

```
#define LIMIT 40
 void main( )
  {
     int i ;
     for ( i = 1 ; i <= LIMIT ; i++ )
     printf ( "\n%d", i ) ;
   }
```

In this program instead of writing 40 in the **for** loop we are writing it in the form of LIMIT, which has already been defined before **main( )** through the statement,

```
#define LIMIT 40
```

This statement is called 'macro definition' or 'macro'. The 'LIMIT' in the above statement is called 'macro templates', whereas, '40' is called 'macro expansions'.

The actual thing: When we compile the program, before the source code passes to the compiler it is examined by the **C** preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

Remember:

1. In C programming it is customary to use capital letters for macro template.
2. A macro template and its macro expansion are separated by blanks or tabs.
3. A space between **#** and **define** is optional.
4. A macro definition is never to be terminated by a semicolon.

Importance of 'Macro':

1. We can change values of a constant at all the places in the program by just making a change in the **#define** directive. This convenience may not matter for small programs shown above, but with large programs macro definitions are almost indispensable.

2. The compiler can generate faster and more compact code for constants than it can for variables and using a variable for what is really a constant makes the program more difficult to understand and there is always a danger that the variable may inadvertently get altered somewhere in the program.

Other usage:

1. A **#define** directive is many a times used to define operators.

Example:
```
#define AND &&
#define OR ||
void main( )
  {
     int a= 21, b=20 ;
     if ( ( a>b ) AND (a!=b) )
     printf ( "True" ) ;
     else
     printf ( "False" ) ;
   }
```

2. A **#define** directive could be used even to replace a condition.

Example:
```
#define AND &&
#define RANGE (a > 25 AND a < 50)
  void main( )
   {
     int a = 30 ;
     if ( RANGE )
     printf ( "within range" ) ;
     else
     printf ( "out of range" ) ;
   }
```

3. A **#define** directive could be used to replace even an entire C statement.

Example:
```
#define PRINT printf ( "Enter a Number : ") ;
    void main()
     {
        int a,b;
        PRINT
        Scanf("%d",&a);
        b=++a;
        printf("%d",b);
      }
```

4. Macros can have arguments, just as functions can.

Example:

```
#define AREA(x) (3.14 * x * x)
  void main ( )
   {
     float b = 57.12, c = 45.23, a ;
     a = AREA ( b) ;
      printf ( "\nArea of circle = %f", a ) ;
      a = AREA ( c) ;
      printf ( "\nArea of circle = %f", a ) ;
    }
```

5. Macros can be split into multiple lines, with a '\' (back slash) present at the end of each line.

Note: 1. Don't leave a blank between the macro template and its argument while defining the macro. Thus, its wrong to write **AREA (x)** instead of **AREA(x)**.

2. The entire macro expansion should be enclosed within parentheses.

Difference with functions-

- In a macro call the preprocessor replaces the macro template with its macro expansion. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

- Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact. But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program.

- If the macro is simple and short like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

File Inclusion

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

```
#include "filename"
```

and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program.

Purpose:

- If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are **#included** at the beginning of main program file.

- There are some functions and some macro definitions that we need almost in all programs that we write. These commonly needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statements in this file to our program as if we have typed them in.

Note: It is common for the files that are to be included to have a **.h** extension. This extension stands for 'header file', possibly because it contains statements which when included go to the head of your program. The prototypes of all the library functions are grouped into different categories and then stored in different header files.

For example prototypes of all mathematics related functions are stored in the header file 'math.h', prototypes of console input/output functions are stored in the header file 'conio.h', and so on.

Actually there exist two ways to write **#include** statement. These are:

```
#include "filename"
#include <filename>
```

#include "filename" - This command would look for the file 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 <filename>- This command would look for the file in the specified list of directories only.