Pre-processors

Pre-processor is a program that processes the source code before it passes through the compiler

Pre-processors

- The lines starting with # are known as preprocessor directives.When the preprocessor finds a line starting with the symbol #,it consider it as a command for itself and works accordingly.All the directives are executed by the preprocessor,and thecompiler does not receive any line starting with # symbol.

Some feature of preprocessor directives are-

1. Pre-processor directives begin with the symbol #.
2. There can be only one preprocessor directive on a line
3. There is no semicolon on a line
4. A directive is active from the point of its appearance till the end of program.

Main functions performed by preprocessor directives are-

1.Simpe macro substitution

2.Macro with arguments

3.Conditional compilation

Pre-processor Directives

Directives	Functions
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for macro definition
#endif	Specifies the end of #if
#ifndef	Test whether the macro is not defined
#if	Test a compiler time condition

#define

#define macro_name macro_expansion

- Here macro_name is any valid C identifier and it is generally taken in capital letters to distinguish it from other variables.
- Macro_expansion can be any text.
- A space is necessary between macro_name and macro_expansion.
- Pre-processor replaces all the occurrence of macro_name with the macro_expansion.
- For example
  - #define PI 3.14
  - #define MAX 100
- Pre-processor searches for the macro_name in the source code and replaces it with macro_expansion.
- For wherever the macro name PI appears in the code,it is replaced by 3.14

Problems using Macros

```
#include<stdio.h>
#include<conio.h>
#define PI  3.14
void main()
{
int radius=12;
area=PI*radius*radius;
circumference=2*PI*radius;
printf("%d",area);
printf("%d",circumference);
getch();
}
```

Macros with arguments

#define macro_name(arg1,arg2,…) macro_expansion

For example:

```
#define SUM(x,y)  ((x)+(y))
#define PRODUCT(x,y)  ((x)*(y))
#define SQUARE(x)  ((x)*(x))
#define MAX(x,y)  ((x)>(y)?(x):(y))
#define CIRCLE(rad)  (3.14*(rad)*(rad))
```

Problems using Macros

```
#include<stdio.h>
#include<conio.h>
#define PROD(a,b) ((a)*(b))
void main()
{
int a=6,b=12;
p=PROD(a,b);
printf("%d",p);
getch();
}
```

Macros Vs Functions

- We have seen macros with arguments can perform task similar to function.
- A macro is expanded into inline code so the text of macro is inserted into the code for each macro call. Hence macro makes the code lengthy and the compilation time increases.On the other hand the code of a function is written only at one place, regardless of the number of times it is called so the use of functions makes the code smaller.
- In function ,the passing of arguments and returning a value takes some time and hence the execution of the program becomes slow while in case of macros this time is saved and they make the program faster.
- So functions are slow but take less memory while macros are fast but occupy more memory due to duplicity of code