Preprocessor

Def^n → Preprocessor converts source code (PR.C) into expanded source code (PR.I).

- The preprocessor offers several features called preprocessor directives.
- Each of these preprocessor directives begin with a # symbol.
- The directives are generally used at the beginning of a program.

Types of Preprocessor Directives

(1) File Inclusion (#include)

(2) Macro expansion (#define)

(3) Conditional Compilation (#ifdef, #endif, #else, #ifndef)

(4) Miscellaneous Directives (#undef, #pragma).

(1) File Inclusion (#include)

Used to include files in your program.

2 ways :-

(a) #include "filename"

(b) #include < filename >

(a) #include "goto.c" ⟹

This command would look for the file goto.c in the current directory as well as the specified list of directories set in the include search path. Means if you are creating your own header file then you will have to include your file using this method.

#include "myfile.h"

(b) #include < goto.c >

This command would look for the file goto.c in the specified list of directories only. Means when you have to include file that have already been designed in C language eg

#include < math.h >

(2) Macro Expansion (# define)

- They are used to define symbolic constants.
  Eg # define PI 3.1415
- They can be used to replace any code with macro.
- They can be used like functions and it is called Macros with arguments.

define MACRO TEMPLATE MACRO EXPANSION.

Eg (1)
```
# define UPPER 25
main()
{
  int i;
  for (i=1; i<=UPPER; i++)
    { printf ("%d", i);
    }
```

In this program, # define UPPER 25 is called macro definition. or just a macro.

UPPER is a MACRO Template

25 is a — Expansion.

Wherever this word UPPER will be used in a program, whenever program executes, UPPER will be replaced by 25.

Eg # define PI 3.1415

```
main()
{
    float r = 6.25;
    float area;
    area = PI * r * r;
    printf("Area of circle = %f", area);
}
```

⇒ When we compile the program, before the source code passes to the compiler, it is examined by C preprocessor for any macro definitions. All macro templates will be replaced by its macro expansion.

⇒ It is customary to use CAPITAL letters for macro template.

⇒ The question may arise, why we use PI as a constant. We can also use it as a variable. and if we change the value of PI in a variable, that will be changed in all places where the PI is used.

Reason to use macro rather than variable.

⇒ It is efficient if we declare it as a variable, since the compiler can generate faster and more compact code for constants than it can for variables.

⇒ Variables takes up space in memory and hold the memory but constants doesn't takes up space.

```
#define AND &&
#define OR ||
main()
{
    if (a>b) AND (a>c)
        printf ("A is greater");
}
```

When this code will be executed, this AND will be replaced by && (logical AND).

[Easier for others to understand].

(3) #define could be used to replace a condition.

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

Macros with Arguments (Just as functions).

Eg ↓↓

① # define AREA (x) (3.14 * x * x)

```
main()
{
    float r1 = 6.25, r2 = 2.5, a;
    a = AREA(r1);
    printf("Area of circle = %f", a);
    a = AREA(r2);
    printf("Area of circle = %f", a);
}
```

AREA(r1) is a macro which will be replaced by

3.14 * r1 * r1;

② # define SUM (a, b) (a+b)

```
main()
{
    int a, b, s;
    a = 10; b = 20;
    s = SUM(a, b);
    printf("Sum = %d", s);
}
```

replaced by ⟹ [s = a+b]

③ ~~# define GREATER(a, b)~~

(7)

```
#define GREATEST(a,b,c) ((a>b)?((a>c)?a:c):((b>c)?b:c))

main()
{
int a,b,c,g;
scanf("%d%d%d",&a,&b,&c);
g=GREATEST(a,b,c);
printf("Greatest no=%d",g);
}
```

Q) OP

```
#define SQUARE(n) n*n
main()
{
int j;
j=64/SQUARE(4);
printf("j=%d",j);
}
```

Always use brackets with expansion like (n*n)

OP → j=64 but expected was 4.

64/4*4 ⟹ 64/16=4 ⟹ ✗ Invalid

64/4*4 ⟹ This will be solved first using associativity rule

16*4 = 64

Imp:

Q Difference between Macros and Functions.

Ans (1). In a macro, all templates will be directly replaced by macro expansion, so Time will be less but in function control will go to the function definition and then return back. So time will be more

⇒ Function takes more time than Macros.

(2) Macro takes up more space than Functions

∵ Whenever a macro is called, that code will be copied multiple times each time a macro is called. but in functions definition has written only one time, if function is calling multiple times, control will go to function definition. so space consuming will be less.

⇒ So if Macro is simple and short, we can use macro. otherwise use functions.

Conditional Compilation

We want some statements or lines to be compiled and some we don't want to compile. We can do this by inserting preprocessing commands #ifdef and #endif.

eg
```
#ifdef macroname
    statement1;
    ---- 2;
    ---- 3;
#endif.
```

If macroname will be defined, only then these statements will be compiled.

eg
```
#ifdef test
    st 1;
    st 2;
    st 3;
#endif
    st 4;
    st 5;
```

→ Macro (test) is defined

Then st1, st2, st3 will be executed, and st 4, st5 will be exec

⇒ If #define test will not be there, then only st 4 and st5 will be executed.

Imp:-

Eg 2

```
#define TEST 2
main()
{
#if TEST <=5
        statement 1;
        —— 2;
        —— 3;
#else
        statement 4;
        —— 5;
        —— 6;
#endif
}
```

⟹

Value of TEST is Misc

So statement 1, 2, 3 will be executed and 4, 5 and 6 will be skipped.

If value of test > 5 Then st 4, 5, and 6 will be executed and 1, 2 and 3 will be skipped.

IIly

```
#define TEST
main()
{
#ifdef TEST
        st 1;
        st 2;
        st 3;
#elif
        st 4;
        st 5;
        st 6;
#endif
}
```

⟹ This becomes false ⟹

Here TEST is defined

st 4, st 5 and st 6 will be executed

=

Miscellaneous Directives

1. #undef
2. #pragma

---

(1) #undef

⟹ To cause a defined name to become 'undefined'.

Eg: #define TEST

```
main()
{
  ===
  ===
  ===
}
```

If we want to undefine TEST macro then we can use this directive

```
#undef TEST
main()
{
  ===
}
```

(12) # pragma

This directive is used to turn on and off some features. These are certain pragmas available with Microsoft C compiler

pragma startup and # pragma exit

These directives allow us to specify functions that are called upon program startup (before main()) or program exit (just before the program terminates).

```
eg  void fun1();
    void fun2();
    #pragma startup fun1
    #pragma exit fun2
    main()
    {
      printf("\n Inside main");
    }
    void fun1()
    {
      printf("\n Inside fun1");
    }
    void fun2()
    {
      printf("\n Inside fun2");
    }
```

o/p

```
Inside fun1
Inside main
Inside fun2
```

First all fun1 will be executed then main function Then function 2 on exit