UNIT 11 THE C PREPROCESSOR

Structure

11.0 INTRODUCTION

Theoretically, the "preprocessor" is a translation phase that is applied to the source code before the compiler gets its hands on it. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. C Preprocessor is just a text substitution tool, which filters your source code before it is compiled. The preprocessor more or less provides its own language, which can be a very powerful tool for the programmer. All preprocessor directives or commands begin with the symbol #.

The preprocessor makes programs easier to develop, read and modify. The preprocessor makes C code portable between different machine architectures & customizes the language.

The preprocessor performs textual substitutions on your source code in three ways:

File inclusion: Inserting the contents of another file into your source file, as if you had typed it all in there.
Macro substitution: Replacing instances of one piece of text with another.
Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions. The syntax of the preprocessor is different from the syntax of the rest of C program in several respects. The C preprocessor is not restricted to use with C programs, and programmers who use other languages may also find it useful. However, it is tuned to recognize features of the C language like comments and strings.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- define, declare preprocessor directives;
- discuss various preprocessing directives, for example file inclusion, macro substitution, and conditional compilation; and
- discuss various syntaxes of preprocessor directives and their applications.

11.2 # *define* TO IMPLEMENT CONSTANTS

The preprocessor allows us to customize the language. For example to replace **{** and **}** of C language to *begin* and *end* as block-statement delimiters (as like the case in PASCAL) we can achieve this by writing:

```
# define begin {
# define end   }
```

During compilation all occurrences of *begin* and *end* get replaced by corresponding **{** and **}**. So the subsequent C compilation stage does not know any difference!

#define is used to define constants.

The syntax is as follows:

 # define <literal> <replacement-value>

literal is identifier which is replaced with *replacement-value* in the program.

For Example,

```
#define MAXSIZE 256
#define PI 3.142857
```

The C preprocessor simply searches through the C code before it is compiled and replaces every instance of *MAXSIZE* with 256.

```
# define  FALSE  0
# define  TRUE   !FALSE
```

The literal *TRUE* is substituted by *!FALSE* and *FALSE* is substituted by the value 0 at every occurrence, before compilation of the program. Since the values of the literal are constant throughout the program, they are called as constant.

The syntax of above *# define* can be rewritten as:

 # define <constant-name> <replacement-value>

Let us consider some examples,

```
# define        M             5
# define        SUBJECTS      6
# define        PI            3.142857
# define        COUNTRY       INDIA
```

Note that no semicolon (;) need to be placed as the delimiter at the end of a # define line. This is just one of the ways that the syntax of the preprocessor is different from the rest of C statements (commands). If you unintentionally place the semicolon at the end as below:

```
#define MAXLINE 100;          /* WRONG */
```

and if you declare as shown below in the declaration section,

```
char line[MAXLINE];
```

the preprocessor will expand it to:

```
char line[100;];                    /* WRONG */
```

which gives you the syntax error. This shows that the preprocessor doesn't know much of anything about the syntax of C.

11.3 # *define* TO CREATE FUNCTIONAL MACROS

Macros are inline code, which are substituted at compile time. The definition of a macro is that which accepts an argument when referenced. Let us consider an example as shown below:

Example 11.1

Write a program to find the square of a given number using macro.

```
/* Program to find the square of a number using marco*/
#include <stdio.h>
# define  SQUARE(x)   (x*x)
main()
    {
     int v,y;
     printf("Enter any number to find its square: ");
     scanf("%d", &v);
     y = SQUARE(v);
     printf("\nThe square of %d is %d", v, y);
}
```

OUTPUT

```
Enter any number to find its square: 10
The square of 10 is 100
```

In this case, *v* is equated with *x* in the macro definition of *square*, so the variable *y* is assigned the square of *v*. The brackets in the macro definition of *square* are necessary for correct evaluation. The expansion of the macro becomes:

*y =(v * v);*

Macros can make long, ungainly pieces of code into short words. Macros can also accept parameters and return values. Macros that do so are called *macro functions*. To create a macro, simply define a macro with a parameter that has whatever name you like, such as *my_val*. For example, one macro defined in the standard libraries is "abs", which returns the absolute value of its parameter. Let us define our own version of ***ABS*** as shown below. Note that we are defining it in upper case not only to avoid conflicting with the existing "abs".

```
#define ABS(my_val) ((my_val) < 0) ? -(my_val) : (my_val)
```

#define can also be given arguments which are used in its replacement. The definitions are then called macros. Macros work rather like functions, but with the following minor differences:

- Since macros are implemented as a textual substitution, by this the performance of program improves compared to functions.
- Recursive macros are generally not a good idea.

- Macros don't care about the type of their arguments. Hence macros are a good choice where we want to operate on reals, integers or a mixture of the two. Programmers sometimes call such type flexibility polymorphism.
- Macros are generally fairly small.

Let us look more illustrative examples to understand the *macros* concept.

Example 11.2

Write a program to declare constants and macro functions using *#define*.

```
/* Program to illustrate the macros */
#include <stdio.h>
#include <string.h>
#define STR1        "A macro definition!\n"
#define STR2        "must be all on one line!\n"
#define EXPR1        1+2+3
#define EXPR2        EXPR1+5
#define ABS(x)       (((x) < 0) ? – (x):(x))
#define MAX(p,q)    ((p < q) ? (q):(p))
#define BIGGEST(p,q,r)   (MAX(p, q) < r)?(r):(MAX(p, q))
main()
{
  printf(STR1);
  printf(STR2);
  printf("Largest number among  %d, %d and %d is %d\n",EXPR1, EXPR2, ABS (–3),
                                                        BIGGEST(1,2,3));
}
```

OUTPUT

```
A macro definition
must be all on one line!
Largest number among 6, 11 and 3 is 3
```

The macro STR1 is replaced with "A macro definition \n" in the first *printf()* function. The macro STR2 is replaced with "must be all on one line! \n" in the second *printf* function. The macro EXPR1 is replaced with 1+2+3 in third *printf* statement. The macro EXPR2 is replaced with EXPR1 +5 in fourth *printf* statement. The macro ABS(–3) is replaced with (– 3<0) ? – (– 3) : 3. And evaluation 3 is replaced. The largest among the three numbers is diplayed.

Example 11.3

Write a program to find out square and cube of any given number using macros.

```
/* Program to find  the square and cube of any given number using macro directive */
# include<stdio.h>
# define   sqr(x)          (x  * x)
# define   cub(x)          (sqr(x) *  x)
main()
{
        int  num;
        printf("Enter a number: ");
        scanf("%d", &num);
        printf(" \n Square of  the number is %d", sqr(num));
        printf(" \n Cube of the number is %d\n", cub(num));
}
```

OUTPUT

```
Enter a number: 5
Square of the number is 25
Cube of the number is 125
```

Note: Multi-line macros can be defined by placing a backward slash (\ ) at end of each line except the last line. This feature permits a single macro (i.e. a single identifier) to represent a compound statement.

Example 11.4

Write a macro to display the string COBOL in the following fashion

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

```
/* Program to display the string as given in the problem*/
# include<stdio.h>
# define        LOOP    for(x=0; x<5; x++)              \
                        {       y=x+1;                  \
                                printf("%-5.*s\n", y, string);  }   \
                        for(x=4; x>=0; x--)             \
                        {           y=x+1;              \
                        printf("%-5.*s \n", y, string); }

main()
{
        int x, y;
        static char string[ ] = "COBOL";
        printf("\n");
        LOOP;
}
```

When the above program is executed the reference to macro (loop) is replaced by the set of statements contained within the macro definition.

OUTPUT

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

Recollect that CALL BY VALUE Vs CALL BY REFERENCE given in the previous uint. By CALL BY VALUE, the swapping was not taking place, because the visibility of the variables was restricted to with in the function in the case of local variables. You can resolve this by using a macro. Here is ***swap*** in action when using a macro:

#define swap(x, y) {int tmp = x; x = y; y = tmp; }

Now we have swapping code that works. Why does this work? It is because the CPP just simply replaces text. Wherever swap is called, the CPP will replace the macro call with the macro meaning, (defined text).

Caution in using macros

You should be very careful in using Macros. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules (precedence rules). Here is an example of a macro, which won't work.

```
#define     DOUBLE(n)          n + n
```

Now if we have a statement,

```
z = DOUBLE(p) *  q;
```

This will be expanded to

```
z = p + p * q;
```

And since * has a higher priority than +, the compiler will treat it as.

```
z = p + (p * q);
```

The problem can be solved using a more robust definition of DOUBLE

```
#define     DOUBLE(n)          (n + n)
```

Here, the braces around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.

Check Your Progress 1

1. Write a macro to evaluate the formula *f(x) = x*x + 2*x + 4.*

……..……

……..

2. Define a preprocessor macro *swap(t, x, y)* that will swap two arguments *x* and *y* of a given type *t*.

……

……

3. Define a macro called *AREA*, which will calculate the area of a circle in terms of radius.

……

……

4. Define a macro called *CIRCUMFERENCE*, which will calculate the circumference of a circle in terms of radius.

……

……

5. Define a macro to display multiplication table.

……

……

6. Define a macro to find sum of *n* numbers.

……

……

……..

11.4 READING FROM OTHER FILES USING # *include*

The preprocessor directive *#include* is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions. The syntax is as follows:

#include <filename.h>

or

#include "*filename.h*"

The above instruction causes the contents of the file "filename.h" to be read, parsed, and compiled at that point. The difference between the suing of # and " " is that, where the preprocessor searches for the *filename.h.* For the files enclosed in **< >** (less than and greater than symbols) the search will be done in standard directories (include directory) where the libraries are stored. And in case of files enclosed in " " (double quotes) search will be done in "current directory" or the directory containing the source file. Therefore, " " is normally used for header files you've written, and # is normally used for headers which are provided for you (which someone else has written).

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is ***/usr/include*** for most UNIX systems. And ***c:/tc/include*** for turbo compilers on DOS / WINDOWS based systems.

Use of *#include* for the programmer in multi-file programs, where certain information is required at the beginning of each program file. This can be put into a file by name "globals.h" and included in each program file by the following line:

#include "globals.h"

If we want to make use of inbuilt functions related to input and output operations, no need to write the prototype and definition of the functions. We can simply include the file by writing:

#include <stdio.h>

and call the functions by the function calls. The standard header file *stdio.h* is a collection of function prototype (declarations) and definition related to input and output operations.

The extension "*.h*"', simply stands for "header" and reflects the fact that *#include* directives usually sit at the top (head) of your source files. "*.h*" extension is not compulsory – you can name your own header files anything you wish to, but *.h* is traditional, and is recommended.

Placing common declarations and definitions into header files means that if they always change, they only have to be changed in one place, which is a much more feasible system.

What should you put in header files?

- External declarations of global variables and functions.
- Structure definitions.
- Typedef declarations.

However, there are a few things *not* to put in header files:

- Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multiply defined.
- Function bodies (which are also defining instances), may not be put in header files. Since these headers may end you up with multiple copies of the function and hence "multiply defined" errors. People sometimes put commonly-used functions in header files and then use #include to bring them (once) into each program where they use that function, or use #include to bring together the several source files making up a program, but both of these are not good practice. It's much better to learn how to use your compiler or linker to combine together separately-compiled object files.

11.5 CONDITIONAL SELECTION OF CODE USING # *ifdef*

The preprocessor has a conditional statement similar to C's if-else. It can be used to selectively include statements in a program. The commands for conditional selection are; *#ifdef, #else* and *#endif*.

#ifdef

The syntax is as follows:

```
#ifdef  IDENTIFIER_NAME
{
  statements;
 }
```

This will accept a name as an argument, and returns true if the name has a current definition. The name may be defined using a # *define*, the *-d* option of the compiler, or certain names which are automatically defined by the UNIX environment. If the identifier is defined then the statements below #ifdef will be executed

#else

The syntax is as follows:

```
#else
{
statements;
```

}

#else is optional and ends the block started with *#ifdef*. It is used to create a 2 way optional selection. If the identifier is not defined then the statements below *#else* will be executed.

#endif

Ends the block started by *#ifdef* or *#else*.

Where the *#ifdef* is true, statements between it and a following *#else* or *#endif* are included in the program. Where it is false, and there is a following *#else*, statements between the *#else* and the following *#endif* are included. Let us look into the illustrative example given below to get an idea.

Example 11.5

Define a macro to find maximum of 3 or 2 numbers using #ifdef , #else

```
/* Program to find maximum of 2 numbers using #ifdef*/

#include<stdio.h>
#define TWO
main()
{
int a, b, c;
clrscr();

#ifdef   TWO
  {
  printf("\n Enter two numbers: \n");
  scanf("%d %d", &a,&b);
  if(a>b)
          printf("\n Maximum  of two numbers is %d", a);
    else
          printf("\n Maximum  is of two numbers is %d", b);
  }

#endif
}   /* end of main*/
```

OUTPUT

```
Enter two numbers:
33
22
Maximum  of two numbers is  33
```

Explanation

The above program demonstrate preprocessor derivative #ifdef. By using #ifdef TWO has been defined. The program finds out the maximum of two numbers.

11.5.1 Using #ifdef for Different Computer Types

Conditional selection is rarely performed using *#define* values. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program, which will run on either type.

A simple application using machine dependent values is illustrated below.

```
#include <stdio.h>
main()
{
#ifdef HP
{
    printf("This is a HP system \n");
    ……
    ……. /* code for HP  systems*/
     }
 #endif

#ifdef SUN
{
        printf("This is a SUN system \n");
        …….        /* code for SUN Systems
}
#endif
}
```

If we want the program to run on HP systems, we include the directive
#define HP at the top of the program.

If we want the program to run on SUN systems, we include the directive
#define SUN at the top of the program.

Since all you're using the macro HP or SUN to control the #ifdef, you don't need to give it *any replacement* text at all. *Any* definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.

11.5.2 Using #ifdef to Temporarily Remove Program Statements

#ifdef also provides a useful means of temporarily "blanking out" lines of a program. The lines in the program are preceded by *#ifdef* NEVER and followed by *#endif*. Of course, you should ensure that the name NEVER isn't defined anywhere.

```
#include <stdio.h>
main()
{
…….
#ifdef NEVER
{
    ……
    ……. /* code is skipped */
   #endif
}
```

11.6 OTHER PREPROCESSOR COMMANDS

Other preprocessor commands are:

- **#ifndef** - If this macro is not defined
- **#if** - Test if a compile time condition is true

- **#else** - The alternative for #if. This is part of an #if preprocessor statement and works in the same way with #if that the regular C else does with the regular if.
- **#elif** - enables us to establish an "if…else…if .." sequence for testing multiple conditions.

Example 11.6

```
#if  processor  == intel
#define FILE "intel.h"
#elif  processor  == amd
#define FILE "amd.h"
#if  processor  == motrola
#define FILE "motrola.h"
#endif
#include FILE
```

- # - Stringizing operator, to be used in the definition of macro. This operator allows a formal parameter within macro definition to be converted to a string.

Example 11.7

```
#define multiply (p*q) printf(#pq " = %f", pq)
main()
{
   ……..
   multiply(m*n);
}
```

The preprocessor converts the line *multiply(m*n)* into *printf("m*n" " = %f", m*n);* And then into printf("m*n = %f", m*n);

- Token merge, creates a single token from two adjacent ones within a macro definition.

Example 11.8

```
#define merge(s1,s2)  s1## s2
main()
{
   ……..
   printf("%f", merge(total, sales);
}
```

The preprocessor transforms the statement *merge(total, sales) into printf("%f", totalsales);*

#error - text of error message -- generates an appropriate compiler error message.

Example 11.9

```
#ifdef  OS_MSDOS
#include <msdos.h>
      #elifdef OS_UNIX
          #include ``default.h"
              #else
```

```
        #error Wrong OS!!
#endif
```

 # line

#line number *"string"* – informs the preprocessor that the number is the next number of line of input. *"string"* is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

#pragma

It is used to turn on or off certain features. Pragmas vary from compiler to compiler. Pragmas available with Microsoft C compilers deals with formatting source listing and placing comments in the object file generated by the compiler. Pragmas available with Turbo C compilers allows to write assembly language statements in C program.

A control line of the form

#pragma token-sequence

This causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

Other preprocessor directives are # - Stringizing operator allows a formal parameter within macro definition to be converted to a string. ## - Token merge, creates a single token from two adjacent ones within a macro definition. **#error** - generates an appropriate compiler error message.

Example 11.10

Write a macro to demonstrate #define, #if, #else preprocessor commands.

/* The following code displays 35 to the screen. */

```
#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#if (CHOICE == 100)
        void set_my_int()
         {    my_int = 35;         }
#else
         void set_my_int()
         {
              my_int = 27;
          }
#endif
main ()
{
              set_my_int();
              printf("%d\n", my_int);
}
```

OUTPUT

35

The *my_int* is initialized to zero and *CHOICE* is defined as 100. *#if* derivative checks whether *CHOICE* is equal to 100. Since *CHOICE* is defined as 100, *void set_my_int* is called and *int* is set 35. And the same is displayed on to the screen.

Example 11.11

Write a macro to demonstrate #define, #if, #else preprocessor commands.

```
/*  The following code displays 27 on the screen */

#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#undef CHOICE
#ifdef CHOICE
        void set_my_int()
        {
                my_int = 35;
        }
#else
        void set_my_int()
        {
                my_int = 27;
        }
#endif

main ()
{
        set_my_int();
        printf("%d\n", my_int);
}
```

OUTPUT

27

The *my_int* is initialized to 0 and *CHOICE* is defined as 100. *#undef* is used to undefine CHOICE. *#else* is invoked , *void set_my_int* is called and *int* is set 27. And the same is displayed on to the screen.

11.7 PREDEFINED NAMES DEFINED BY PREPROCESSOR

These are identifiers defined by the preprocessor, and cannot be undefined or redefined. They are:

LINE an integer constant containing the current source line number.

FILE a string containing the name of the file being complied.

DATE a string literal containing the date of compilation, in the form “mm-dd-yyyy”.

TIME a string literal containing the time of compilation, in the form “hh:mm:ss”.

STDC the constant 1. This identifier is defined to be 1 only in the implementations conforming to the ANSI standard.

11.8 MACROS Vs FUNCTIONS

Till now we have discussed about macros. Any computations that can be done on macros can also be done on functions. But there is a difference in implementations and in some cases it will be appropriate to use macros than function and vice versa. We will see the difference between a macro and a function now.

Macros	Functions
Macro calls are replaced with macro expansions (meaning).	In function call, the control is passed to a function definition along with arguments, and definition is processed and value may be returned to call
Macros run programs faster but increase the program size.	Functions make program size smaller and compact.
If macro is called 100 numbers of times, the size of the program will increase.	If function is called 100 numbers of times, the program size will not increase.
It is better to use Macros, when the definition is very small in size.	It is better to use functions, when the definition is bigger in size.

Check Your Progress 2

1. Write an instruction to the preprocessor to include the math library `math.h`.

……

……

2. Write a macro to add user defined header file by name *madan.h* to your program.

……

……

3. What will be the output of the following program?

```
#include<stdio.h>
main()
{
    float  m=7;
  #ifdef DEF
    i*=i;
  #else
    printf("\n%f", m);
  #endif      }
```

……

……

4. Write a macro to find out whether the given character is upper case or not.

……

……

……..

11.9 SUMMARY

The preprocessor makes programs easier to develop and modify. The preprocessor makes C code more portable between different machine architectures and customize the language. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. All preprocessor lines begin with #. C Preprocessor is just a text substitution tool on your source code in three ways: File inclusion, Macro substitution, and Conditional compilation.

File inclusion - inserts the contents of another file into your source file.
Macro Substitution - replaces instances of one piece of text with another.
Conditional Compilation - arranges source code depending on various circumstances.

11.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
# include<stdio.h>
# define f(x)        x*x + 2 * x + 4
main()
{
int num;
printf("enter value x: ");
scanf("%d",&num);
printf("\nvalue of f(num) is %d", f(num));
}
```

2.
```
# include<stdio.h>
# define swap(t, x, y)  { t  tmp = x; x = y; y = tmp; }
main( )
{
 int      a, b;
 float   p, q;
printf("enter integer values for a, b: ");
scanf("%d %d", &a, &b);
printf("\n Enter float values for p, q: ");
scanf("%f %f", &p, &q);
swap(int, a, b);
printf(" \n After swap the values of  a and b are %d %d", a, b);
swap(float, p, q);
printf("\n After swap the values of p and q are %f %f", p, q);
}
```

3.
```
# include<stdio.h>
# define   AREA(radius)      3.1415 * radius * radius
main( )
{
     int radius;
     printf("Enter value of radius:  ");
     scanf("%d ", &radius );
     printf("\nArea is  %d", AREA(radius));
}
```

4.
```
# include<stdio.h>
# define   CIRCUMFERENCE(radius)        2 * 3.1415 * radius

 main( )
```

```
{
    int radius;
    printf("Enter value for radius ");
    scanf("%d ", &radius );
    printf("Circumference is  %d", CIRCUMFERENCE(radius));
}
```

5.
```
# include<stdio.h>
# define MUL_TABLE(num)              for(n=1;n<=10;n++)   \
                                  printf("\n%d*%d=%d",num,n,num*n)
 main()
{
    int number;      int n;
    printf("enter number");
    scanf("%d",&number);
    MUL_TABLE(number);
 }
```

6.
```
# include<stdio.h>
# define SUM(n)  ( (n * (n+1)) / 2 )
main()
{
    int number;
    printf("enter number");
    scanf("%d", &number);
    printf("\n sum of n numbers %d", sum(number));       }
```

Check Your Progress 2

1.
```
# include <math.h>
```

2.
```
#include "madan.h"
```

3. 7

4.
```
#  include<stdio.h>
# define isupper(c) (c>=65 && c<=90)
main()
{
  char c;
  printf("Enter character:");
  scanf("%c",&c);
  if(isupper(c))
        printf("\nUpper case");
    else
         printf("\nNo it is not an upper case character");
}
```

11.11 FURTHER READINGS

1. The C programming language, *Brian W. Kernighan & Dennis Ritchie*, Pearson Education, 2002.
2. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education, 2002.
3. Computer Programming in C, *Raja Raman. V*, PHI, 2002.