MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 3: Advanced C Programming Concepts

Unit 11: Preprocessor Directives and Macros

Structure	
11.0	Introduction
11.1	Objectives
11.2	'C' Preprocessor
11.3	# define to Implement Constants
11.4	# define to Create Functional Macros
11.5	Pre-Processor Operators
11.6	Reading from other Files using #include
11.7	Conditional Selection of Code using #ifdef
11.8	Other Pre-Processor Commands
11.9	Pre-defined Names defined by Pre-Processor
11.10	Macros Vs Functions
11.11	Conclusion
11.12	Questions
11.13	References

11.0 Introduction

This unit discusses 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.

This unit will be discussing the separate step compilation process of language C.

11.1 Objectives

After completing 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 'C' Preprocessor

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 preprocessing before the actual compilation. We'll refer to the C Preprocessor as CPP.

11.2.1 # Description of Pre-Processors

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:

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

11.2.2 Pre-Processors Examples

Analyze the following examples to understand various directives.

#define MAX_ARRAY_LENGTH 20

This directive tells the CPP to replace instances of **MAX_ARRAY_LENGTH** with **20**. Use **#define** for constants to increase readability.

#include <stdio.h>

#include "myheader.h"

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

#undef FILE_SIZE

#define FILE_SIZE 42

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

#ifndef MESSAGE

#define MESSAGE "You wish!"

#endif

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

#ifdef DEBUG

/* Your debugging statements here */

#endif

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off onthe-fly during compilation.

11.3 # 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:

 # define <literal> <replacement-value>

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

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.

 # define can be rewritten as:

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

Let us consider few 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 */

11.4 # define to Create Functional Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

```
int square(int x)
{
return x * x;
}
```

We can rewrite the above code using a macro as follows:

#define square(x) ((x) * (x))

Macros with arguments must be defined using the #define directive before they can be used.

The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis.

Example-1:

```
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(void)
{
printf("Max between 20 and 10 is %d\n", MAX(10, 20));
return 0;
}
```

The following result:

Max between 20 and 10 is 20.

Example-2:

```
/* 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);
}
```

The following result:

Enter any number to find its square: 10

The square of 10 is 100.

11.4.1 Caution is using Macros

It 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.

11.5 Pre-Processor Operators

The C preprocessor offers the following operators to help create macros:

11.5.1 The Macro Continuation (\)Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line.

Example:

```
#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
```

11.5.2 The Stringize (#)Operator

The stringize or number-sign operator (#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list.

Example:

```
#include #define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
int main(void)
{
message_for(Carole, Debra);
return 0;
}
```

The following result:

```
Carole and Debra: We love you!
```

11.5.3 The Token Passing (##)Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

Example:

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void)
{
```

```
int token34 = 40;
tokenpaster(34);
return 0;
}
```

The following result:

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

11.5.4 The Defined() Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero).

The defined operator is:

```
#include <stdio.h>
#if !defined (MESSAGE)
#define MESSAGE "Well-done!"
#endif
int main(void)
{
printf("Here is the message: %s\n", MESSAGE);
return 0;
}
```

The following result:

```
Here is the message: Well-done!
```

11.6 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:

```
#include <filename.h>
```

or

```
#include “filename.h”
```

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>

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

11.7 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:

```
#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:

```
#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.

Example:

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*/
```

The following result:

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

11.7.1 Using #ifdef for different computer type

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.

```
#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
}
```

11.7.2 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
}
```

11.8 Other Pre-Processor 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.

 # 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.

11.9 Pre-defined Names defined by Pre-Processor

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
DATE	The current date as a character literal in "MMM DD YYYY" format.
TIME	The current time as a character literal in "HH:MM:SS" format.
FILE	This contains the current filename as a string literal.
LINE	This contains the current line number as a decimal constant.
STDC	Defined as 1 when the compiler complies with the ANSI standard.

Example:

```
#include <stdio.h>
main()
{
printf("File :%s\n", __FILE__ );
printf("Date :%s\n", __DATE__ );
printf("Time :%s\n", __TIME__ );
printf("Line :%d\n", __LINE__ );
printf("ANSI :%d\n", __STDC__ );
}
```

When the file test.c is compiled and executed, it produces the following result:

```
File :test.c
Date :Dec 04 2023
Time :13:26:14
Line :8 ANSI :1
```

11.10 Macros Vs Functions

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.

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.

11.11 Conclusion

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.

11.12 Questions

1. **What is a preprocessor directive?**

Ans.1 A preprocessor directive is an instruction that is processed by the preprocessor before the actual compilation of the code begins. These directives begin with the # symbol and perform various tasks such as including files, defining constants, and conditional compilation.

2. **What are macros in C?**

Ans.2 Macros are fragments of code that are given a name. Whenever the name is used in the code, it is replaced by the content of the macro. They are defined using the **#define** directive.

3. **What is the purpose of the #include directive?**

Ans.3 The **#include** directive is used to include the contents of a file or library into the current file. This is typically used to include header files that contain function declarations and macro definitions.

4. **Explain the difference between #include <file> and #include "file".**

Ans.4 **#include <file>** is used to include standard library files and searches the compiler's standard library directories. **#include "file"** is used for including user-defined files and searches the current directory first before searching the standard library directories.

5. **What are the potential pitfalls of using macros?**

Ans.5 **Potential pitfalls include:**

- Lack of type checking.
- Unexpected side effects due to macro expansion.
- Difficulties in debugging.

6. **What is the difference between macros and inline functions?**

Ans.6 Macros are expanded by the preprocessor and do not have type checking, whereas inline functions are type-checked and provide better debugging support. Inline functions are also safer and can be optimized by the compiler.

7. **Can you give an example of a multiline macro and explain how it works?**

Ans.7 A multiline macro uses the \ character to continue the macro definition onto the next line. For example:

```
#define PRINT_DEBUG(msg) \
 do { \
printf("Debug: %s\n", msg); \
}
while (0)
```

This macro defines a block of code that prints a debug message.

11.13 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***