PROGRAMMING IN C

Chapter Notes

Chapter: Pre-processor Directives

1. Introduction

In the C programming language, a pre-processor is a program that processes the source code before it is handed over to the actual compiler. It is the very first stage of the compilation process, and it works purely on a textual level — it does not understand C syntax or semantics in the way the compiler does; it simply scans the source file and carries out the instructions given to it through pre-processor directives.

A pre-processor directive is a line in the source code that begins with the hash (#) symbol. Because a directive is an instruction to the pre-processor and not an executable C statement, it does not end with a semicolon, and it must normally begin at the start of a line (leading whitespace is allowed).

Pre-processor directives are commonly used to:

- Substitute symbolic names with constant values or code fragments (macro expansion).
- Include the contents of header files into the source file.
- Compile or skip parts of the code conditionally, depending on certain conditions.
- Control line numbering and generate custom compiler errors or warnings.

2. Types of Pre-processor Directives

C pre-processor directives can be broadly classified into the following categories:

- Macro Definition Directives — #define and #undef, used to define and remove macros.
- File Inclusion Directive — #include, used to insert the contents of another file (usually a header file) into the source file.
- Conditional Compilation Directives — #if, #ifdef, #ifndef, #elif, #else and #endif, used to compile selected parts of the code based on certain conditions.
- Line Control Directive — #line, used to change the compiler's internal line number and file name for the purpose of error reporting.
- Error Directive — #error, used to deliberately generate a compiler error with a custom message.
- Pragma Directive — #pragma, used to give special, implementation-defined instructions to the compiler.

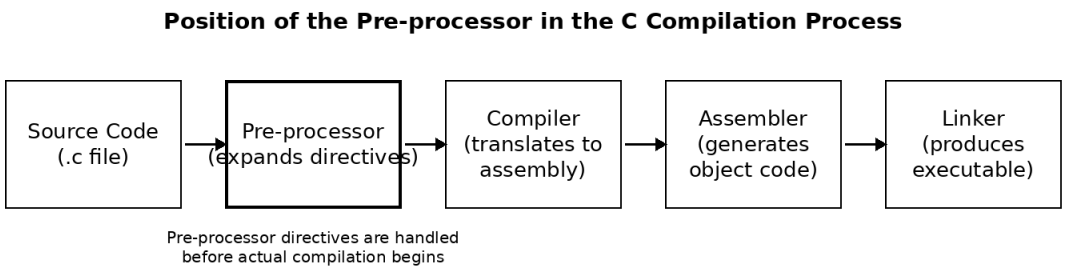

Fig. 1 — Position of the pre-processor in the C compilation process

3. Macros

A macro is a fragment of code that is given a name. Wherever that name is used in the program, the pre-processor replaces it with the actual defined code or value. This process is known as macro expansion, and it takes place before compilation, so the compiler only ever sees the expanded code.

Macros are defined using the #define directive and are broadly of two kinds:

- Object-like macros — simple name-to-value substitutions, generally used to define symbolic constants.

```
#define PI 3.14159
#define MAX_SIZE 100
```

- Function-like macros — macros that accept arguments, and resemble a function call in appearance, but are still expanded purely as text.

```
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

When the pre-processor encounters SQUARE(5) in the source code, it textually replaces it with ((5) * (5)) before the compiler ever processes the statement.

4. Rules for Using Macros

Since macro substitution is a purely textual operation performed before compilation, a number of conventions and precautions should be followed while defining macros:

- A macro definition does not end with a semicolon; if one is added, it becomes part of the replacement text.
- Macro names are conventionally written in uppercase letters to distinguish them from ordinary variable names.
- The entire macro expression, as well as every parameter used within it, should be enclosed in parentheses to avoid errors caused by operator precedence during expansion.
- There should be no space between the macro name and the opening parenthesis when defining a function-like macro; a space would cause the pre-processor to treat it as an object-like macro.

- Arguments that contain side effects, such as i++ or i--, should be avoided in macro calls, because the argument may be evaluated more than once during expansion.
- A macro can be removed using the #undef directive if it is no longer required or needs to be redefined.
- Macro definitions can refer to other macros, but the macros being referred to must already be defined earlier in the file.
- Since macros are not type-checked, care should be taken to pass arguments of the correct and intended type.

5. Distinction Between Functions and Macros

Although macros with arguments look similar to function calls, they behave very differently internally. The table below summarises the key differences:

Basis of Comparison	Macro	Function
Processing	Expanded by the pre-processor before actual compilation begins	Compiled once and called at run time
Execution speed	Generally faster — no function-call overhead	Slightly slower due to call and return overhead
Type checking	No type checking is performed (pure text substitution)	Arguments and return type are type-checked by the compiler
Code size	Increases, since code is expanded at every point of use	Remains compact — only one copy of the code exists
Memory usage	No separate memory or stack frame is used	Requires stack memory for parameters, return address, local variables
Argument evaluation	An argument may be evaluated multiple times if it appears more than once	Each argument is evaluated exactly once
Debugging	Harder to debug and trace, since expansion happens before compilation	Easier to debug using standard tools and breakpoints
Recursion	Cannot be used recursively	Can be called recursively
Keyword used	Defined using #define	Defined using a return type, name and parameter list

Multiple Choice Questions (MCQs)

Answer the following questions by choosing the most appropriate option. The answer key is provided at the end of this section.

1. Which symbol is used to indicate a pre-processor directive in C?

A. $
B. #
C. &
D. @

2. Pre-processor directives are processed:

A. After compilation
B. During linking
C. Before compilation begins
D. At program run time

3. Which directive is used to include a header file in a C program?

A. #include
B. #insert
C. #import
D. #header

4. Which of the following is used to define a macro in C?

A. #macro
B. #define
C. #let
D. #const

5. Which directive removes a previously defined macro?

A. #remove
B. #delete
C. #undef
D. #unset

6. A C pre-processor directive statement ends with:

A. A semicolon (;)
B. A colon (:)
C. No terminator (newline ends it)
D. A full stop (.)

7. Which of these is an example of conditional compilation directive?

A. #ifdef
B. #define
C. #include
D. #pragma

8. Which directive is used to suppress compiler warnings for a specific section of code (compiler-specific)?

A. #warning
B. #pragma

C. #error

D. #line

9. A macro defined as #define SQUARE(x) x*x may give an incorrect result for the call SQUARE(a+b) because:

A. Macros cannot take arguments

B. Of missing parentheses causing wrong operator precedence

C. Macros are type-checked at compile time

D. The macro name is too long

10. Which of the following best describes an object-like macro?

A. A macro that takes arguments like a function

B. A macro that is simply replaced by a constant value or code fragment

C. A macro used only inside classes

D. A macro that returns a value at run time

11. The #include directive using angle brackets < > is generally used for:

A. User-defined header files in the current directory

B. Standard/system header files

C. Only source files

D. Object files

12. Macro expansion in C is performed by:

A. The linker

B. The pre-processor

C. The operating system

D. The assembler

13. Which statement about macros and type checking is correct?

A. Macros are strongly type-checked by the compiler

B. Macros undergo no type checking since they are simple text substitutions

C. Macros are checked only at run time

D. Macros can only accept integer arguments

14. Repeated evaluation of an argument with side effects (e.g., i++) inside a macro can cause:

A. A compilation error

B. Unexpected or unintended results

C. Faster program execution

D. No difference compared to a function call

15. Compared to a function call, a macro call generally results in:

A. Slower execution due to call overhead

B. Larger code size due to inline expansion at every occurrence

C. A separate stack frame being created

D. Run-time type checking of arguments

Answer Key

Q. No.	Answer	Q. No.	Answer	Q. No.	Answer
1	**B**	6	**C**	11	**B**
2	**C**	7	**A**	12	**B**
3	**A**	8	**B**	13	**B**
4	**B**	9	**B**	14	**B**
5	**C**	10	**B**	15	**B**

Broad / Descriptive Questions

Answer the following questions in detail, with suitable examples and diagrams wherever applicable.

1. What is a pre-processor in C? Explain its role in the overall compilation process with the help of a diagram.
2. Discuss, with suitable examples, the different types of pre-processor directives available in C.
3. What is a macro? Distinguish between object-like macros and function-like macros with examples.
4. Explain macro expansion with an example. How does the pre-processor handle nested macros?
5. State and explain the important rules that should be followed while defining and using macros in C.
6. Why is it recommended to enclose macro arguments and the entire macro expression in parentheses? Illustrate with an example where omitting parentheses leads to an incorrect result.
7. Differentiate between macros and functions in C on the basis of processing, speed, type checking, and memory usage.
8. What is conditional compilation? Explain the use of #ifdef, #ifndef, #else and #endif with a suitable example.
9. Explain the purpose of the #include directive. What is the difference between using angle brackets (< >) and double quotes (" ") with #include?
10. What do you understand by the term 'side effects' in the context of macro arguments? Explain with an example why macros with side-effect arguments can produce unexpected results.