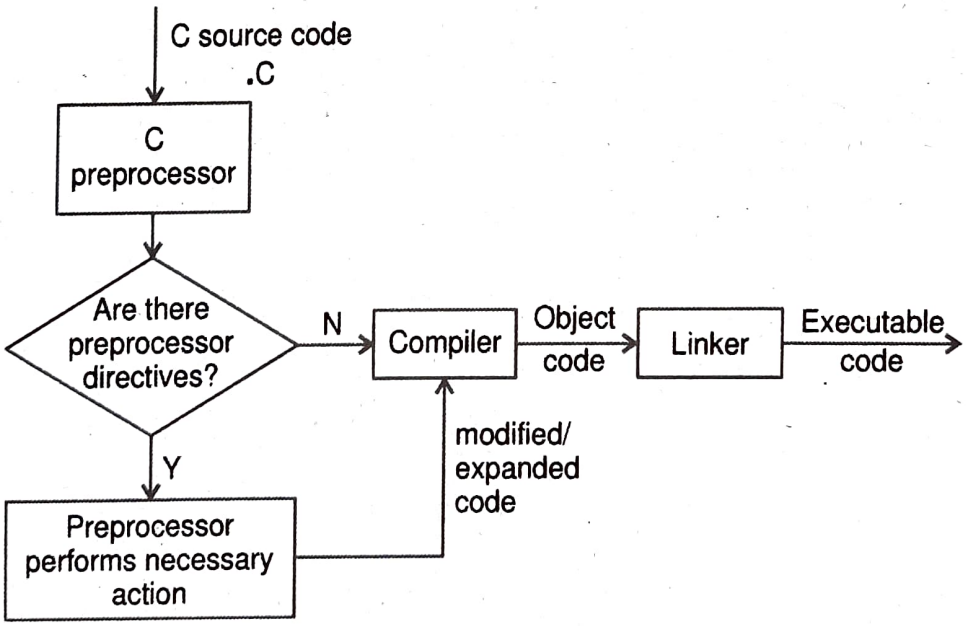
CONTENTS

Unit 1

Preprocessor

1. Concept

The 'C' compiler converts the source code to machine code. Before the program is given to the compiler, it is preprocessed. A Preprocessor is a program that processes or analyzes the source code file before it is given to the compiler. It performs some action according to special instructions called Preprocessor directives (which begin with #) in the source code. The preprocessor operation is shown in the following *figure*.



It performs the following tasks:

1. Processes directives and expands macros.
2. Removes comments, replacing them by a single space.
3. Divides the program into a stream of tokens.
4. Joins any lines that end with a backslash character into a single line.
5. Replaces escape sequences by their equivalent internal representation.
6. Concatenates adjacent constant character strings.
7. Replaces trigraph sequences by their equivalents. Trigraph sequences are used to handle non ASCII characters sets.

2. Format of Preprocessor Directive

Preprocessor directives are special instructions for the preprocessor.

A **preprocess or directive** begins with # and does not end with semicolon (;).

1. They begin with a '#' which must be the first non-space character on the line.
2. They do not end with a semicolon.
3. Except for some #pragma directives, preprocessor directives can appear anywhere in a program.
4. Some directives require arguments and must be separated by a space after the directive name.

Format:

```
#directive-name arguments
```

The directive-name is a predefined word. There are 11 proprocessor directives in 'C'. These are – include, define, undef, ifdef, ifndef, if, else, elif, endif, error, pragma

Example

```
#include<stdio.h>
```


3. File Inclusion Directive

File inclusion directive is **#include**

The file inclusion directive is **#include**. This directive instructs the compiler to include contents of the specified file, i.e., it inserts the entire contents of the file at that position.

Syntax

```
#include <filename>
        OR
#include "filename"
```

- In the first format, the filename is searched in standard directories only.
- In the second, the file is first searched in the current directory. If it is not found there, the search continues in the standard directories.
- Any external file-containing user defined functions, macro definitions, etc. can be included.
- An included file can include other files.

Examples

1.
```
#include<stdio.h>
#include<string.h>
```

2.
```
/* group.h */
#include<stdio.h>
#include<math.h>
#include"myfile.c"
```

3.
```
/* mainprog.c      */
#include"group.h"
main()
{
  -
  -
}
```

4. Macro Substitution Directive

Macro substitution directive is **#define**

The macro substitution directive is **#define**. It defines a symbol which represents a value. Wherever the macro name occurs in a program the preprocessor substitutes the code of the macro at that position.

It defines symbolic constants which occur frequently in a program.

Syntax

```
#define  macro_name  value
```

1

Oct. 2017 – 4M
Explain macro substitution in brief with example.

Examples

i. #define PI 3.142
ii. #define TRUE 1
iii. #define AND &&
iv. #define LESSTHAN <
v. #define GREET printf("Hello");
vi. #define MESSAGE "Welcome to C"
vii. #define INRANGE (a >= 60 && a<70)

Example

```
int a=65;
if(INRANGE)
printf("First class");
```

Parameterized Macro

Just like a function, a macro can be defined with parameters. Such a macro is also called a **function macro** because it looks like a function. It represents a small executable code. Each time the macro name is encountered, its arguments are replaced by the actual arguments which are used in the macro call.

Example

```
#define HALFOF(x)    x/2
#define SQR(x)       x*x
#define MAX(x,y)     ((x)>(y)?(x):(y))
```

Wherever the macro occurs in the program, its argument(example x) is replaced by the variable or value used in the program.

For example:

```
result = HALFOF(10);   //replaced by result = 10/2;
ans = SQR(a);          //replaced by ans = a*a;
```

It is advisable to enclose the arguments in (). If the arguments are not enclosed in (), it may yield wrong results.

Example

```
result=HALFOF(10+6);
```

This will be evaluated as result=(10+6/2); and will give incorrect results. The correct way to define the macro would be:

```
#define HALFOF(x)   ((x)/2)
```

Now, result = ((10+6)/2); which will give the correct result.

Nested Macro

A macro can be contained within another macro. This is called nesting of macros.

Example

```
#define CUBE(x)  SQR(x)*(x)
```

In this *example*, macro SQR is called in macro named CUBE.

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

Note We cannot define two macro's with the same name unless their value is the same.

Macros Versus Functions

[Apr. 2018 – 4M]

No.	Macro	Function
1.	A macro is small in size. It usually does contain more than one line.	A function can contain several lines of code.
2.	A macro is preprocessed.	A function is compiled.
3.	A macro is expanded in the code, i.e., its code replaces the macro name.	Function code is not inserted into the program when a function is called.
4.	The program size increases.	The program size does not increase.
5.	Execution of macros is faster since the code is replaced.	Functions execute slower since control has to be passed to the function and then returned back to the program.
6.	No type checking is done for macros. Hence there can be side-effects or unforeseen errors.	Strict type checking is done for functions.
7.	Usually used for small code which appear many times.	Usually used for large code that needs to be executed many times.

5. Other Directives

#error

The `#error` directive causes the preprocessor to report a fatal error. The rest of the line that follows `#error` is used as the error message. This directive is useful for notifying the users if there is some program inconsistency or the violation of a constraint.

Syntax

```
#error  error-string
```

Examples

```
#if SIZE%2==0
#error SIZE should not be an even number
#endif
```

#pragma

This directive is used to specify options to the compiler. These options are specific for the platform and the compiler.

1 Apr. 2018 – 1M
What is # pragma directive?

It tells the compiler to turn on or off some features. A compiler provides its own #pragma directives.

Syntax

```
#pragma  directive
```

Some pragma directives are:

1. `#pragma GCC dependency`

   Checks the dates of current and other file. If other file is recent, it shows a warning message.

   Example

   ```
   #pragma GCC dependency "parse.y"
   ```

2. `#pragma GCC poison`

   Used to block an identifier from the program.

Example:

```
#pragma GCC poison printf scanf
```

1. `#pragma GCC system_header`

   It treats the code of current file as if it came from system header.

6. Conditional Compilation

These directives allow specific parts of the program to be included or discarded based on a certain condition. Six directives are available to control conditional compilation. These are #if, #ifdef, #ifndef, #else, #elif and #endif.

The program text within the blocks may consist of preprocessor directives, C statements, and so on. The beginning of the block consists of one of the following three directives:

- `#if`
- `#ifdef`
- `#ifndef`

Optionally, an alternative block of text can be indicated by the following two directives:

- `#else`
- `#elif`

The end of the block or alternative block is marked by the `#endif` directive.

If the condition given in `#if, #ifdef, or #ifndef` is true (nonzero), then all lines between the matching `#else (or #elif )` and an `#endif` directive, if present, are ignored. If the condition is false (0), then the lines between the `#if, #ifdef,` or `#ifndef` and an `#else, #elif,` or `#endif` directives are ignored. These directives can be nested.

#if

The `#if` directive has the following syntax:

```
#if constant-expression
```

This directive checks whether the *constant-expression* is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++), decrement (- -), `sizeof`, pointer (*), address (&), and cast operators.

1

Oct. 2017 – 1M
Give the use of #ifdef with example.

#ifdef

Syntax

```
#ifdef identifier
```

This directive checks whether the identifier is currently defined (using `#define` directive).

#ifndef

Syntax

```
#ifndef identifier
```

This directive checks if the identifier is not currently defined.

#else

Syntax

```
#else
```

This directive is used with `#if, #ifdef,` or `#ifndef` to delimit alternative source text to be compiled if the condition is false. It is optional.

#elif

Syntax

```
#elif constant-expression
```

The `#elif` directive performs a task similar to the combined use of the `else-if` statements in C.

This directive is used with `#if, #ifdef, #ifndef,` or another `#elif` to indicate alternative code if the condition is false. It is optional. The difference between #else and #elif is that #elif must be followed by a condition. Also, `#elif` can have the following `#elif` or `#else` but `#else` cannot be followed by an `#else` or `#elif`.

#endif

Syntax

```
#endif newline
```

This directive ends the scope of the `#if, #ifdef, #ifndef , #else,` or `#elif` directive.

Note

i. Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed.

ii. The **#else** directive, if present, must be the last directive before **#endif**.

iii. The **#if, #elif, #else,** and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else, #elif,** or **#endif** directive belongs to the closest preceding **#if** directive.

iv. All conditional-compilation directives, such as **#if** and **#ifdef,** must be matched with closing **#endif** directives prior to the end of file.

Examples

Check if macro NUM has an even value and display appropriate message.

```
#if(NUM%2==0)
  printf("NUM is even");
#endif
```

Check if macro NUM has an even or odd value and display appropriate message.

```
#if(NUM%2==0)
printf("NUM is even");
#else
printf("NUM is odd");
#endif
```

Check if macro NUM has 0, positive or negative value and display appropriate message.

```
#if(NUM == 0)
        printf("\nNumber is Zero");
#elif(NUM > 0)
        printf("\nNumber is Positive");
#else
        printf("\nNumber is Negative");
#endif
```

If macro NUM is defined, then define macro MAX having value 20.

```
#ifdef NUM
      #define MAX 20
#endif
```

If macro NUM is not defined, then define macro NUM having value 20.

```
#ifndef NUM
  #define NUM 20
#endif
```

If macro NUM is defined, then redefine it with value 75. If it is not defined, define it with value 100.

```
#ifdef NUM
  #undef NUM
  #define NUM 75
#else
  #define NUM 100
#endif
```

1

Apr. 2018 – *4M*
Explain any three predefined macros.

7. Predefined Macros

There are some predefined macros in the 'C' language. They all begin and end with two underscore characters. Some of the commonly used macros are described in the table below.

Macro	Value
__LINE__	Integer value representing the current line in the source code file.
__FILE__	A string literal containing a C string constant which is the name of the source file being compiled.
__DATE__	A string literal in the form "mmm dd yyyy" containing the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Jan 20 2017".
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__STDC__	This macro expands to the constant 1, to indicate that this is ANSI Standard C and expands to 0 otherwise.

Example

```
#include<stdio.h>
#ifdef __STDC__
#define CONFORM "conforms"
#else
#define CONFORM "does not conform"
```



```
#endif
int main()
{
  printf("Line %d of file %s has been executed\n", __LINE__, __FILE__);
  printf("This file was compiled at %s on %s\n", __TIME__, __DATE__);
  printf("This program %s to ANSI standards\n", CONFORM);
}
```

Output
Line 9 of file p1.c and function main has been executed
This file was compiled at 16:48:31 on Nov 2 2016
This program conforms to ANSI standards

Solved Programs

1. What will be the output of the following code?

```
#define SQR(x) x*x
void main()
{
   int a=2, b=3, ans;
   ans = SQR(a+b);
   printf("ans = %d", ans);
}
```

Solution

SQR(a+b) will be replaced by a+b*a+b

= 2+3*2+3.

Here, 3*2 will be evaluated first

= 2+6+3

= 8+3 = 11

Output
ans = 11

2. What will be the output of the following code?

```
#define MAX(x,y) x>y?x:y
#define MAXTHREE(x,y,z)  MAX(MAX(x,y),z)
void main()
{
  printf("%d",MAXTHREE(6,2,10));
}
```

Solution

MAXTHREE(x,y,z) will be expanded as x>y?x:y>z?x>y?x:y:z

Output
ans = 6

MAXTHREE(6,2,10) is expanded as 6>2?6:2>10?6>2?6:2:10 which results in 6 because conditional operator has right to left associativity. Hence 6>2?6:2 will be evaluated first resulting in 6.

Next, 2>10?6:10 will be evaluated resulting in 10. Next 6>2?6:10 is evaluated which results in 6.

The answer would be 10 if the macro is defined as:

#define MAX(x,y) ((x)>(y)?(x):(y))

3. Define a macro EQUALSTR which accepts two strings and gives 1 if they are equal and 0 otherwise.

```
#define EQUALSTR(s1,s2)
strcmp(s1,s2)?0:1
void main()
{
  char str1[10], str2[10];
  printf("Enter the two strings :");
  gets(str1);
  gets(str2);
  if(EQUALSTR(str1, str2))
    printf("Strings are equal");
  else
    printf("Strings are not equal");
  getch();
}
```

4. Give a macro definition

#define SQUARE(X) (x)*(x)

What will be the output of following statement:

printf("%d", SQUARE(4+4));

Solution

Output of the given statement is:

Output
64

64

8 (4+4) will be passed as argument to the macro SQUARE and then (8)*(8) will be performed.

5. Write a program in 'C' for finding largest of 2 numbers using a macro.

```
#include<stdio.h>
#define large(a, b) ((a)>(b) ?(a): (b))
main()
{
  int x,y;
  printf("\nEnter two numbers:");
  scanf("%d%d",&x,&y);
  printf("largest=%d",large(x,y));
}
```

Exercises

A. Predict the output

1.
```
#define GREAT "xyz"
main()
{
  printf(GREAT);
}
```

2.
```
#define GREET HELLO
main()
{
printf(GREET);
}
```

3.
```
main()
{
  #include<stdio.h>
}
```

4.
```
#define MAIN main()
#define BEGIN
{
  #define END
}
#define GREET printf("Hello")
MAIN
BEGIN
  GREET;
END
```

5.
```
#define SQUARE(x) (x*x)
main()
{
int i=20, j=10,k;
  k=SQUARE(i-j)
  printf("%d",k);
}
```

6.
```
#define SQUARE(x) (x)*(x)
main()
{
  int i=20,j=10,k;
  k=SQUARE(i-j);
  printf("%d",k);
}
```

7.
```
/*File abc.h */
printf("Hello");
/*File my.c */
main()
{
  #include "abc.h"
  printf("C");
}
```

8.
```
/*File xxx.h */
printf("Hello")
/*File my.c */
main()
{
  #include "xxx.h"
  ;
  printf("C");
}
```

B. Review Questions

1. Write a note on the C Preprocessor.
2. Explain Macro substitution in brief with examples.
3. When a parameterized macro is defined, why should each argument be enclosed in parenthesis?
4. Can one macro be used inside another macro?

5. List the differences between macros and functions.
6. What are the two ways to include a file? What is the difference between the two methods?
7. List the conditional compilation directives.
8. What is the difference between #else and #elif?
9. With suitable examples, illustrate the use of #ifdef and #ifndef.
10. Explain the predefined macros in C.
11. List the preprocessor operators in 'C'.
12. State the purpose of #error directive.
13. What is the purpose of pragma?
14. Give the format of a preprocessor directive.

Questions Asked in Previous Exams

1 Mark

A. Choose correct option

1. _____ symbol is used to terminate a string. **[Apr. 2018]**

   i. Null. **ii. \0**

   iii. . iv. /0

2. Which of the following is macro continuation preprocessor operator. **[Apr. 2018]**

   i. # ii. ##

   iii. / **iv. **

3. Which of the following is compiler control directive?

   i. #ifdef ii. #define

   iii. #include iv. All of the above

4. What will be the output of the program? [Oct. 201

```
#include<stdio.h>
#define square (x) x *x
void main()
{
   int i;
  i = 64/square(4);
   printf("%d",i);
}
```

i. 4 ii. **64**

iii. 16 iv. None of the above

B. Answer the following question

1. What is # pragma directive?
2. Give the use of # ifdef with example. [Apr. 2018] [Oct. 2017]

4 Marks

1. Explain any three predefined macros.
2. Compare Macro and Function. [Apr. 2018]
3. Explain different types of preprocessor directives in C with example. [Apr. 2018]
4. Explain macro substitution in brief with example. [Oct. 2017] [Oct. 2017]