Pre-Processor Directives:

Pre-Processor Directives are not the instructions to the microprocessor but the instructions to the compiler, without using this, you are asking compiler to translate code into machine language instructions that can be executed by microprocessor, on the other hand preprocessor directive operated on directly by the compiler before the compilation process even begins. This concept almost be considered as language with in the C/C++ language that does not exist in many other high level language. The directives define anywhere in program but are most often used at the beginning of a file of before main () or before the beginning of a particular function. Pre-processor directive mast be start # symbol. Some list of pre-processor directive.

#define	#include	#undef	#if	#endif
#ifdef	#ifndef	#error	#pragma	#elif

The most common preprocessor directives are

1) #include directive
2) #define directive
3) #undef directive

1) **#include directive:** The include directive is used to include files like as we include header files in the beginning of the program using ***#include*** directive like

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

2) **#define directive:** It is used to assign names to different constants or statements which are to be used repeatedly in a program. These defined values or statement can be used by main or in the user defined functions as well. They are used for

a. defining a constant
b. defining a statement
c. defining a mathematical expression

Example:

```
#define PI 3.142
#define AreaOfCircle(r) (3.142 * (r *r) )
#define PR printf("enter a new number ");
#define CUBE(a) (a*a*a)
```

Why use #define directives instead of variable name?

- Compiler can generate faster & more compact code for constant than it can for variable.
- If something never changes it is confusing to make it a variable
- #define variables cannot be altered whereas variables can be accidently altered in the program.

Constant modifier:

The new ANSI standard C/C++ defines a modifier, ***const***, that can be used to achieve almost the same effect as ***#define*** when creating constants.

```
const data-type constant_name_identifier = value ;
```

It is similar to a normal function declaration however, the ***const*** tells the compiler not to permit any changes to the variable.

Macro:

The ***#define*** directive has another ability to use of arguments, called macro.

There are two types of macros:

1. Object-like Macros
2. Function-like Macros

1. Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric and string constants. For example:

```
#define PI 3.142
#define PR printf("Enter  a new number ");
```

```
#include <stdio.h>
#define PI 3.1415
#define PR printf("Enter  a new number ");

main()
{
  printf("%f",PI);
  PR;
}
```

2. **Function-like Macros**

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
#define AreaOfCircle(r) (3.142 * (r *r) )
#define CUBE(a) (a*a*a)
```

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
 {
  printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
 }
```

Macro vs Functions:

Macros increases code multiple copies of same code are substituted in the program whereas function transfer control to the same copy again & again. A function is a piece of code that can relatively independently be executed and performs a specific task. Functions executes relatively slower than macros as they transfer control to the function as it is called.

Advantages of Macros:

- They can make your program easier to read
- They can improve efficiency (as they can be calculated at compile time)
- They can shorten long or complicated expressions that are used a lot.

Disadvantages of Macros:

- Expand the size of your executable
- Can flood your name space if not careful. For example, if you have too many preprocessor macros you can accidentally use their names in your code, which can be very confusing to debug.

Example

```
#define INCREMENT(x) x++
```

Macro named INCREMENT has one argument **x**, value of x is incremented by 1 every time macro is called.

3) **#undef:** To undefined a macro means to cancel its definition. This is done with the ***#undef*** directive.

Runtime compile error.

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{
  printf("%f",PI);
}
```

Exercise

Theory Question

1. What is pre-processor directive?
2. What is difference between the macro and function?
3. Advantages and disadvantages of macro?
4. Write five list of pre-processor directive.
5. How many type of macro.

Practical Question

1) Write a program which takes three integers ***a, b, c*** as input and prints the largest one using define directive macro.
2) Which of the following are correctly formed ***#define*** statements and justify your reasons.
   i. #define INCH PER FEET 12
   ii. #define SQR (X) (X * X)
   iii. #define SQR(X) X * X

MCQ Questions

1) Pre-processor directive must be start _____ symbol
   a) $
   b) #
   c) _
   d) &
2) The directives define anywhere in program but are most often used ___.
   a) At the beginning of a file
   b) Within the main() function
   c) Within the any function
   d) Nothing all

3) The directives is use to import other file in the current file.
   a) #define
   b) #undef
   c) #if
   d) #include

4) Macro is widely used to represent _________.
   a) Numeric
   b) String
   c) Both a and b
   d) Nothing all

5) Functions executes relatively slower than macros
   a) True
   b) false