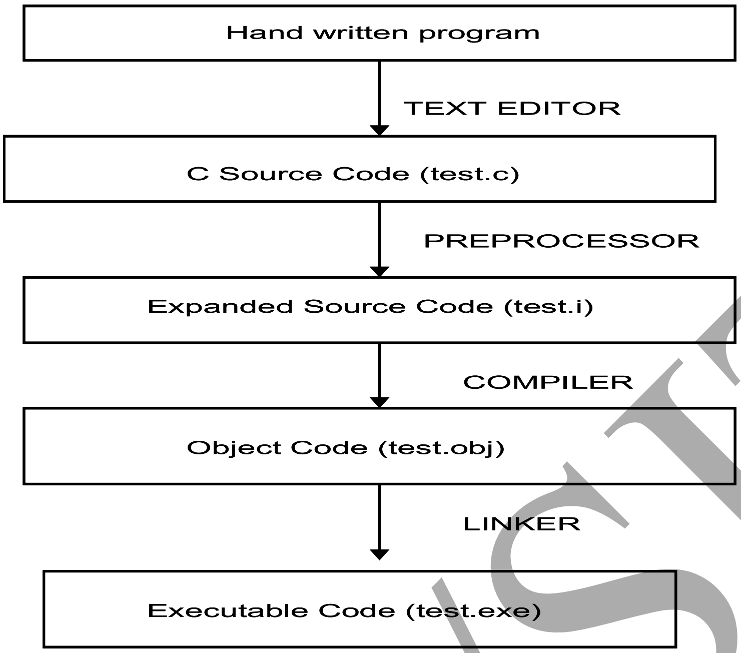
We MUST know the steps involved from writing a C program to getting it executed. These steps are shown in the following figure.



(NOTE that

- file extensions shown above in respect of working in TURBO C environment.
- TEXT EDITOR, COMPILER etc. are program)

Now from the above figure, we can say

- ✓ PREPROCESSOR is a program
- ✓ It processes the source code
- ✓ It does so before our source code goes to compiler.

What is Preprocessor?

- Preprocessor is a program that **processes the source code** before it passes through the compiler.
- Preprocessor operates under the control of preprocessor **directives**.
- **Preprocessor directives** can be placed anywhere but usually placed at the beginning of a program.
- **Preprocessor directives** start with symbol # and **do not** require a **semicolon** at the end.

Examples:

Directives	Functions performed
#define	Defines a macro substitution
#include	Specifies the files to be included
#ifdef	Tests for a macro definition
#ifndef	Tests whether a macro is **not** defined

Directive Category:

The directives can be divided into three categories:

i) **Macro substitution** directives.
ii) **File inclusion** directives.
iii) **Conditional compilation** directives.

(**Macro**: It is a **single identifier** that is **equivalent to** expression or complete statements.)

Explain Macro substitution directive with example.

Macro substitution is a **process** in which an identifier in a program is replaced by predefined string.

Example:

```
#define         PI      3.141
void main()
{
        float area, r = 2.0;
        area = PI * r * r;
        printf("\n area of circle = PI * r * r = %f", area);
}
```

- ✓ Here **PI** is a macro.
- ✓ The preprocessor directive **#define** will replace all occurrences of **PI** with 3.141, starting from the line of definition to the end of the program.
- ✓ However a **macro** inside a string **does not** get replaced.
- ✓ Thus the following line

```
        area = PI * r * r;
    will ONLY be changed as follows:
        area = 3.141 * r *r;
```

===

Examples to clarify the concept of macro substitution further:

1. Example 1: OBSERVE the explanation carefully.

```
#include <stdio.h>
#define Z     x-y
void main()
{
   int x = 5, y = 4, a;
   a = Z * 3;
   printf("\n a = %d", a);
 }
```

Output:
 a = -7

```
#include <stdio.h>
#define Z     (x-y)
void main()
{
   int x = 5, y = 4, a;
   a = Z * 3;
   printf("\n a = %d", a);
 }
```

Output:
 a = 3

Explanation:

In the LHS program, the line **a = Z * 3;** will be changed to **a = x – y *3;**
So, putting values of x and y we get
a = 5 – 4 * 3
= 5 – 12 [As operator precedence of * is higher than –]
= – 7

In the RHS program, the line **a = Z * 3;** will be changed to **a = (x – y) *3;**
So, putting values of x and y we get
a = (5 – 4) * 3
= 1 * 3 [Natural order of evaluation is changed due to parentheses]
= 3

2. Example 2: OBSERVE carefully.

```
#include <stdio.h>
#define SQR(x)     x * x
void main()
{
   int  a , b = 3;
   a = SQR(b+2);
   printf("\n a = %d", a);
 }
```

Output:
 a = 11

```
#include <stdio.h>
#define SQR(x)     (x) * (x)
void main()
{
   int  a , b = 3;
   a = SQR(b+2);
   printf("\n a = %d", a);
 }
```

Output:
 a = 25

First try to **explain** the **outputs** and then go to the **NEXT** page**.**

Explanation:

In the LHS program, the line **a = SQR(b+2);** will be changed to **a = b + 2 * b + 2;**

So, putting values of b we get

```
a  = 3 + 2 * 3 + 2
   = 3 + 6 + 2   [As operator precedence of  * is higher than + ]
   =  9 + 2       [ As Associativity of + operator is Left to Right ]
   =  11
```

In the RHS program, **a = SQR(b+2);** will be changed to **a = (b + 2) * (b + 2);**

So, putting values of b we get

```
a  = (3 + 2) * (3 + 2)
   = 5 * 5   [Natural order of evaluation is changed due to parentheses]
   =  25
```

NOTE that Macro and function are different. The basic difference is that macro is preprocessed but function is compiled.

Review Question 1:

Find the OUTPUTS of the following programs.

#include <stdio.h>
#define CUBE(x)   (x * x * x)
void main()
{
   int a , b = 3;
   a = CUBE(b+2);
   printf("\n a = %d", a);
 }
	#include <stdio.h>
	#define CUBE(x) (x) * (x) * (x)
	void main()
	{
	   int a , b = 3;
	   a = CUBE(b+2);
	   printf("\n a = %d", a);
	 }

Output: **?**	Output: **?**

REMEMBER HERE YOU NEED TO SPECIFY OUTPUTS WITH EXPLANATIONS.

Explain File inclusion directive with example.

File Inclusion Directive: It is used to include the entire contents of an external file containing functions declarations & definitions and macro definitions into the source code.

It has two forms:
i) #include “**filename**”

ii) #include <**filename**>

In either form, it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program.

In the **first form**, the search for the file is **first made in the current directory** and if the required file is not found in current directory, then search is made in the **standard directories**.

In the second form, the search is made **only in the standard directories**. If the file is not found at all then an error is reported and compilation is terminated.

Thus when we write

#include <stdio.h>

The search is made in standard directories and the content of header file **stdio.h** is inserted into the source program as if it is a part of the source program and is sent to the compiler for compilation.

==

Explain Conditional Compilation directive with example.

Conditional Compilation: It can be used to include a file or macro depending on some conditions. A section of a source code may be compiled conditionally using the conditional compilation facilities. The directive for this purpose are #ifdef, #ifndef, #else, #endif etc.

Example:

```
#include <stdio.h>
#define CUBE
void main()
{
        int x=5, result;
        #ifdef  CUBE
              result =  x * x * x;
        #else
              result = x * x;
        #endif
        printf("\nresult=%d",result);
  }
```

The output of this program is 125.

But if we omit the line **#define CUBE** from the program and then recompile & run it, the output will be 25.

WHY do we get above outputs?

Because –
if macro CUBE is defined then compiler compiles the statement

```
result = x * x * x;
```

and then on execution, result is assigned a value of 5 * 5 * 5 i.e. 125, which is displayed as output.

otherwise (i.e. if we omit the line #define CUBE) the compiler compiles the statement

```
result = x * x;
```

and then on execution, result is assigned a value of 5* 5 i.e. 25, which is displayed as output.

==

What is macro?

Macro is a **single identifier** that is **equivalent to** expressions or complete statements or group of statements.

Example:

```
#include <stdio.h>
#define     AREA      length x width
void main()
{
        int len = 10, width = 5;
        printf(“\n Area = %d”, AREA);
}
```

Above program contains macro **AREA**, which represents the expression **length x width**.

==

How a macro is different from a C function?

Macro is preprocessed. This means that a macro would be processed before the C program is compiled. However function is not preprocessed but compiled.

Macro	Function

#include <stdio.h>
#define   NUM   5
void main( )
{
        printf(“\n%d”, NUM);
 }
	#include <stdio.h>
	void main( )
	{
	     int NUM( );
	     printf(“\n %d”, NUM( ));
	 }
	int NUM( )
	{
	       return(5);
	}

Output: 5	Output: 5
Here NUM is **macro** and preprocessed. During preprocessing, the statement printf(“\n%d”, NUM); is changed to printf(“\n%d”, 5); which gives 5 as output	Here NUM() is a **function** and it is compiled. When this function is called, control goes to NUM( ) function and **return(5)**; statement is encountered. Value 5 is returned from the function and control comes back to printf statement which prints 5 as output.

Compare macro and C function:

Macro	Function
Macro is preprocessed. This means that macro is preprocessed before the C program is compiled.	Function is compiled.
Before compilation macro name is replaced by macro value.	During function call, transfer of control takes place.
No type checking is done	Type checking is done.
Speed of execution is faster	Speed of execution is slower because of function call overheads.

#include <stdio.h>
#define NUM 5
void main( )
{
printf("\n%d", NUM);
}
	#include <stdio.h>
	void main( )
	{
	int NUM( );
	printf("\n %d", NUM( ));
	}
	int NUM( )
	{
	return(5);
	}

Output: 5	Output: 5

Here NUM is **macro** and preprocessed.
During preprocessing, the statement
printf("\n%d", NUM);
is changed to
printf("\n%d", 5);
which gives 5 as output
	Here NUM() is a **function** and it is compiled. When this function is called, control goes to NUM( ) function and **return(5)**; statement is encountered. Value 5 is returned from the function and control comes back to printf statement which prints 5 as output.

Advantages of Preprocessor:

- It improves the readability of programs.
- It facilitates easier modifications.
- It helps in writing portable programs.
- It enables testing a part of a program.
- It helps in developing generalized program.

Review Question 2:

Explain the role of C preprocessor.