Recursion in C

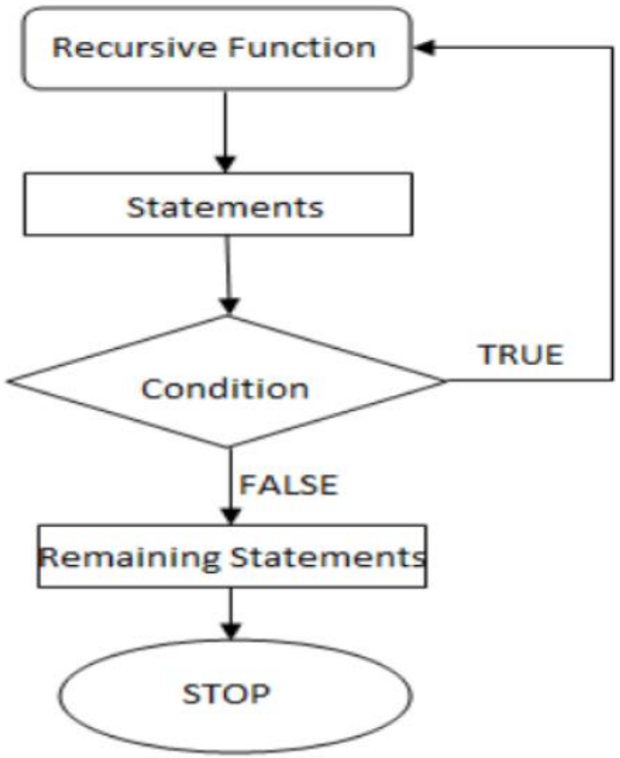
By: Atiya Jokhio

Recursion

- When a function invokes itself, the call is known as a recursive call.
- Recursion (the ability of a function to call itself) is an alternative control structure to repetition (looping). Rather than use a looping statement to execute a program segment, the program uses a selection statement to determine whether to repeat the code by calling the function again or to stop the process.

Recursion

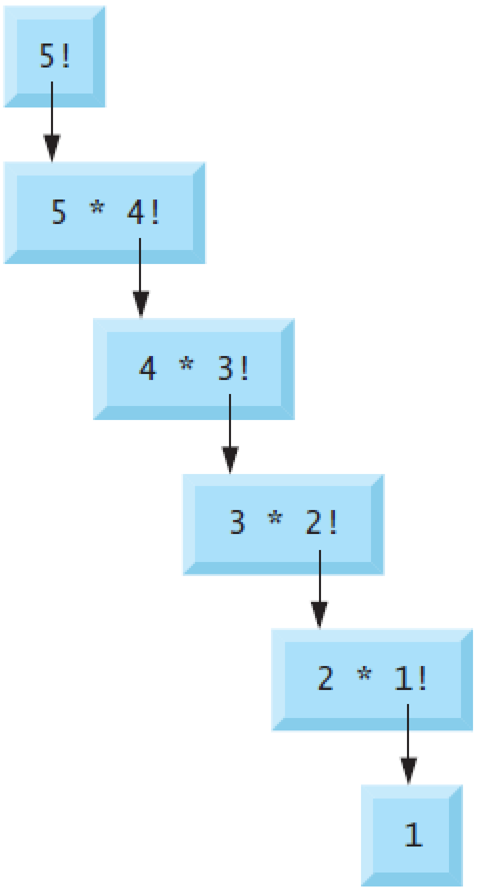
Flowchart for recursion:



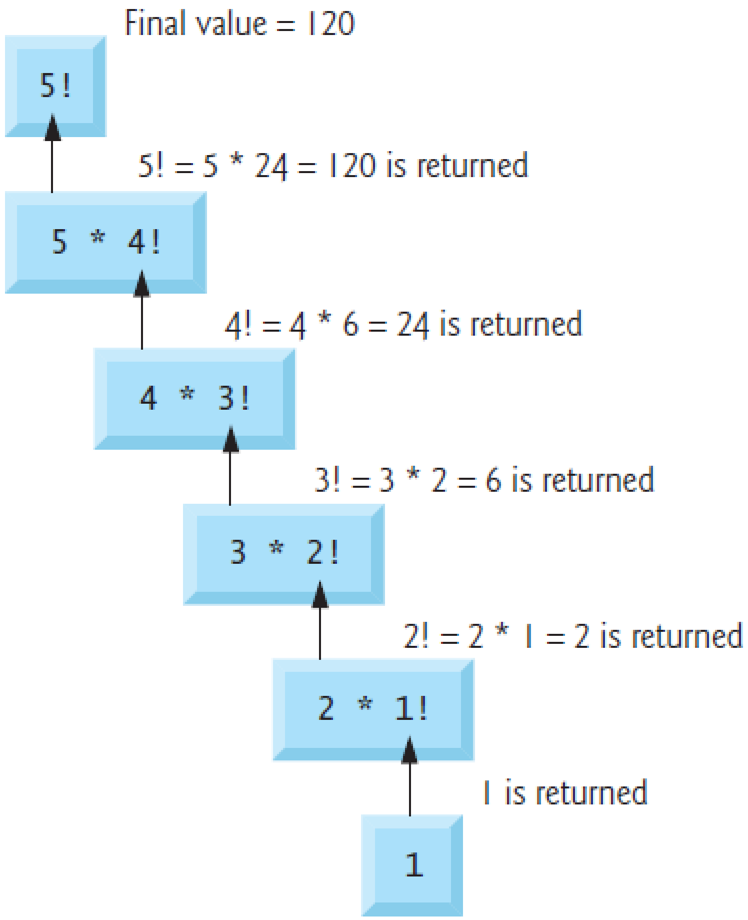
Recursion

- Each recursive solution has at least two cases: the base case and the general case.
  - The **base case** is the one to which we have an answer;
  - the **general case** expresses the solution in terms of a call to itself with a smaller version of the problem. Because the general case solves a smaller and smaller version of the original problem, eventually the program reaches the base case, where an answer is known, and the recursion stops.

Recursion



(a) Sequence of recursive calls



(b) Values returned from each recursive call

Recursion

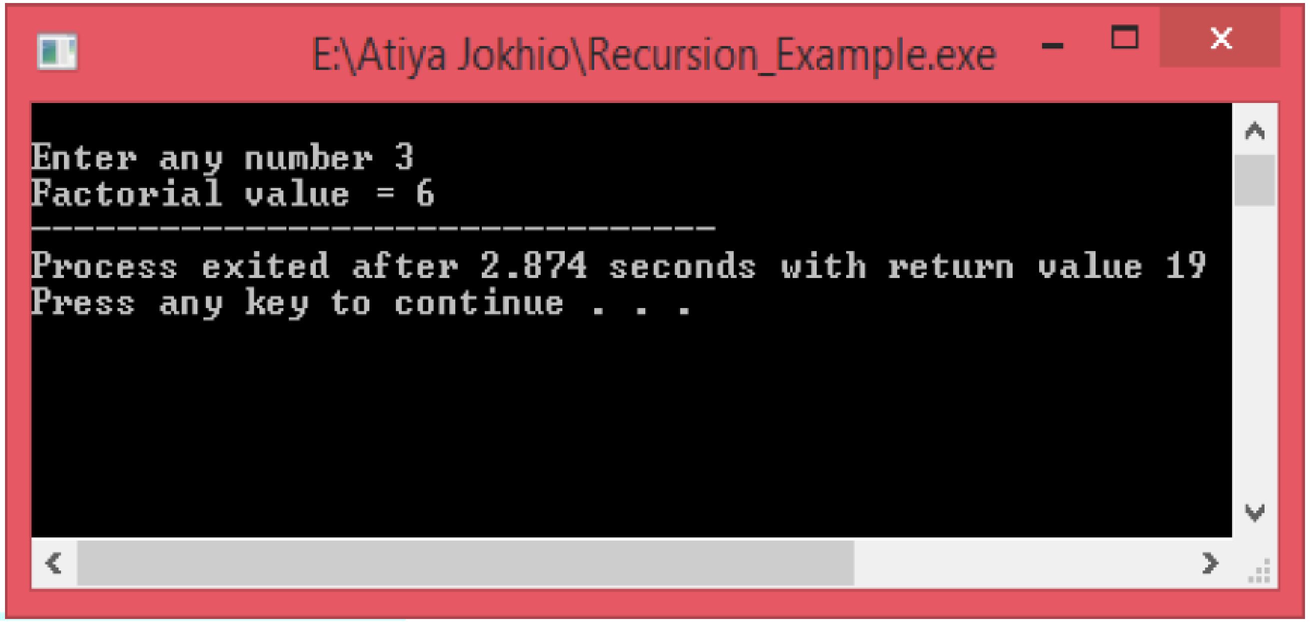
- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

Recursion

- For example, a classic recursive problem is the factorial. The factorial of a number is defined as the number times the product of all the numbers between itself and 0: N! = N * (N-1)!
- The factorial of 0 is 1. We have a base case, Factorial (0) is 1, and we have a general case, Factorial (N) is N * Factorial (N-1). An if statement can evaluate N to see if it is 0 (the base case) or greater than 0 (the general case). Because N is clearly getting smaller with each call, the base case is reached.

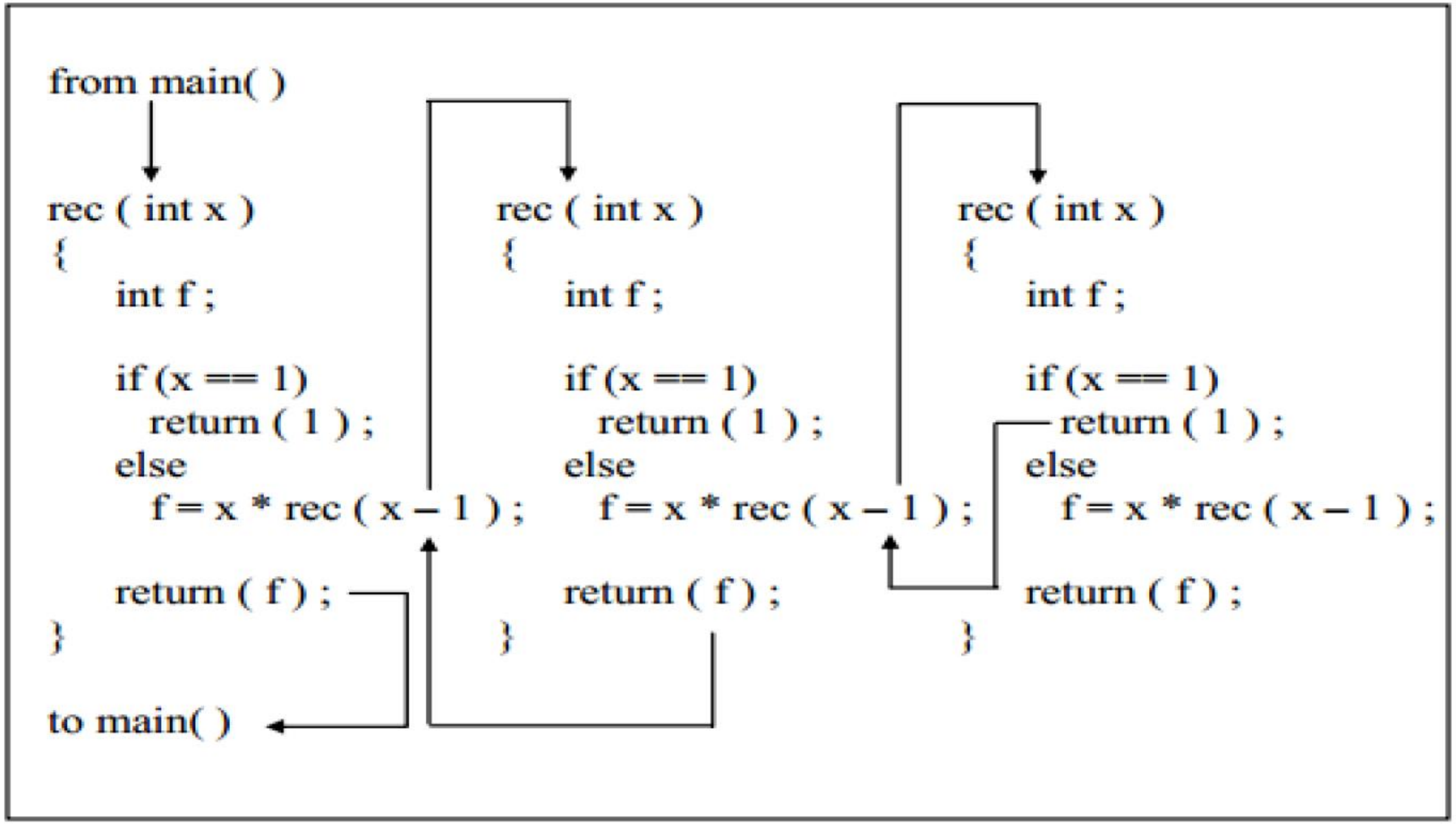
Recursion

```c
#include <stdio.h>
int main( )
{
    int a, fact ;
    printf ( "\nEnter any number " ) ;
    scanf ( "%d", &a ) ;
    fact = rec ( a ) ;
    printf ( "Factorial value = %d", fact ) ;
    }
    rec ( int x )
    {
    int f ;
    if ( x == 1 )
    return ( 1 ) ;
    else
    f = x * rec ( x - 1 ) ;
    return ( f ) ;
    }
```



Recursion

- Assume that the number entered through scanf( ) is 3. The figure below explains what exactly happens when the recursive function rec( ) gets called.



Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:

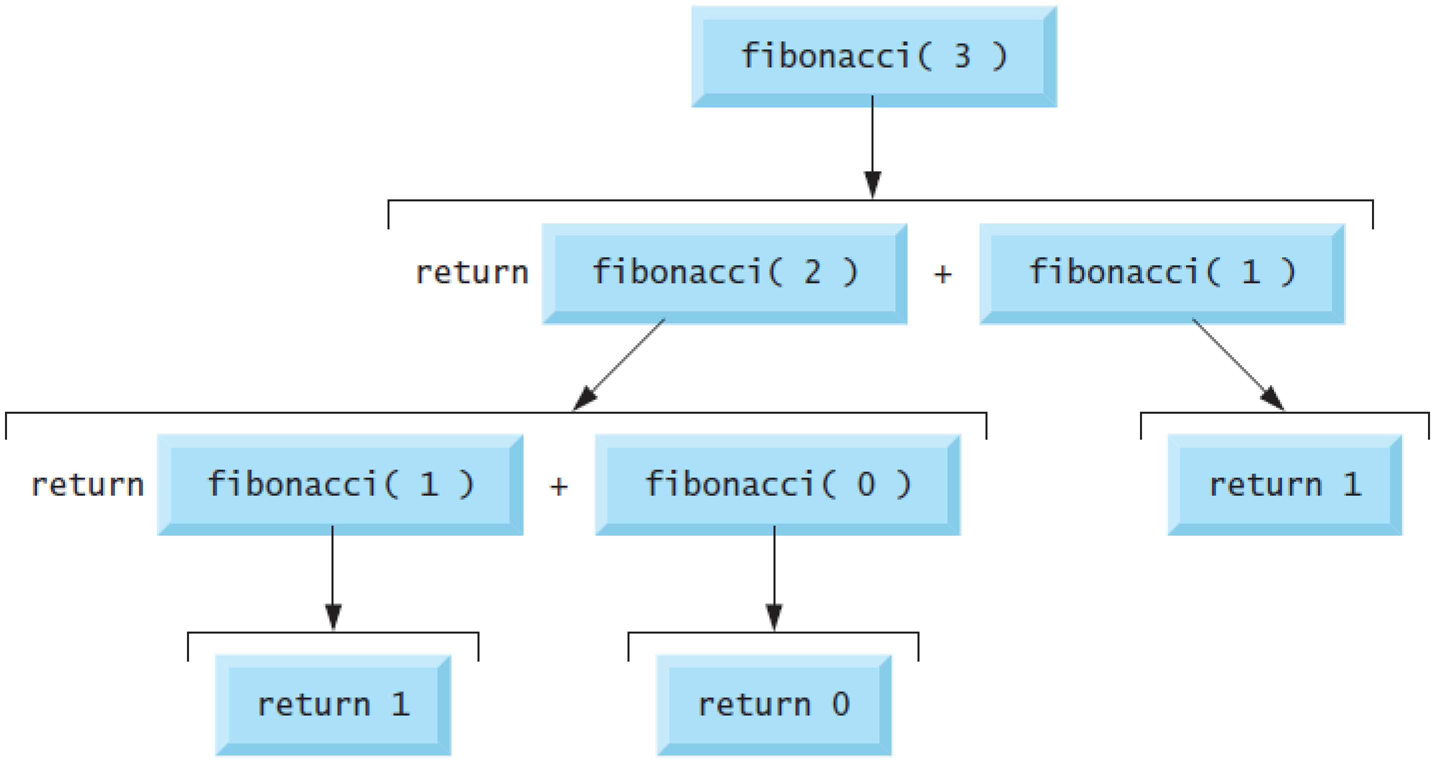
```
fib( n ) = fib( n - 1 ) + fib( n – 2 )
```

  - Code for the `fibonacci` function

```
long fibonacci( long n )
{
    if (n == 0 || n == 1)  // base case
     return n;
    else
     return fibonacci( n - 1) +
      fibonacci( n – 2 );
}
```

Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function `Fibonacci`



Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

Recursion

- **Disadvantages of recursion**

  – Recursive programs are generally slower than non-recursive programs. This is because, recursive function needs to store the previous function call addresses for the correct program jump to take place.

  – Requires more memory to hold intermediate states. It is because, recursive program requires the allocation of a new stack frame and each state needs to be placed into the stack frame, unlike non-recursive (iterative) programs.

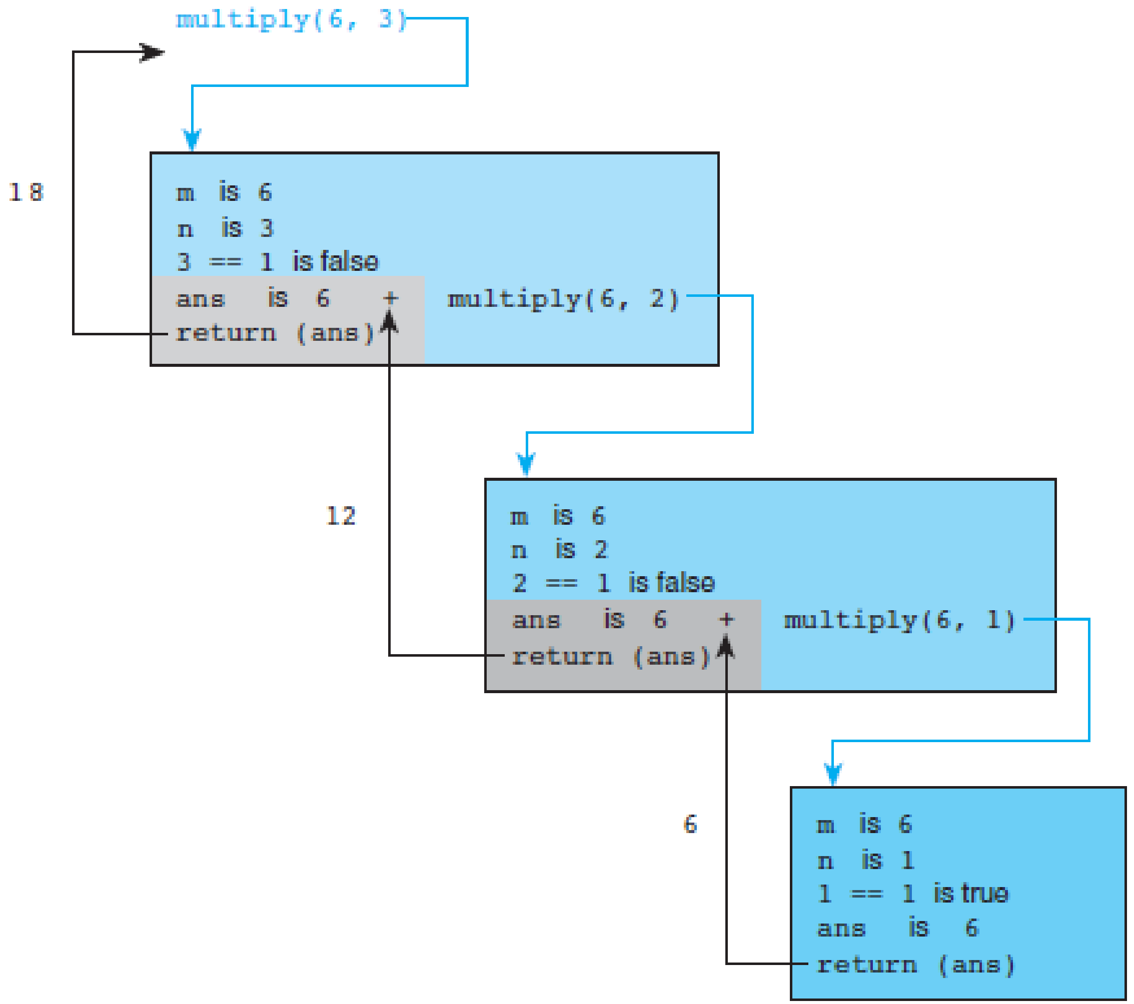
Write a program that performs integer multiplication using + operator.

```
int
multiply(int m, int n)
{
    int ans;

    if (n == 1)
          ans = m;    /* simple case */
    else
          ans = m + multiply(m, n - 1); /* recursive step */

    return (ans);
}
```

Tracing



multiply(6, 3)
18
m is 6
n is 3
3 == 1 is false
ans is 6 +
return (ans)
multiply(6, 2)
12
m is 6
n is 2
2 == 1 is false
ans is 6 +
return (ans)
multiply(6, 1)
6
m is 6
n is 1
1 == 1 is true
ans is 6
return (ans)

Recursion Practice Questions

- Display series from 1 to 10.
- Display following pattern:

  a

  a b

  a b c

  a b c d

  a b c d e

- Get the character and display until he/she hits the enter key. Finally show the count of total typed characters.

Preprocessor Directives

- The **C preprocessor executes** ***before a program is compiled***.
- Normal program statements are instructions to the microprocessor; preprocessor directives are instructions to the compiler.
- The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with a #.

Preprocessor Directives

- Use of the preprocessor is advantageous since it makes:
  - programs easier to develop,
  - easier to read,
  - easier to modify
  - C code more transportable between different machine architectures.
- Here we'll examine two of the most common preprocessor directives, #define and #include.

The #define Directive(Symbolic Constant)

The simplest use for the define directive is to assign names to constants. e.g.

```
#define PI 3.14159 //macro definition
float area (float);
   void main()
   {
    float radius;
    printf(“Enter radius of sphere: “);
    scanf(“%f”,&radius);
    printf(“Area of sphere is %.2f”, area(radius) );
   }
   float area( float rad)
   {return(4 * PI * rad * rad ); }
```

Macros

The macro that we have used so far is called simple macro. Macros can have arguments, just as functions can.

e.g.

```
#define Area(x) (3.14 * x * x)
main()
{
   float r1=4.6; r2= 6.25;
   a=Area(r1);
   printf("\nArea of Circle = %f ", a);
   a=Area(r2);
   printf("\nArea of Circle = %f", a);
}
```