FUNCTIONS IN C

Functions

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - Local variables
- Benefits of functions
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoid code repetition

Function Definitions

- Function definition format

  return-value-type function-name(parameter-list)
  {
  declarations and statements
  }

  - Function-name: any valid identifier
  - Return-value-type: data type of the result (default `int`)
    - `void` – indicates that the function returns nothing
  - Parameter-list: comma separated list, declares parameters
    - A type must be listed explicitly for each parameter unless, the parameter is of type `int`

Function Definitions

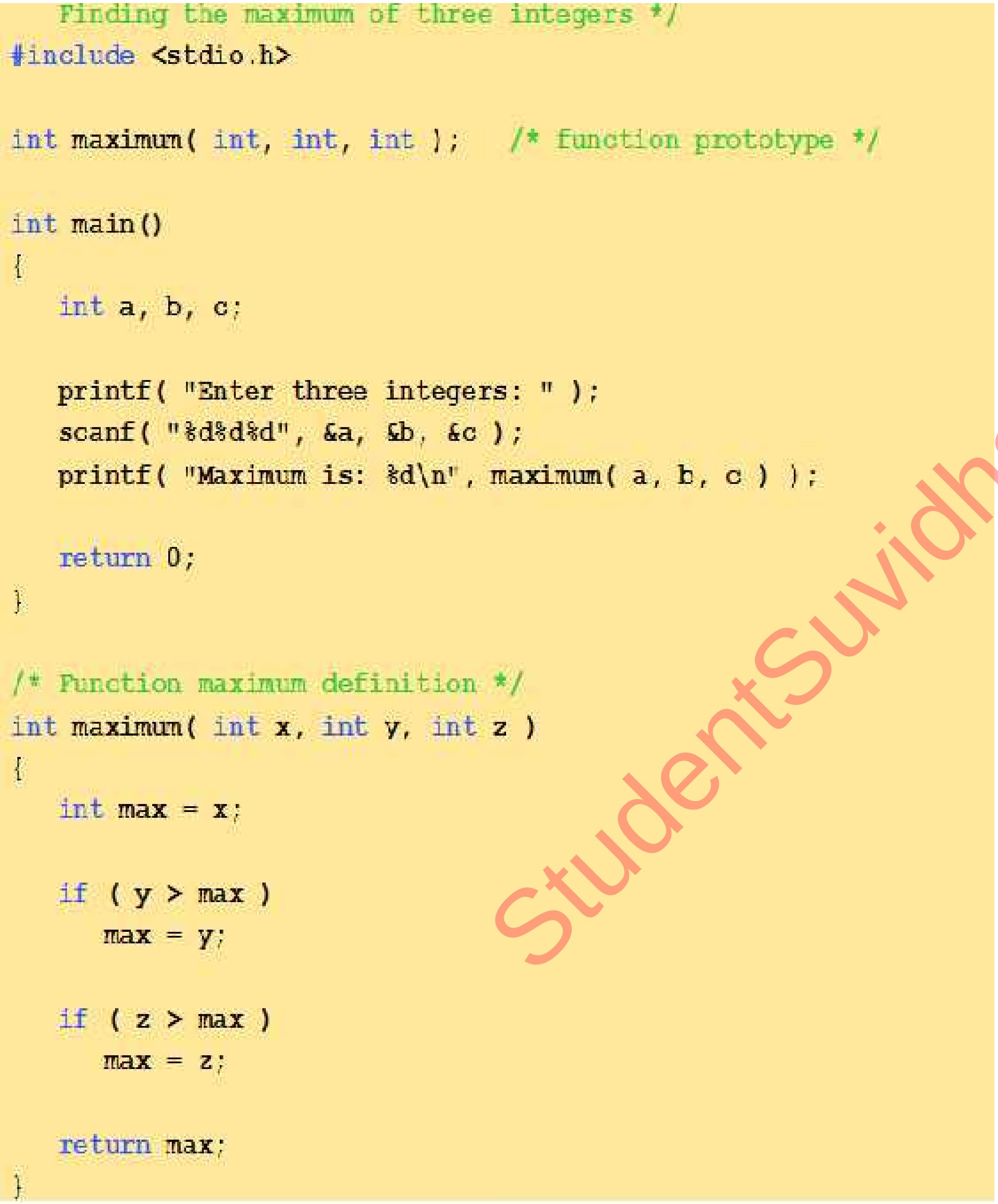
- Function definition format (continued)

  return-value-type function-name(parameter-list)
  {
  declarations and statements
  }

  - Declarations and statements: function body (block)
    - Variables can be declared inside blocks (can be nested)
    - Functions can not be defined inside other functions
  - Returning control
    - If nothing returned
      - **`return;`**
      - or, until reaches right brace
    - If something returned
      - **`return`** *expression*;

Example function

```
#include<stdio.h>
void fun(int a);                    //declaration
int main()
{
   fun(10);                         //Call
}
void fun(int x)                     //definition
{
   printf(“%d”,x);
}
```



```
   Finding the maximum of three integers */
#include <stdio.h>

int maximum( int, int, int );   /* function prototype */

int main()
{
   int a, b, c;

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );

   return 0;
}

/* Function maximum definition */
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

3. Function definition

Function Prototypes

- Function prototype
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default `int`)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
  - The function with the prototype

    ```
    int maximum( int, int, int );
    ```

    - Takes in 3 `int`s
    - Returns an `int`

Actual and Formal parameters

- Actual parameters are those that are used during a function call

- Formal parameters are those that are used in function definition and function declaration

Calling Functions: Call by Value and Call by Reference

- Call by value => copying value of variable in another variable. So any change made in the copy will not affect the original location.
- Call by reference => Creating link for the parameter to the original location. Since the address is same, changes to the parameter will refer to original location and the value will be over written.

Call by value

- Calling a function with parameters passed as values

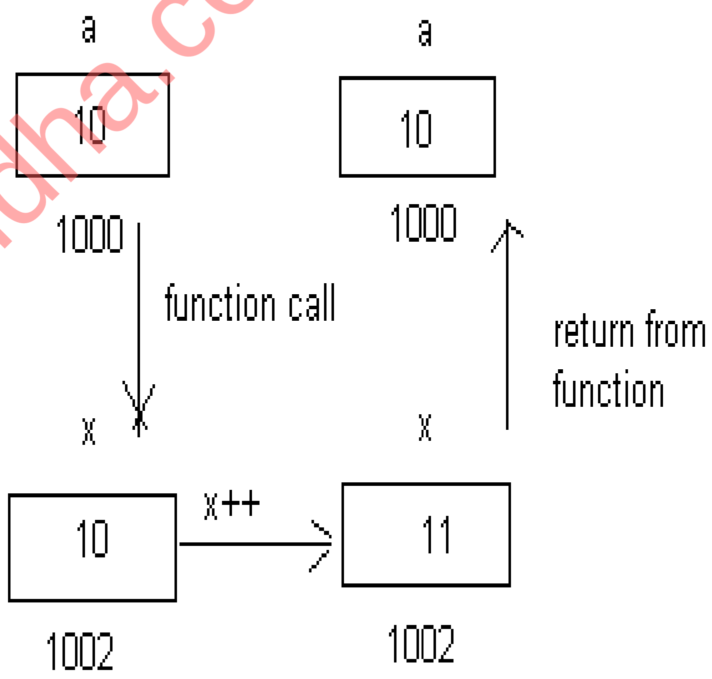
```
int a=10;                    void fun(int a)
fun(a);                      {
                                     defn;
                             }
```

Here fun(a) is a call by value.

Any modification done with in the function is local to it and will not be effected outside the function

Example program – Call by value

```
#include<stdio.h>
void main()
{
    int a=10;
    printf("%d",a);          a=10
    fun(a);
    printf("%d",a);          a=10
}
void fun(int x)
{
    printf("%d",x)           x=10
    x++;
    printf("%d",x);          x=11
}
```



Call by reference

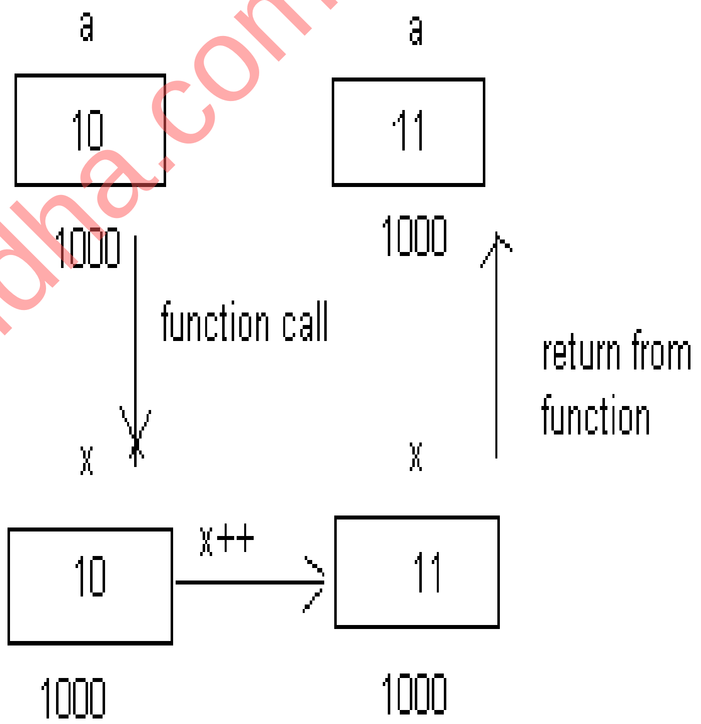
- Calling a function by passing pointers as parameters (address of variables is passed instead of variables)

```
int a=1;                    void fun(int *x)
fun(&a);                    {
                                    defn;
                            }
```

Any modification done to variable a will effect outside the function also

Example Program – Call by reference

```
#include<stdio.h>
void main()
{
      int a=10;
      printf(“%d”,a);              a=10
      fun(a);
      printf(“%d”,a);              a=11
}
void fun(int x)
{
      printf(“%d”,x)               x=10
      x++;
      printf(“%d”,x);              x=11
}
```



a and x are referring to same location. So value will be over written.

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem
- 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;`**

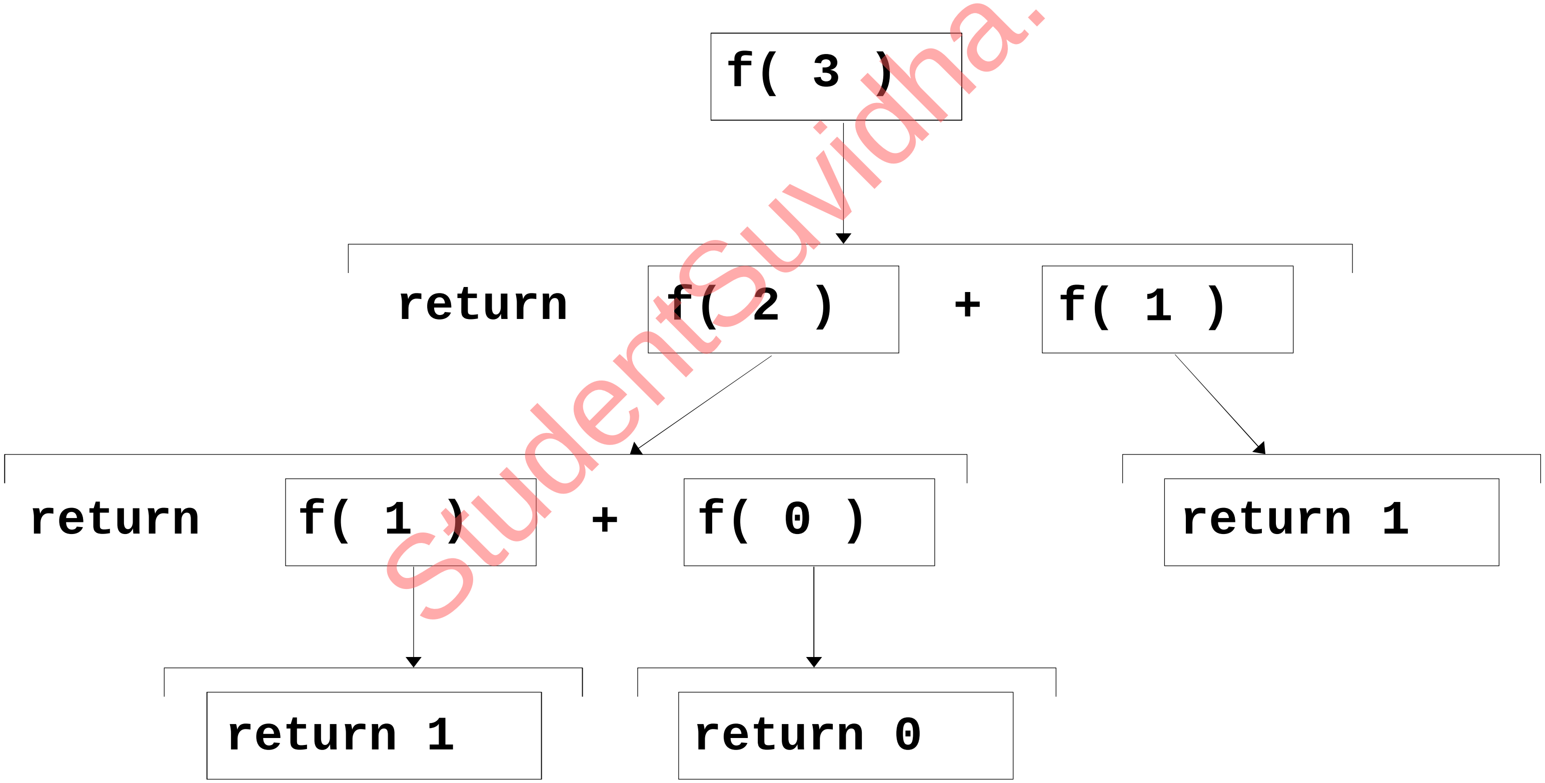
Example Using Recursion: The Fibonacci Series

- 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 **`fibaonacci`** function

```
int fibonacci( int 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`**



f(3)

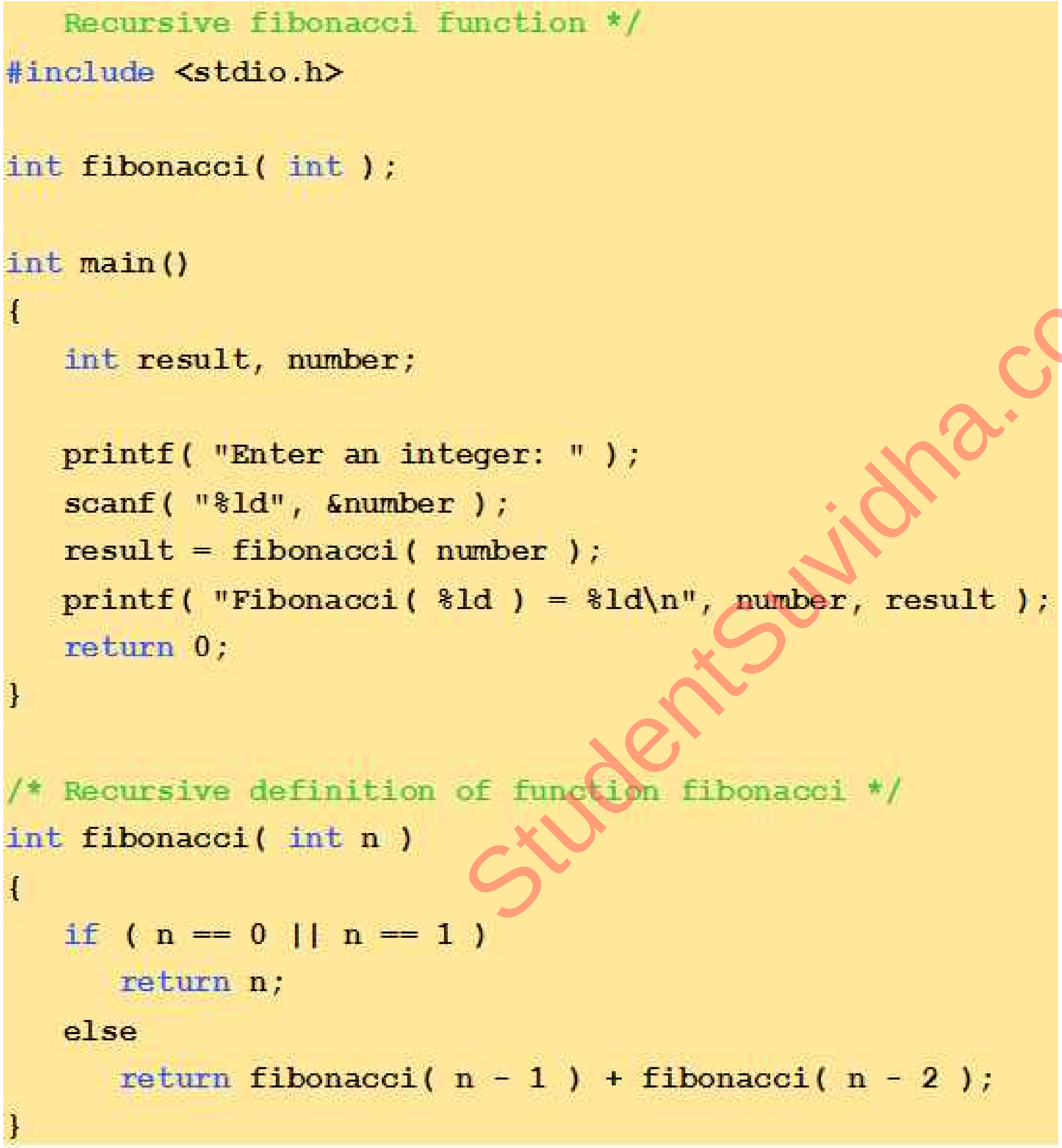
return f(2) + f(1)

return f(1) + f(0)

return 1

return 1

return 0



```
   Recursive fibonacci function */
#include <stdio.h>

int fibonacci( int );

int main()
{
   int result, number;

   printf( "Enter an integer: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Recursive definition of function fibonacci */
int fibonacci( int n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

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)

Storage Classes

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known (more in Chapter 14)
- Automatic storage
  - Object created and destroyed within its block
  - **`auto`**: default for local variables

```
auto double x, y;
```

  - **`register`**: tries to put variable into high-speed registers
    - Can only be used for automatic variables

```
register int counter = 1;
```

Storage Classes

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - **`static`**: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
  - **`extern`**: default for global variables and functions
    - Known in any function

Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
- Used only for labels (`start:`, `case:` , etc.)
- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list

Assignment

- Write a program to calculate factorial of a number using recursion
- Write a program to calculate x to the power y using recursion.
- Explain storage classes in C in detail.
- Differentiate between call by value and call by reference.