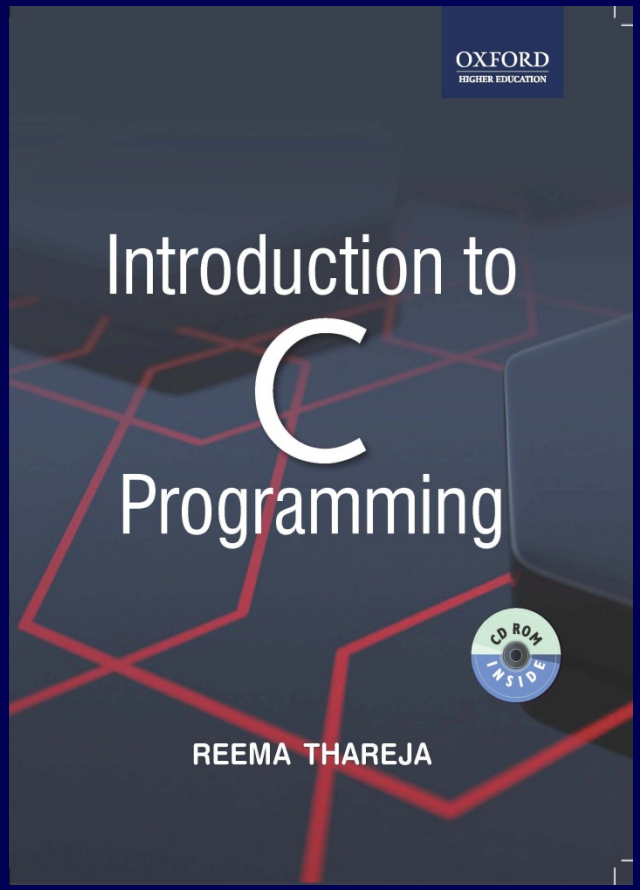
Introduction to C Programming

Reema Thareja, Assistant Professor,
Institute of Information Technology and Management

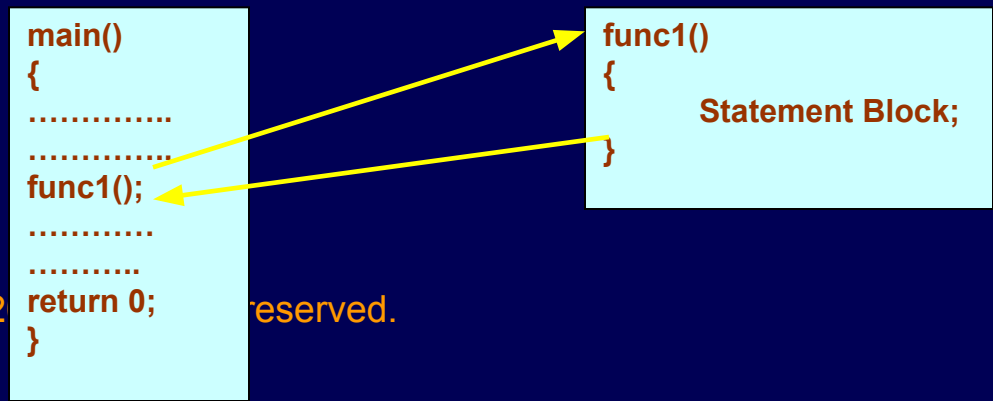


CHAPTER 4

FUNCTIONS

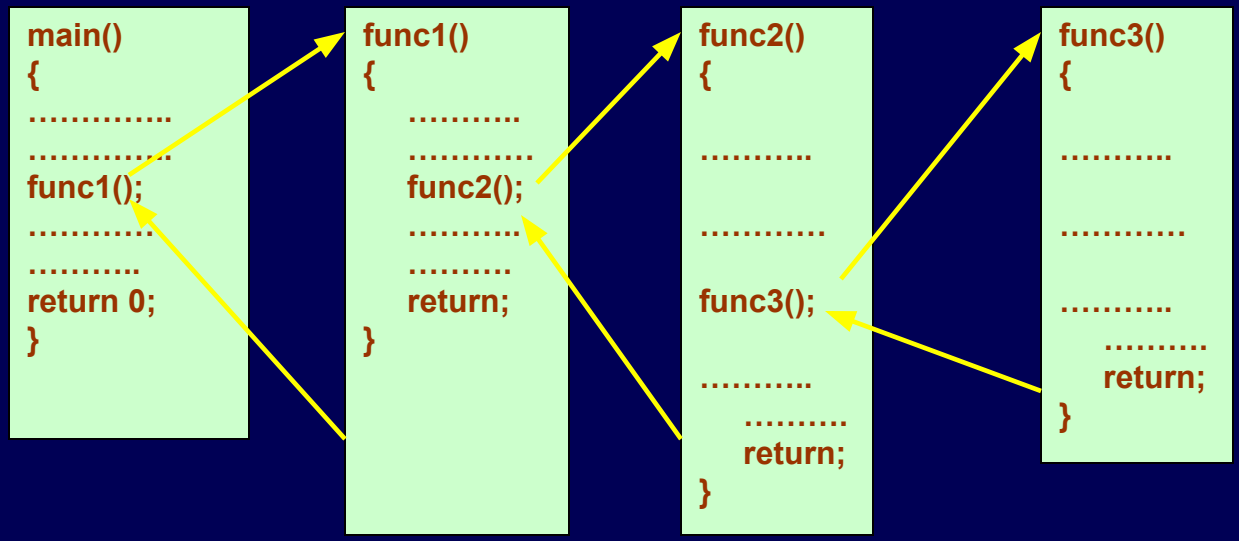
INTRODUCTION

- C enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others.
- Every function in the program is supposed to perform a well defined task. Therefore, the program code of one function is completely insulated from that of other functions.
- Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it.
- **In the fig, main() calls another function, func1() to perform a well defined task.**
- **main() is known as the calling function and func1() is known as the called function.**
- **When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.**
- **After the called function is executed, the control is returned back to the calling program.**



INTRODUCTION CONTD….

- It is not necessary that the main() can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for loop, while loop or do-while loop may call the same function multiple times until the condition holds true.
- It is not that only the main() can call another functions. Any function can call any other function. In the fig. one function calls another, and the other function in turn calls some other function.



WHY DO WE NEED FUNCTIONS?

- Dividing the program into separate well defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function (except main()), then there will be countless lines in the main() .
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been prewritten and pre-tested, so the programmers use them without worrying about their code details. This speeds up program development.

TERMINOLOGY OF FUNCTIONS

- A function, *f* that uses another function *g*, is known as the *calling function* and g is known as the *called function*.
- The inputs that the function takes are known as *arguments*
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- Main() is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.

FUNCTION DECLARATION

- *Function declaration* is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- The general format for declaring a function that accepts some arguments and returns some value as result can be given as:

  return_data_type function_name(data_type variable1, data_type variable2,..);
- No function can be declared within the body of another function.

  Example, float avg (int a, int b);

FUNCTION DEFINITION

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function
- When a function defined, space is allocated for that function in the memory.
- The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
 ……..
 statements
 ……..
 return( variable);
}
```

- The no. and the order of arguments in the function header must be same as that given in function declaration statement.

FUNCTION CALL

- The function call statement invokes the function.
- When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.
- Once the called function is executed, the program control passes back to the calling function.
- Function call statement has the following syntax.

```
function_name(variable1, variable2, …);
```

Points to remember while calling the function:

- Function name and the number and type of arguments in the function call must be same as that given in the function declaration and function header of the function definition
- Names (and not the types) of variables in function declaration, function call and header of function definition may vary
- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

```
variable_name = function_name(variable1, variable2, …);
```

PROGRAM THAT USES FUNCTION

```
#include<stdio.h>
int sum(int a, int b);       // FUNCTION DECLARATION
int main()
{
      int num1, num2, total = 0;
      printf("\n Enter the first number : ");
      scanf("%d", &num1);
      printf("\n Enter the second number : ");
      scanf("%d", &num2);
      total = sum(num1, num2);           // FUNCTION CALL
      printf("\n Total = %d", total);
      return 0;
}
// FUNCTION DEFNITION
int sum ( int a, int b)            // FUNCTION HEADER
{                          //  FUNCTION BODY
      return (a + b);
}
```

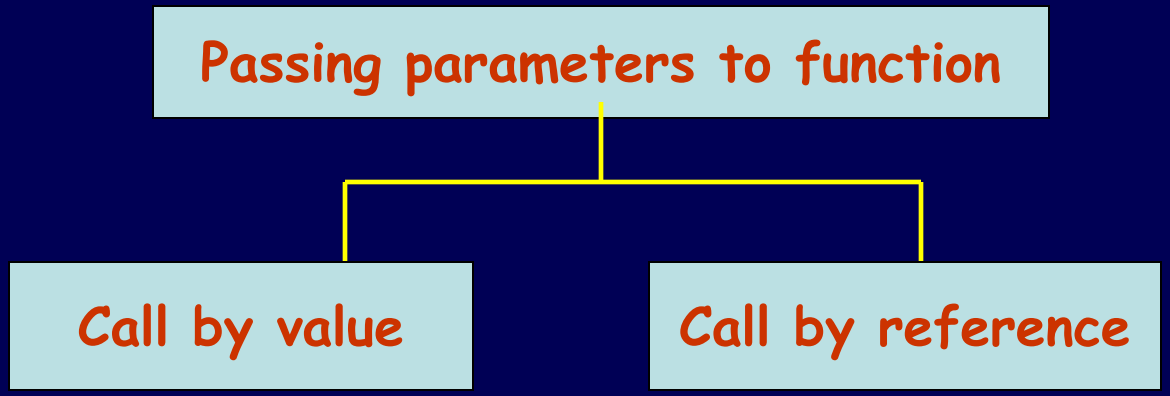
RETURN STATEMENT

- The **return** statement is used to terminate the execution of a function and return control to the calling function. When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- **Programming Tip:** It is an error to use a return statement in a function that has void as its return type.
- A **return** statement may or may not return a value to the calling function. The syntax of return statement can be given as

  return <*expression***> ;**
- Here expression is placed in between angular brackets because specifying an expression is optional. The value of *expression*, if present, is returned to the calling function. However, in case *expression* is omitted, the return value of the function is undefined.
- Programmer may or may not place the *expression* within parentheses.
- By default, the return type of a function is *int.*
- For functions that has no **return** statement, the control automatically returns to the calling function after the last statement of the called function is executed.

PASSING PARAMETERS TO THE FUNCTION

- There are two ways in which arguments or parameters can be passed to the called function.
- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.



CALL BY VALUE

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

```
#include<stdio.h>
void add( int n);
int main()
{
  int num = 2;
  printf("\n The value of num before calling the function = %d", num);
  add(num);
  printf("\n The value of num after calling the function = %d", num);
  return 0;
}
void add(int n)
{
  n = n + 10;
  printf("\n The value of num in the called function = %d", n);
}
```

The output of this program is:

The value of num before calling the function = 2

The value of num in the called function = 20

The value of num after calling the function = 2

CALL BY REFERENCE

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

PROGRAM ILLUSTRATING CALL BY REFERENCE TECHNIQUE

```
#include<stdio.h>
void add( int &n);
int main()
{
  int num = 2;
  printf("\n The value of num before calling the function = %d", num);
  add(num);
  printf("\n The value of num after calling the function = %d", num);
  return 0;
}
void add( int &n)
{
  n = n + 10;
  printf("\n The value of num in the called function = %d", n);
}
```

The output of this program is:

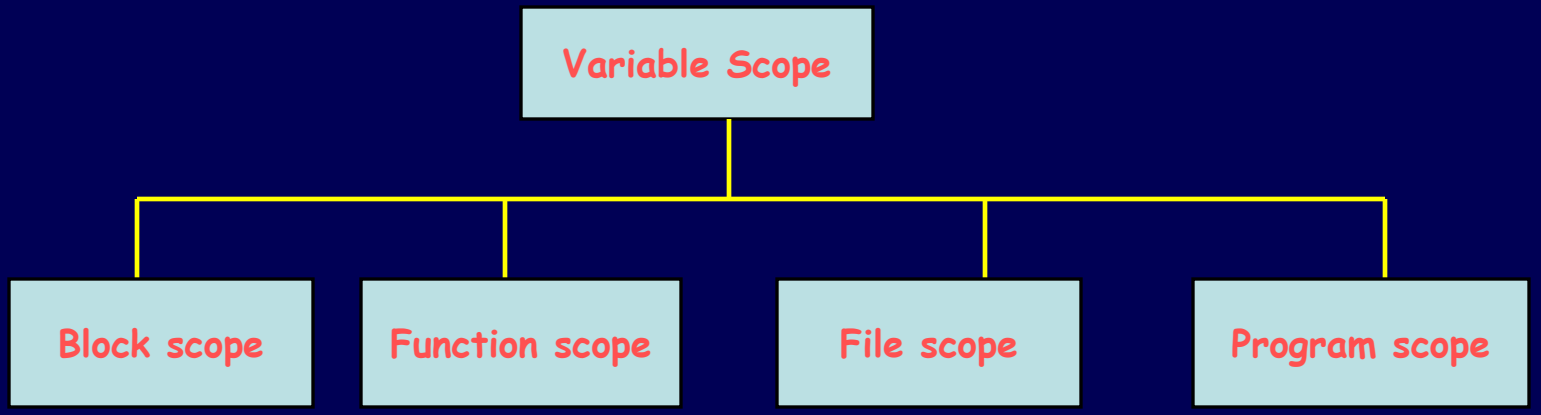
The value of num before calling the function = 2

The value of num in the called function = 20

The value of num after calling the function = 20

VARIABLES SCOPE

- In C, all constants and variables have a defined scope.
- By scope we mean the accessibility and visibility of the variables at different points in the program.
- A variable or a constant in C has four types of scope: block, function, file and program scope.



BLOCK SCOPE

- A statement block is a group of statements enclosed within an opening and closing curly brackets ({ }). If a variable is declared within a statement block then, as soon as the control exits that block, the variable will cease to exist. Such a variable also known as a local variable is said to have a block scope.

```
#include <stdio.h>
int main()
 {
        int x = 10.  i;
       printf("\n The value of x outside the while loop is %d", x);
       while (i<3)
       {       int x = i;
          printf("\n The value of x inside the while loop is %d", x);
              i++;
       }
    printf("\n The value of x outside the while loop is %d", x);
    return 0;
}
```

Output:

```
The value of x outside the while loop is 10
 The value of x inside the while loop is 0
 The value of x inside the while loop is 1
 The value of x inside the while loop is 2
 The value of x outside the while loop is 10
```

FUNCTION SCOPE

- Function scope is applicable only with goto label names. That is the programmer can not have the same label name inside a function.

PROGRAM SCOPE

- If you want that functions should be able to access some variables which are not passed to them as arguments, then declare those variables outside any function blocks. Such variables are commonly known as global variables. Hence, global variables are those variables that can be accessed from any point in the program.

```
#include<stdio.h>
int x = 10;
void print();
int main()
{       printf("\n The value of x in the main() = %d", x);
        int x = 2;
        printf("\n The value of local variable x in the main() = %d", x);
        print();
}
void print()
{       printf("\n The value of x in the print() = %d", x);
}
```

FILE SCOPE

- When a global variable is accessible until the end of the file, the variable is said to have file scope.
- To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type, like this:

```
static int x = 10;
```

- A global static variable can be used any where from the file in which it is declared but it is not accessible by any other files.
- Such variables are useful when the programmer writes his **own header files.**

STORAGE CLASSES

- The storage class of a variable defines the scope (visibility) and life time of variables and/or functions declared within a C Program. In addition to this, the storage class gives the following information about the variable or the function.
- It is used to determine the part of memory where storage space will be allocated for that variable or function (whether the variable/function will be stored in a register or in RAM)
- it specifies how long the storage allocation will continue to exist for that function or variable.
- It specifies the scope of the variable or function. That is, the part of the C program in which the variable name is visible, or accessible.
- It specifies whether the variable or function has internal, external, or no linkage
- It specifies whether the variable will be automatically initialized to zero or to any indeterminate value

FEATURE	STORAGE CLASS			
	Auto	Extern	Register	Static
Accessibility	Accessible within the function or block in which it is declared	Accessible within all program files that are a part of the program	Accessible within the function or block in which it is declared	Local: Accessible within the function or block in which it is declared Global: Accessible within the program in which it is declared
Storage	Main Memory	Main Memory	CPU Register	Main Memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Exists throughout the execution of the program	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Local: Retains value between function calls or block entries Global: Preserves value in program files
Default value	Garbage	Zero	Garbage	Zero

```
// FILE 1.C

#include<stdio.h>
#include<FILE2.C>
int x;
void print(void);
int main()
{
     x = 10;
     print();
     return 0;
}

// END OF FILE1.C
```

```
// FILE2.C

#include<stdio.h>

extern x;
void print()
{
     printf("\n x = %d", x);
}
main()
{
// Statements
}
// END OF FILE2.C
```

```
#include<stdio.h>
void print(void);
int main()
{
     printf("\n First call of print()");
     print();
     printf("\n\n Second call of print()");
     print();
     printf("\n\n Third call of print()");
     print();
     return 0;
}

void print()
{
     static int x;
     int y = 0;
     printf("\n Static integer variable, x = %d", x);
     printf("\n Integer variable, y = %d", y);
     x++;
     y++;
}

Output:

First call of print()
Static integer variable, x = 0
Integer variable, y = 0

Second call of print()
Static integer variable, x = 1
Integer variable, y = 0

Third call of print()
Static integer variable, x = 2
Integer variable, y = 0
```

RECURSIVE FUNCTIONS

- A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases, they are

  base case, in which the problem is simple enough to be solved directly without making any further calls to the same function

  recursive case, in which first the problem at hand is divided into simpler sub parts. Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem. In recursive function, complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

FINDING FACTORIAL OF A NUMBER USING RECURSION

PROBLEM	SOLUTION
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

Base case is when n=1, because if n = 1, the result is known to be 1

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as

factorial(n) = n X factorial (n-1)

```
#include<stdio.h>
int Fact(int)
{       if(n==1)
                retrun 1;
        return (n * Fact(n-1));
}
main()
{       int num;
        scanf("%d", &num);
        printf("\n Factorial of %d = %d", num, Fact(num));
        return 0;
}
```

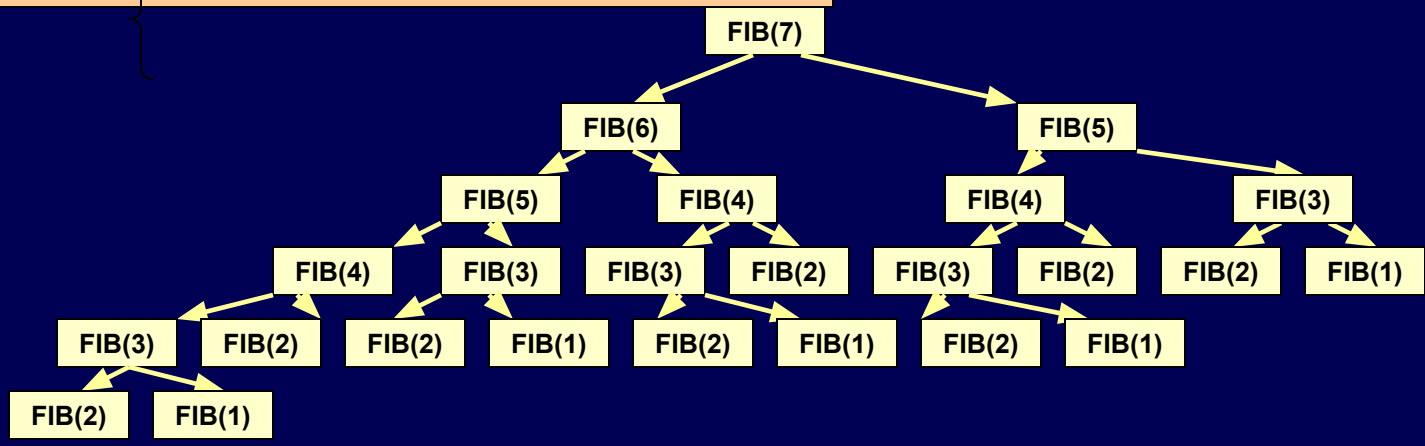
THE FIBONACCI SERIES

- The Fibonacci series can be given as:

  0 1 1 2 3 5 8 13 21 34 55……

- That is, the third term of the series is the sum of the first and second terms. On similar grounds, fourth term is th sum of second and third terms, so on and so forth. Now we will design a recursive solution to find the nth term o the Fibonacci series. The general formula to do so can be given as

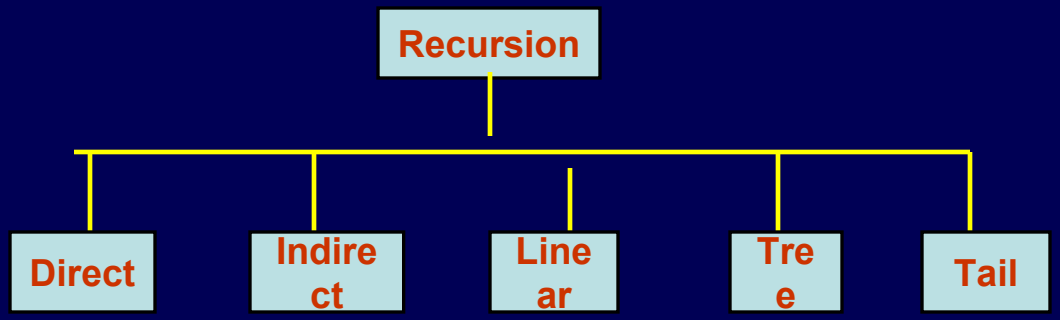
$$FIB(n) = \begin{cases} 1, & \text{if } n<=2 \\ FIB(n-1) + FIB(n-2), & \text{otherwise} \end{cases}$$




```
main()
{       int n;
        printf("\n Enter the number of terms in the series : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
                printf("\n Fibonacci (%d) = %d", i, Fibonacci(i));
}
int Fibonacci(int num)
{       if(num <= 2)
                return 1;
        return ( Fibonacci (num – 1) + Fibonacci(num – 2));
}
```

TYPES OF RECURSION

- Any recursive function can be characterized based on:
  whether the function calls itself directly or indirectly (direct or indirect recursion).
  whether any operation is pending at each recursive call (tail-recursive or not).
  the structure of the calling pattern (linear or tree-recursive).



DIRECT RECURSION

A function is said to be *directly* recursive if it explicitly calls itself. For example, consider the function given below.

```
int Func( int n)
{
      if(n==0)
            retrun n;
      return (Func(n-1));
}
```

INDIRECT RECURSION

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
```

```
int Func2(int x)
{
        return Func1(x-1);
}
```

TAIL RECURSION

- A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calli function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in the case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
  return Fact1(n, 1);
}
```

```
int Fact1(int n, int res)
{
     if (n==1)
         return res;
     return Fact1(n-1, n*res);
}
```

LINEAR AND TREE RECURSION

- Recursive functions can also be characterized depending on the way in which the recursion grows- in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function. For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.
- On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

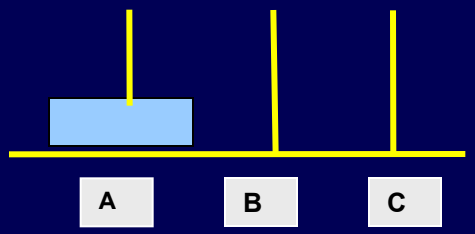
```
int Fibonacci(int num)
{
        if(num <= 2)
                return 1;
        return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

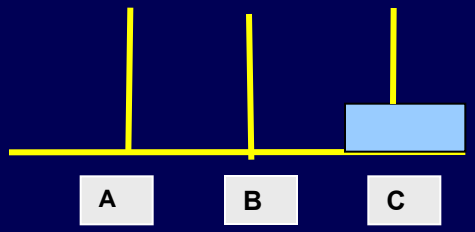
PROS AND CONS OF RECURSION

- Pros: Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use
- Recursion represents like the original formula to solve a problem.
- Follows a divide and conquer technique to solve problems
- In some (limited) instances, recursion may be more efficient
- Cons: For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.
- It is difficult to find bugs, particularly when using global variables

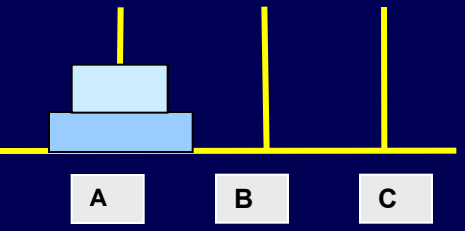
TOWER OF HANOI

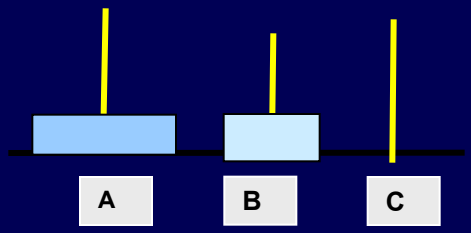
- **Tower of Hanoi is one of the main applications of a recursion. It says, "if you can** solve ***n-1*** **cases, then you can easily solve the** ***nth*** **case?"**

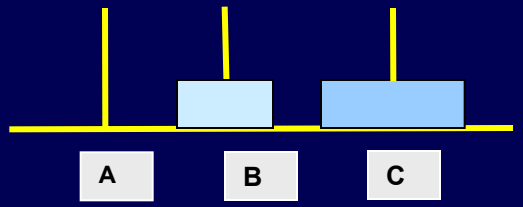


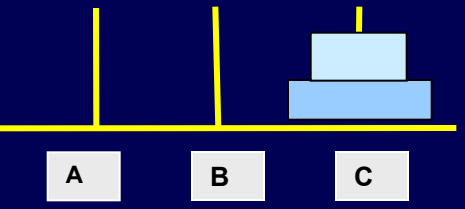


If there is only one ring, then simply move the ring from source to the destination









If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from the source to the destination

- **Consider the working with three rings.**