C++ FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat** to concatenate two strings, function **memcpy** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

Defining a Function:

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called **max**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers

int max(int num1, int num2)
{
   // local variable declaration  int result;

   if (num1 > num2)  result = num1;
else
   result = num2;

return result;
}
```

Function Declarations:

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max, following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke thatfunction.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>  using namespace std;

// function declaration
int max(int num1, int num2);

int main ()
{
   // local variable declaration:
   int a = 100;  int b = 200;  int ret;

   // calling a function to get max value.  ret = max(a, b);

   cout << "Max value is : " << ret << endl;  return 0;
}

// function returning the max between two numbers  int max(int num1, int num2)
{
   // local variable declaration  int result;

   if (num1 > num2)  result = num1;
else
   result = num2;
```

```
    return result;
 }
```

I kept max function along with main function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max function used the same method.

Default Values for Parameters:

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>  using namespace std;

int sum(int a, int b=20)
{
  int result;

  result = a + b;  return (result);
}

int main ()
{
   // local variable declaration:
```

```
   int a = 100;  int b = 200;  int result;

   // calling a function to add the values.  result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.  result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300  Total value is :120
```

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

Recursive Function

- Afunction is called recursive if it calls itself
- In C, all functions can be used recursively
- Example:

```c
#include <stdio.h>

int main(void)
{
  printf("The universe is never ending\n");
  main();
  return 0;
}
```


– *This will act like an infinite loop*

Recursive Function: Example

- This code computes the sum of first n positive integers.
- For n = 4

```
int sum(int n)
{
    if(n <= 1)
        return n;
    else
        return (n+sum(n-1));
}
```

Function Call	Value returned
sum(1)	1
sum(2)	2+sum(1) or 2+1
sum(3)	3+sum(2) or 3+2+1
Sum(4)	4+sum(3) or 4+3+2+1

Recursive Function

- There is a base case (or cases) that is tested upon entry
- And a general recursive case

– in which one of the variables, is passed as an argument in such a way as to ultimately lead to the base case.

```
int sum(int n)
{
    if(n <= 1)
        return n;
    else
        return (n+sum(n-1));
}
```

Problems Defined Recursively

- There are many problems whose solution can be defined recursively

Example: *factorial n*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases}$$

(*closed form* solution)

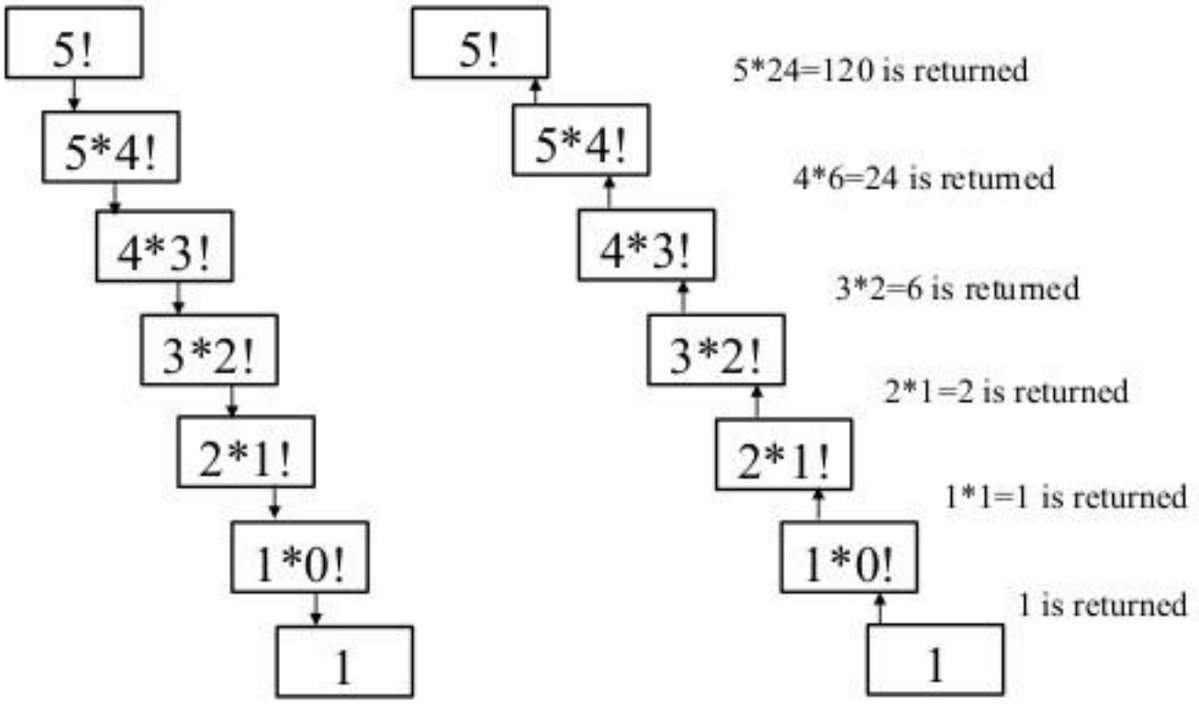
Coding the Factorial Function

- Recursive Implementation

```
int Factorial(int n)
{
 if (n==0)   // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```

- For n > 12 this function will return incorrect value as the final result is too big to fit in an integer

Trace of Recursion: Factorial



Coding the Factorial Function (cont.)

- Iterative Implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

return fact;
}
```

- Both recursive and iterative version returns same value

Another Example: *n choose k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad 1 < k < n \qquad \textit{(recursive solution)}$$

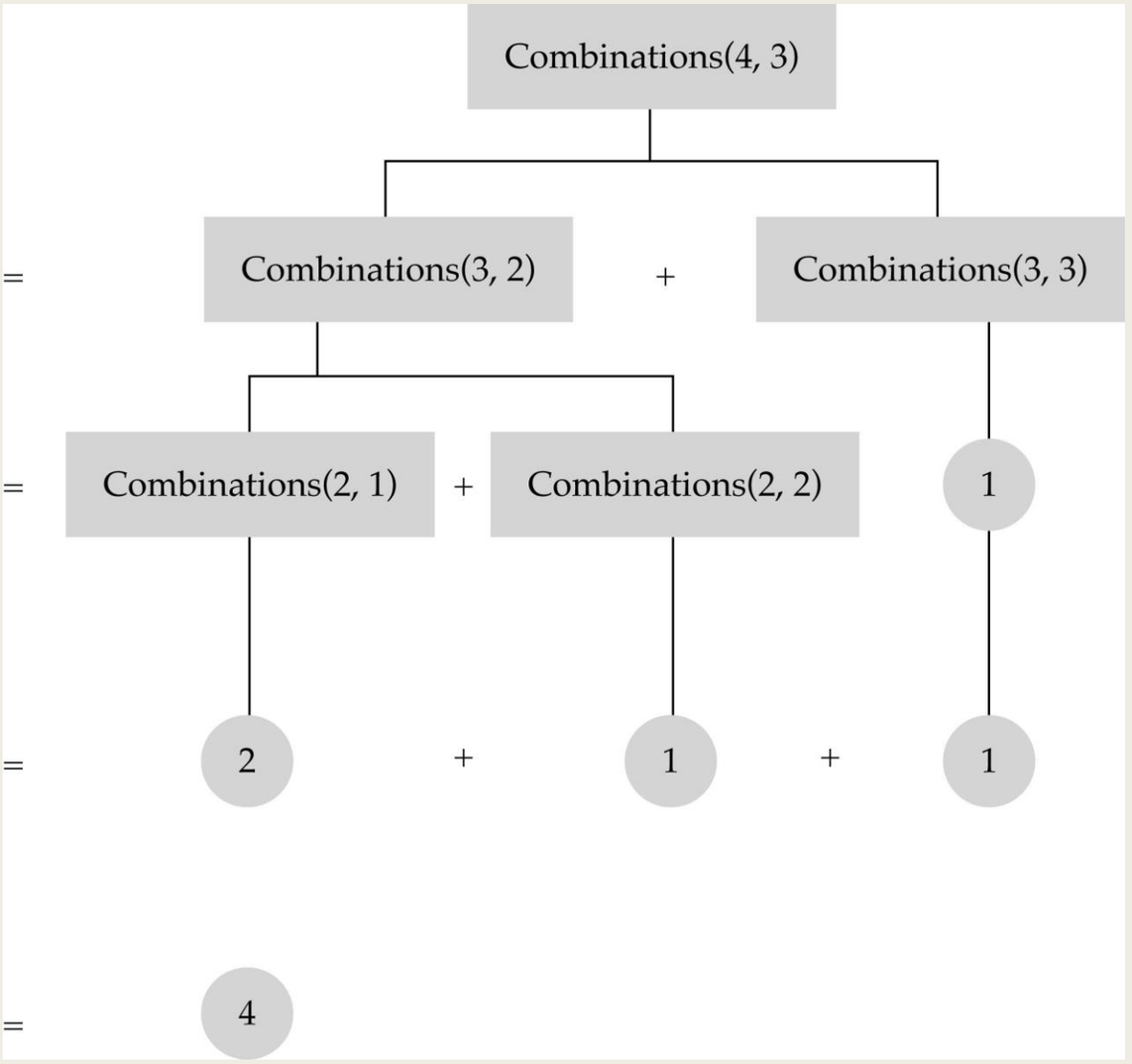
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \ (k = 1), \quad \binom{n}{n} = 1 \ (k = n)$$

n choose k implementation

```
int Combinations(int n, int k)
{
 if(k == 1)   // base case 1
   return n;
 else if (n == k)   // base case 2
   return 1;
 else
   return(Combinations(n-1, k) +
  Combinations(n-1, k-1));
 }
```


Combinations(4, 3)
=
Combinations(3, 2)
+
Combinations(3, 3)
=
Combinations(2, 1)
+
Combinations(2, 2)
1
=
2
+
1
+
1
=
4

Recursion vs Iteration

- Iteration can be used in place of recursion
  - *An iterative algorithm uses a looping construct*
  - *A recursive algorithm uses a branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How to write a recursive function?

- Determine the size factor
- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (use the "Three-Question-Method")

Three Question Verification

1. The Base-Case Question

– *Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?*

2. The Smaller-Caller Question

– *Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?*

3. The General-Case Question

– *Assuming that the recursive call(s) work correctly, does the whole function work correctly?*

Recursion: Calculation of Fibonacci Sequence

- Recursive solution

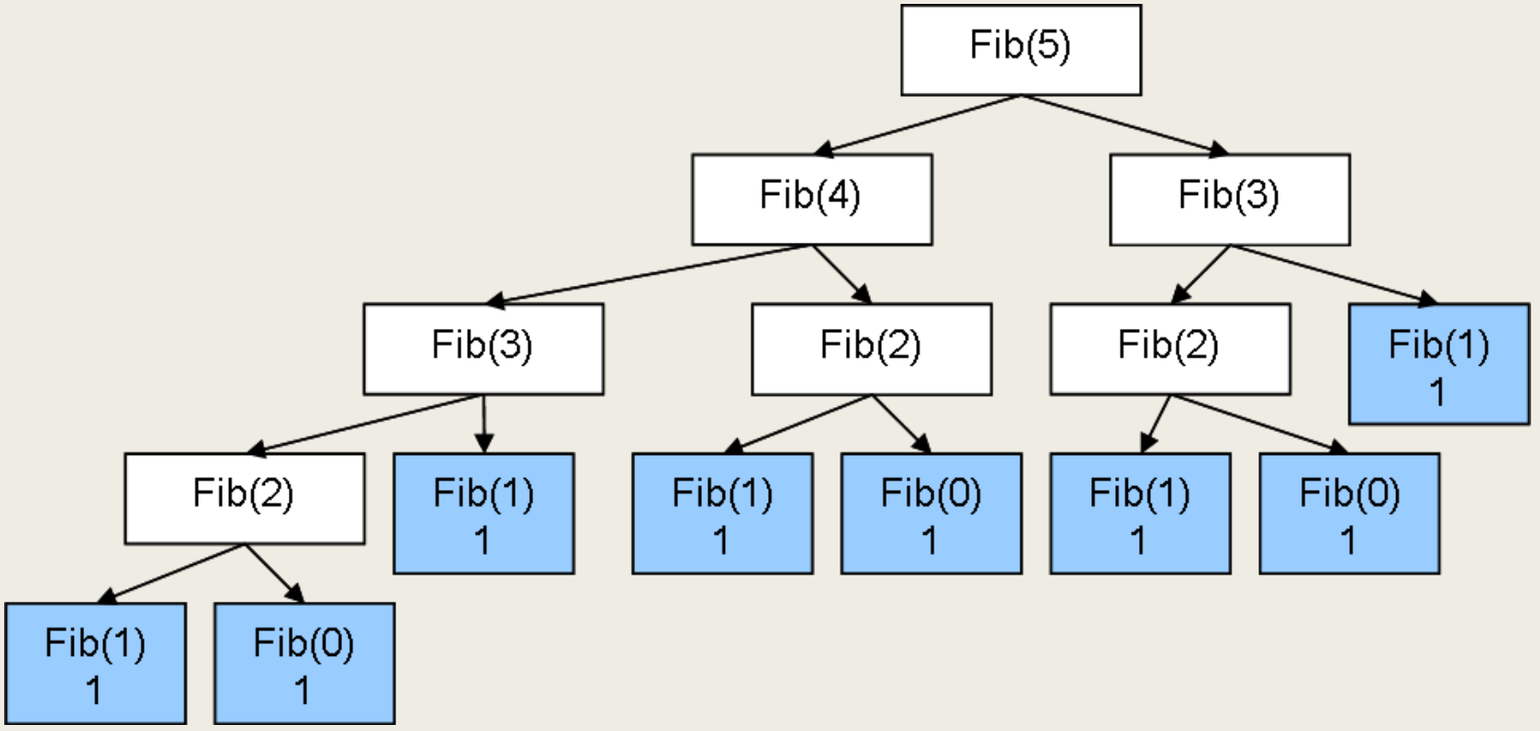
  $f_0 = 0$, $f_1 = 1$, $f_i = f_{i-1} + f_{i-2}$, for $i = 1, 2, \ldots$

  – *Except for* f_0 and f_1, *every element in the sequence is the sum of the previous two elements*

- The sequence begins 0, 1, 1, 2, 3, 5, 8, …

```
int Fibonacci(int n)
{
 if(n <= 1)   // base case
   return n;
else
   return(Fibonacci(n-1) + Fibonacci(n-2));
}
```

Recursion: Calculation of Fibonacci Sequence



Number of Function Calls for Recursive Fibonacci

Value of n	Value of Fibonacci(n)	#of function calls
0	0	1
1	1	1
2	1	3
…	…	…
23	28657	92735
24	46368	150049
…	…	…
42	267914296	866988873
43	433494437	1402817465

A large number of function calls is required to compute the nth Fibonacci for even moderate values of n

Pitfalls of Recursion

- Missing base case – failure to provide an escape case.
- No guarantee of convergence – failure to include within a recursive function a recursive call to solve a subproblem that is not smaller.
- Excessive space requirements - a function calls itself recursively an excessive number of times before returning; the space required for the task may be prohibitive.
- Excessive recomputation – illustrated in the recursive Fibonacci method which ignores that several sub-Fibonacci values have already been computed.

```
    }
    cout<<i<<endl;
    }.

for (int i=0; i<40; i++)
    {
        if(i==2)
        {
            continue;
        }
        cout<<i<<endl;
        }.
```

Function and Recursion

Sometimes our program gets bigger in size and it is not possible for a programmer to track which piece of code is doing what.

function is a way to break over code into chunks so that it is not possible for a programmer to reuse them.

* What is function?

A function is a block of code which performs a particular task.

A function can be reused by the programmer in a program any no. of times.

eg and syntax of function:

```
#include <iostream>

void display();  => function prototype.

int main(){
    int a;
    display();  => function call
    return 0;
}

void display(){  -> function definition
    cout<<"Hi, I am display";
}
```

* function prototype:

function prototype is a way to tell the compiler about the function we are going to define in the program.
Here void indicates that the function returns nothing.

* function Call:

function Call is a way to tell the compiler to execute the function body at the time the call is made.
Note that the program execution starts from the main function in the sequence the instruction are written.

* Function defination:

This part contains the exact set of instruct which are executed during the function Call when a function called from main(), the suspended. During this time the control goes to the function being called. When the function body is done executing main() resume

* Important Points:

→ Execution of a C++ program starts from main().
→ A C++ program can have more than one functio
→ Every function gets called directly or indirectl from main().
→ There are two types of function in C++. Let's talk about them.

* Types of function:

1. Library function: Commonly required grouped together in a libary file on disk.
2. User defined function: These are the function declared and defined by the user.

* Why use function?

→ To avoid rewritting the same logic again and ag
→ To keep track of what are doing in a Progra
→ To test and check logic independently.

* Passing values to functions:

We can pass values to a function and can get a value in return of a function.

```
int sum(int a, int b)
```

The above prototype means that sum is a function which takes values a (of type int) and b (of type int) and return a value of type int.

function defination of sum can be:

```
int sum(int a, int b){
    int c;
    c = a+b;   => a and b are parameters
    return c;
}
```

Now we can call sum(2,3); from main to get 5 in return.

=> 2 & 3 are arguments

```
int d = sum(2,3);   => d becomes 5
```

Note:

(1) Parameters are the values or variable placeholder in the function defination. eg: a & b.

(II) Arguments are the actual values passed to the function to make a call. eg:- 2, 33.

(III) A function can return only one value at a time.

(IV) If the passed variable is changed inside the function, the function call does not change the value in the calling function.

```
int change(int a) {
    a=77                    => Mismatch.
    return 0;
}
```

change is a function which changes a to 77. No if we call it from main like this

```
int b=22
change(b);   => The value of b remains 22
cout << "b is " << b;  prints b is 22.
```

This happens because a copy of b is passed to the change function.

```
#include<iostream>

using namespace std;
```

Function Prototype

```
type function-name (arguments);
int sum(int a, int b); ---> Acceptable
int sum(int a, b); ---> Not-Acceptable.
int sum(int, int); ---> Acceptable.
void g(void); ---> Acceptable
void g(); ---> Acceptable

int main(){
int num1, num2;
  Cout<<"Enter the 1st num: ";
  Cin>>num1;
Cout<<"Enter the 2nd num: ";
  Cin>>num2;

// num1 & num2 are actual parameters.

Cout<<"the sum is "<<sum(num1, num2);
 g();
   return 0;
}

int sum(int a, int b){
// formal parameters a and b will be taking
values from actual parameter num1 and num2
  int c=a+b;
  return c;
}
void g(){
 Cout<<"\n Hellow, Good morning";
}
```

* Call by value in C++ :

Call by value is a method in C++ to pass the values to the function argument. In case of call by value the copies of actual parameters are sent to the formal parameters which means that if we change the value inside the function that will not affect the actual values.

* Call by Reference :

Call by reference is a method in C++ to pass the values to the function argument. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the value inside the function that will affect the actual values.

```
#include <iostream>
using namespace std;
int sum(int a, int b) {
    int c = a + b;
    return c;
}
```

This will not swap a and b.

```
void swap(int a, int b){          // temp  a  b
    int temp = a;                 // 4     4  5
    a = b;                        // 4     5  5
    b = temp;                     // 4     5  4
}
```

// Call by reference using pointers.

```
void swapPointer(int* a, int* b){   // temp  a  b
    int temp = *a;                  // 4     4  5
    *a = *b;                        // 4     5  5
    *b = temp;                      // 4     5  4
```

* Recursion:

A function defined in C++ can call itself. This is called recursion.
A function calling itself is also called recursive function.

⇒ Eg of Recursion:

A very good exam of recursion is factorial.

factorial(n) = 1x2x3 --- xn.
factorial(n) = 1x2x3 --- n-1 x n
factorial = factorial(n-1) x n

Since we can write factorial of a number in terms of itself, we can program it using recursion.

```
int factorial(int x){
    int f;
    if (x==0 || x==1)
        return 1;
    else
        f = x * factorial(x-1)
    return f;
    }
```

⇒ A program to Calculate factorial using recursion

How does it work:

factorial(5)

5 x factorial(4)

5 x 4 x factorial(3)

5 x 4 x 3 x factorial(2)

5 x 4 x 3 x 2 x factorial(1)

5 x 4 x 3 x 2 x 1.

* Notes

→ Recursion is sometimes the most direct way to code an algorithm.

→ The condition which does not call the function further in a recursive function is called as the base condition.

→ Sometimes, due to a mistake made by the programmer a recursive function can keep running without resulting in a memory error.

```
#include<iostream>
using namespace std;

int fib(int n){

    if(n<2){
        return 1;
    }
    return fib(n-2) + fib(n-1);
}
```

* fib(5)
  fib(4) + fib(3)
  fib(2) + fib(3) + fib(2) + fib(3)

```
int factorial(int n){
    if(n<=1){
        return 1;
    }
    return n* factorial(n-1);
    }

int main(){
```



```
int a;
cout<<"Enter a number"<<endl;
cin>>a;
cout<<"the factorial of"<<a<<"is"<<factorial(a);
cout<<"the term in fabonacci seq. at position"<<a<<
    "is"<<fib(a)<<endl;
    return 0; }
```