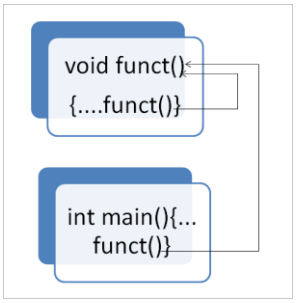
What Is Recursion?

Recursion is a process in which a function calls itself. The function that implements recursion or calls itself is called a Recursive function. In recursion, the recursive function calls itself over and over again and keeps on going until an end condition is met.

The below image depicts how Recursion works:

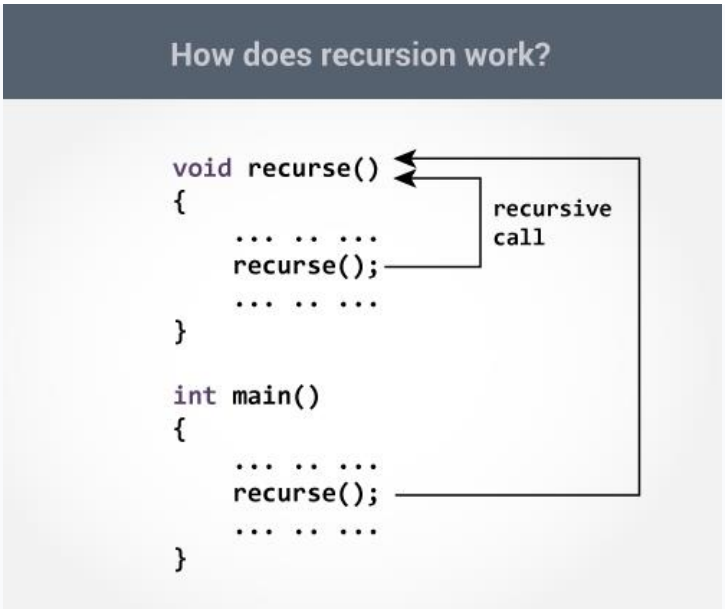


A function that calls itself is known as recursive function. And, this technique is known as recursion.

How recursion works in C++?

```
void recurse()
{
   ... .. ...
   recurse();
   ... .. ...
}

int main()
{
   ... .. ...
   recurse();
   ... .. ...
}
```

The figure below shows how recursion works by calling itself over and over again.



The recursion continues until some condition is met.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

Example 1: Factorial of a Number Using Recursion

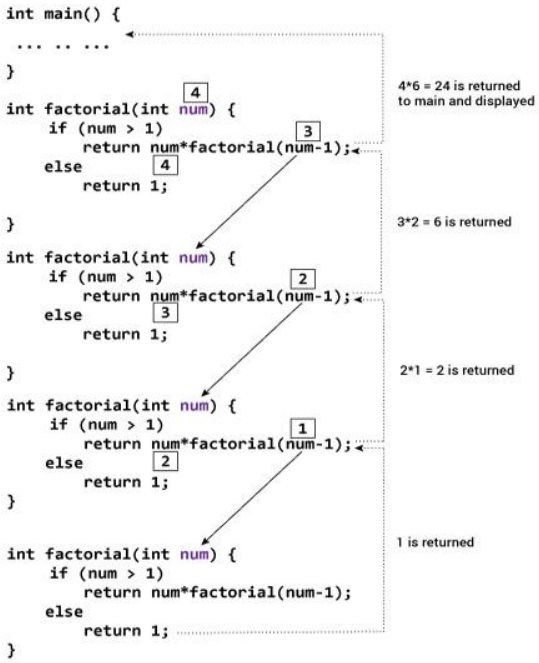
```
// Factorial of n = 1*2*3*...*n

#include <iostream>
using namespace std;

int factorial(int);

int main()
{
   int n;
   cout<<"Enter a number to find factorial: ";
   cin >> n;
   cout << "Factorial of " << n <<" = " << factorial(n);
   return 0;
}

int factorial(int n)
{
   if (n > 1)
   {
      return n*factorial(n-1);
   }
   else
   {
```

```
        return 1;
    }
}
```

Output

Enter a number to find factorial: 4
Factorial of 4 = 24

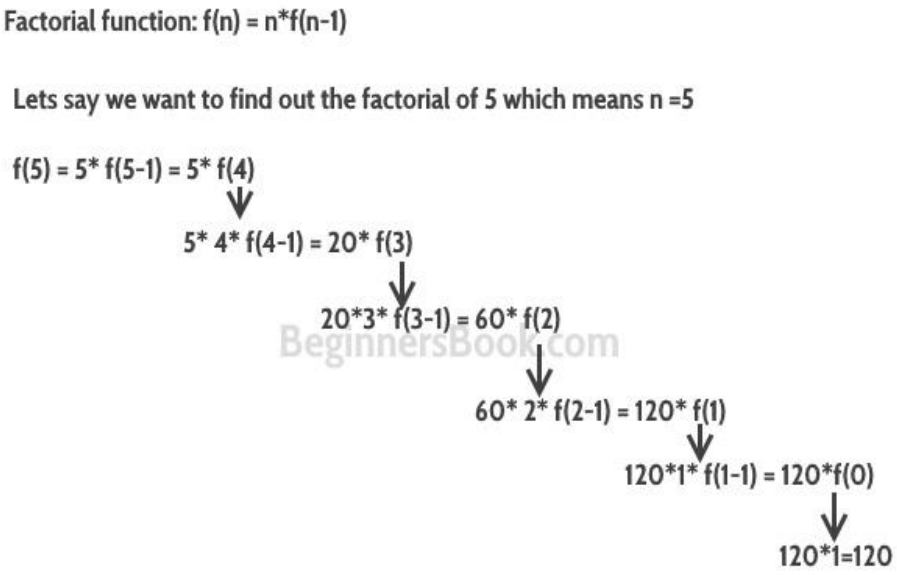
Explanation



Suppose the user entered 4, which is passed to the factorial() function.

1. In the first factorial() function, test expression inside if statement is true. The return num*factorial(num-1); statement is executed, which calls the second factorial() function and argument passed is num-1 which is 3.
2. In the second factorial() function, test expression inside if statement is true. The return num*factorial(num-1); statement is executed, which calls the third factorial() function and argument passed is num-1 which is 2.
3. In the third factorial() function, test expression inside if statement is true. The return num*factorial(num-1); statement is executed, which calls the fourth factorial() function and argument passed is num-1 which is 1.

4. In the fourth factorial() function, test expression inside if statement is false. The return 1; statement is executed, which returns 1 to third factorial() function.
5. The third factorial() function returns 2 to the second factorial() function.
6. The second factorial() function returns 6 to the first factorial() function.
7. Finally, the first factorial() function returns 24 to the main() function, which is displayed on the screen.




```
#include <iostream>
using namespace std;
//Factorial function
int f(int n){
  /* This is called the base condition, it is very important to specify the base condition in recursion, otherwise your program will throw stack overflow error.
   */
  if (n <= 1)
      return 1;
  else
     return n*f(n-1);
}
int main(){
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  cout<<"Factorial of entered number: "<<f(num);
  return 0;
}
```

Output:

```
Enter a number: 5
```

Factorial of entered number: 120

Base condition

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if (n <= 1)
    return 1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

Direct recursion vs Indirect recursion

Direct recursion: When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

Indirect recursion: When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

```
// An example of direct recursion
void directRecFun()
{
   // Some code....

   directRecFun();

   // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
   // Some code...

   indirectRecFun2();

   // Some code...
```

```
}
void indirectRecFun2()
{
   // Some code...

   indirectRecFun1();

   // Some code...
}
```

Indirect Recursion

```
#include <iostream>
using namespace std;
int fa(int);
int fb(int);
int fa(int n){
  if(n<=1)
    return 1;
  else
    return n*fb(n-1);
}
int fb(int n){
  if(n<=1)
    return 1;
  else
    return n*fa(n-1);
}
int main(){
  int num=5;
  cout<<fa(num);
  return 0;
}
```

Output:
120

Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
   // wrong base case (it may cause
   // stack overflow).
   if (n == 100)
      return 1;

   else
      return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Let us take the example how recursion works by taking a simple function.

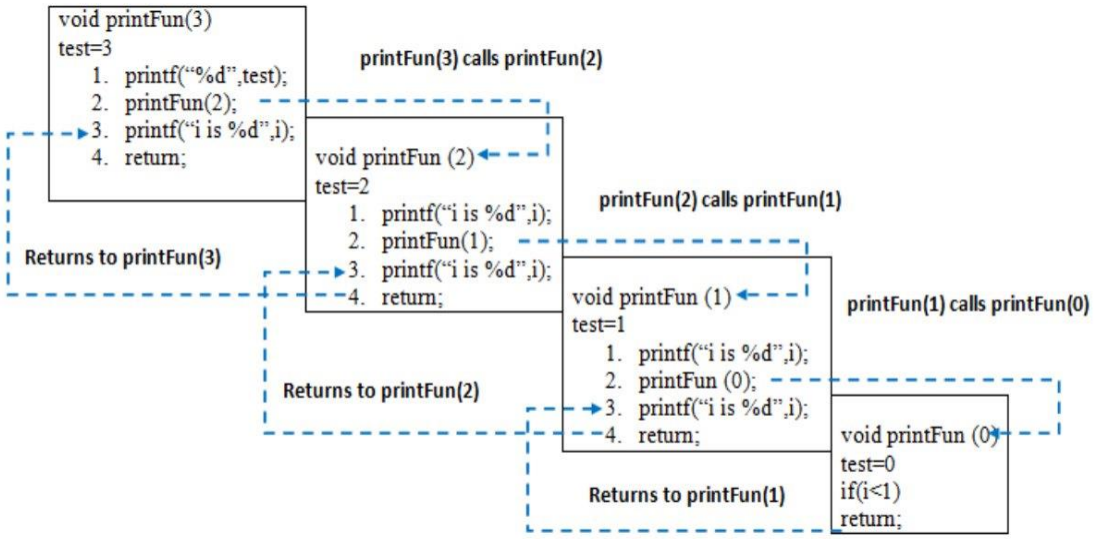
```
// A C++ program to demonstrate working of  recursion
#include <bits/stdc++.h>
using namespace std;

void printFun(int test)
{
   if (test < 1)
      return;
   else {
      cout << test << " ";
      printFun(test - 1); // statement 2
      cout << test << " ";
      return;
   }
```

```
}

int main()
{
   int test = 3;
   printFun(test);
}
```

Output :

3 2 1 1 2 3

When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, printFun(2) is called and memory is allocated to printFun(2) and a local variable test is initialized to 2 and statement 1 to 4 are **pushed in the stack**. Similarly, printFun(2) calls printFun(1) and printFun(1) calls printFun(0). printFun(0) goes to if statement and it return to printFun(1). Remaining statements of printFun(1) are executed and it returns to printFun(2) and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Few practical problems which can be solved by using recursion and understand its basic working.

Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2 .

Mathematical Equation:

n if n==0, n==1;

fib(n) = fib(n-1) + fib(n-2) otherwise;

Recurrence Relation:

T(n) = T(n-1) + T(n-2) + O(1)

Recursive program:

Input: n = 5

Output:

Fibonacci series of 5 numbers is : 0 1 1 2 3

Implementation:

```
// C code to implement Fibonacci series
#include <stdio.h>

// Function for fibonacci
int fib(int n)
{
   // Stop condition
   if (n == 0)
      return 0;

   // Stop condition
   if (n == 1 || n == 2)
      return 1;

   // Recursion function
   else
      return (fib(n - 1) + fib(n - 2));
}
int main()
{
   // Initialize variable n.
   int n = 5;
   printf("Fibonacci series  of %d numbers is: ", n);

   // for loop to print the fiboancci series.
   for (int i = 0; i < n; i++) {
      printf("%d ", fib(i));
   }
   return 0;
}
```

Output:

Fibonacci series of 5 numbers is : 0 1 1 2 3

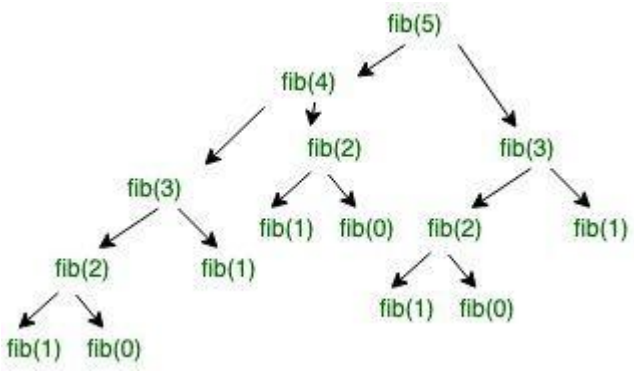
Here is the recursive tree for input 5 which shows a clear picture of how a big problem can be solved into smaller ones.
fib(n) is a Fibonacci function. The time complexity of the given program can depend on the function call.

fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n*O(1) -> T(n) = O(2^n)

For Best Case.

T(n) = ?(2^n\2)

Working:



Problem 2: Write a program and recurrence relation to find the Factorial of n where n>2 .
Mathematical Eqution:

1 if n=0 or 1;

f(n) = n*f(n-1) if n>1;

Recurrence Relation:

T(n)=1 for n=0

T(n)=1+T(n-1) for n>0

Recursive Program:
Input: n = 5
Output:
factorial of 5 is: 120

Implementation:

```
// C code to implement factorial
#include <stdio.h>

// Factorial function
int f(int n)
{
   // Stop condition
   if (n == 0 || n == 1)
      return 1;

   // Recursive condition
   else
      return n * f(n - 1);
}

// Driver method
int main()
{
   int n = 5;
   printf("factorial of %d is: %d",
         n, f(n));
   return 0;
}
```

Output:

factorial of 5 is: 120

Working:

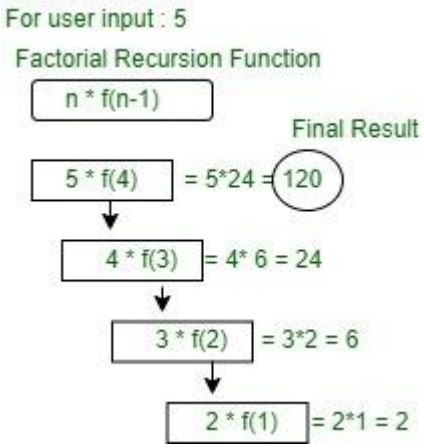


Diagram of factorial Recursion function for user input 5.

What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, **Tower of Hanoi**, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer **Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi**.

Pros/Cons Of Recursion Over Iterative Programming

Recursive programs provide compact and clean code. A recursive program is a simple way of writing programs. There are some inherent problems like factorial, Fibonacci sequence, towers of Hanoi, tree traversals, etc. which require recursion for solving.

In other words, they are solved efficiently with recursion. They can also be solved with iterative programming using stacks or other data structures but there are chances to become more complex to implement.

Problem-solving powers of recursive as well as iterative programming are the same. However, recursive programs take more memory space as all the function calls need to be stored on the stack until the base case is matched.

Recursive functions also have a time overhead because of too many function calls and return values.

Stack Overflow In Recursion

When recursion continues for an unlimited amount of time, it can result in a stack overflow.

When can recursion continue like this? One situation is when we do not specify the base condition. Another situation is when the base condition is not reached while executing a program.

For Example, we modify the above factorial program and change its base condition.

```
int factorial(int n){
  if(n ==1000)
    return 1;
```

```
  else
    return n*factorial(n-1);
}
```

In the above code, we have changed the base condition to (n==1000). Now, if we give the number n = 10, we can conclude that the base condition will never reach. This way at some point, the memory on the stack will be exhausted thereby resulting in a stack overflow.

Hence, while designing recursive programs, we need to be careful about the base condition we provide.

What is difference between tailed and non-tailed recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function.

What is tail recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function print() is tail recursive.

```
// An example of tail recursive function
void print(int n)
{
   if (n < 0)  return;
   cout << " " << n;

   // The last executed statement is recursive call
   print(n-1);
}
```

Why do we care?

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

Can a non-tail recursive function be written as tail-recursive to optimize it?

Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n)

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

```
// A NON-tail-recursive function. The function is not tail  recursive because the value returned by fact(n-
//1) is used in  fact(n) and call to fact(n-1) is not the last thing done by fact(n)

unsigned int fact(unsigned int n)
{
        if (n == 0) return 1;

        return n*fact(n-1);
}

// Driver program to test above function
int main()
{
        cout << fact(5);
        return 0;
}
```

Output :

120

The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in second argument. When n reaches 0, return the accumulated value.

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

// A tail recursive function to calculate factorial
unsigned factTR(unsigned int n, unsigned int a)
{
        if (n == 0) return a;

        return factTR(n-1, n*a);
}

// A wrapper over factTR
unsigned int fact(unsigned int n)
{
return factTR(n, 1);
}

// Driver program to test above function
int main()
{
        cout << fact(5);
        return 0;
}
```

Output :

120

Tail Call Elimination

We have discussed (in tail recursion) that a recursive function is tail recursive if recursive call is the last thing executed by the function.

```
// An example of tail recursive function
void print(int n)
{
        if (n < 0)
        return;
        cout << " " << n;

        // The last executed statement is recursive call
        print(n-1);
}
```

We also discussed that a tail recursive is better than non-tail recursive as tail-recursion can be optimized by modern compilers. Modern compiler basically do tail call elimination to optimize the tail recursive code.

If we take a closer look at above function, we can remove the last call with goto. Below are examples of tail call elimination.

```
// Above code after tail call elimination
void print(int n)
{
start:
        if (n < 0)
        return;
        cout << " " << n;

        // Update parameters of recursive call
        // and replace recursive call with goto
        n = n-1
        goto start;
}
```

QuickSort is also tail recursive (Note that MergeSort is not tail recursive, this is also one of the reason why QuickSort performs better)

```
/* Tail recursive function for QuickSort
```

```
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
	if (low < high)
	{
		/* pi is partitioning index, arr[p] is now
		at right place */
		int pi = partition(arr, low, high);

		// Separately sort elements before
		// partition and after partition
		quickSort(arr, low, pi - 1);
		quickSort(arr, pi + 1, high);
	}
}
```

The above function can be replaced by following after tail call elimination.

```
/* QuickSort after tail call elimination arr[] --> Array to be sorted, low --> Starting index, high --> Ending
index */

void quickSort(int arr[], int low, int high)
{
start:
	if (low < high)
	{
		/* pi is partitioning index, arr[p] is now at right place */
		int pi = partition(arr, low, high);

		// Separately sort elements before  partition and after partition
		quickSort(arr, low, pi - 1);

		// Update parameters of recursive call and replace recursive call with goto
		low = pi+1;
		high = high;
		goto start;
	}
}

void recurse()
{
  recurse(); //Function calls itself
}

int main()
{
```

```
  recurse(); //Sets off the recursion
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash.

Let us write a program to see how many times the function is called before the program terminates.

```
#include <iostream>

using namespace std;

void recurse ( int count ) // Each call gets its own count
{
  cout<< count <<"\n";
  /* It is not necessary to increment count since each function's  variables are separate (so each count
will be initialized one greater) */
  recurse ( count + 1 );
}

int main()
{
  recurse ( 1 ); //First function call, so it starts at one
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count + 1. Keep in mind, it is not a function restarting itself, it is hundreds of functions that are each unfinished with the last one calling a new recurse function.