Introduction to Recursion and its types

hitbullseye

Recursion:

- Recursion is a routine that calls itself again and again directly or indirectly. There are two types of recursions in the C language **Direct calling** and **Indirect calling**. The calling refers to the recursive call. The recursion is possible in C language by using method and function. The problems like the Tower of Hanoi, the Fibonacci series, and the nth derivative can be solved using recursion. The recursion uses a stack to store its calls in memory.

What is Recursion in C?

- Recursion, in general, can be defined as the repetition of a process in a similar way until the specific condition reaches. In C Programming, if a function calls itself from inside, the same function is called recursion. The function which calls itself is called a recursive function, and the function call is termed a recursive call. The recursion is similar to iteration but more complex to understand. If the problem can be solved by recursion, that means it can be solved by iteration. Problems like sorting, traversal, and searching can be solved using recursion. While using recursion, make sure that it has a base (exit) condition; otherwise, the program will go into the infinite loop.

- The recursion contains two cases in its program body.

Base case: When you write a recursive method or function, it keeps calling itself, so the base case is a specific condition in the function. When it is met, it terminates the recursion. It is used to make sure that the program will terminate. Otherwise, it goes into an infinite loop.

Recursive case: The part of code inside the recursive function executed repeatedly while calling the recursive function is known as the recursive case.

Basic Syntax of Recursion

The syntax for recursion is:

- The call to recursive fun() from the main function is a standard call; recursive fun() contains another recursive fun(), thus this is a recursive call, and recursive fun() itself is a recursive function. The base case is the halting condition for the recursive function.

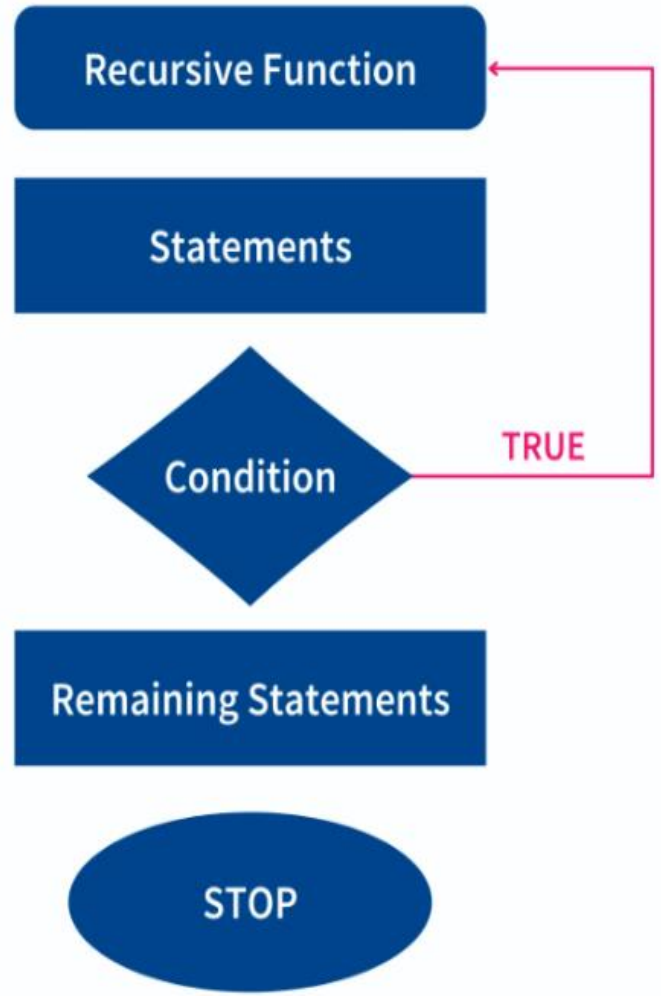
```
void recursive_fun() //recursive function
{
Base_case; // Stopping Condition

recursive_fun(); //recursive call
}

int main()
{
   recursive_fun(); //function call
}
```

Flowchart of Recursion

- The following diagram depicts a recursive function, inside of which is another recursive call that invokes the original recursive function until the underlying issue condition is satisfied. If the condition is true, the program will continue until the end of the statements until the condition is false.



How does Recursion work?

- The recursion is possible using a method or function in C language. The recursive function or method has two main parts in its body, i.e., the base case and the recursive case. While the recursive method is executed, first, the base case is checked by the program. If it turns out true, the function returns and quits; otherwise, the recursive case is executed. Inside the recursive case, we have a recursive call that calls the function inside which it is present.

- The representation of recursion in the program is as follows.

```
recursive_function()
{
    //base case
    if base_case = true;
    return;

    else
    //recursive case
    return code_for_recursion; //includes recursive call
}
```

Why Stack Overflow occurs in recursion?

- When the base case is not reached because of a mistake in the definition or because the base case is not specified, a stack overflow error might occur because the stack or memory has become full owing to repeated recursion. For Example:

So in this, if we pass the function with a value less than 1000, we will never reach any base condition so it will lead to a stack overflow error.

```
int fibo(int n) {
        if (n == 1000)
          return 0;

        return fibo(n - 1) + fibo(n - 2);
        }
```

Types of Recursions in C

- There are two types of recursions in the C language.
    1. Direct Recursion
    2. Indirect Recursion

Direct Recursion:

- Direct recursion in C occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions. Following is the structure of direct recursion.

- In the direct recursion structure, the function_01() executes, and from inside, it calls itself recursively.

```
function_01()
{
        //some code
        function_01();
        //some code
}
```

Indirect Recursion:

- Indirect recursion in C occurs when a function calls another function and if this function calls the first function again. Such functions are also called indirect recursive functions. Following is the structure of indirect recursion.

- In the indirect recursion structure, the function_01() executes and calls function_02(). After calling now, function_02 executes where inside it there is a call for function_01, which is the first calling function.

```
function_01()
{
        //some code
        function_02();
}

function_02()
{
    //some code
    function_01();
}
```

Difference Between Direct and Indirect Recursion

- If the calling function in recursion calls the same function itself, it is known as direct recursion, while in the case of indirect recursion, one function calls another function directly or indirectly.

Below is the example for both:-

```
// An example of direct recursion
void directRecursion()
{
 directRecFun();
}

// An example of indirect recursion
void indirectRecursion1()
{
 indirectRecFun2();
}

void indirectRecursion2()
{
 indirectRecFun1();
}
```

C Program Function to show direct recursion

- Here is a simple C program to print the Fibonacci series using direct recursion.

Code:

```c
#include<stdio.h>

int fibonacci_01(int i) {

  if (i == 0) {
    return 0;
  }

  if (i == 1) {
    return 1;
  }

  return fibonacci_01(i - 1) + fibonacci_01(i - 2);

}

int main() {

  int i, n;
  printf("Enter a digit for fibonacci series: ");
  scanf("%d", & n);

  for (i = 0; i < n; i++) {
    printf(" %d ", fibonacci_01(i));
  }

  return 0;
}
```

Output:

```
Enter a digit for fibonacci series: 8
 0  1  1  2  3  5  8  13
```

In the C program above, we have declared a function named fibonacci_01(). It takes an integer i as input and returns the ith element of the Fibonacci series. At first, the main() function will be executed where we have taken two variables i and n. We will take input from the user that will be stored in n, and the for loop will execute till n iteration where with each iteration, it will pass the parameter to the fibonacci_01() function where the logic for the Fibonacci series is written.

Now inside the fibonacci_01() function, we have nested if-else. If input = 0, it will return 0, and if the input = 1, it will return 1. These are the base cases for the Fibonacci function. If the value of i is greater than 1, then fibonacci(i) will return fibonacci_01 (i - 1) + fibonacci_01 (i -2) recursively, and this recursion will be computed till the base condition.

Here is a C program to print numbers from 1 to 10 in such a manner that when an odd no is encountered, we will print that number plus 1. When an even number is encountered, we would print that number minus 1 and will increment the current number at every step.

Code:

```
#include<stdio.h>
void odd();
void even();
int n=1;

void odd()
{
    if(n <= 10)
    {
        printf("%d ", n+1);
        n++;
        even();
    }
    return;
}
```

```
void even()
{
    if(n <= 10)
    {
        printf("%d ", n-1);
        n++;
        odd();
    }
    return;
}

int main()
{
    odd();
}
```

Output: 2 1 4 3 6 5 8 7 10 9

In this C program we have a function named odd() and even(). A variable n is assigned with a value 1 as we have to take values from 1 to 10. Now inside the odd() function, we have an if statement which states that if the value of n is less than or equals 10 add 1 to it and print. Then the value of n is incremented by 1(it becomes even), and the even() function is called. Now inside the even() function, we again have an if statement which states that if the value of n is less than or equals 10 subtract 1 from it and print.

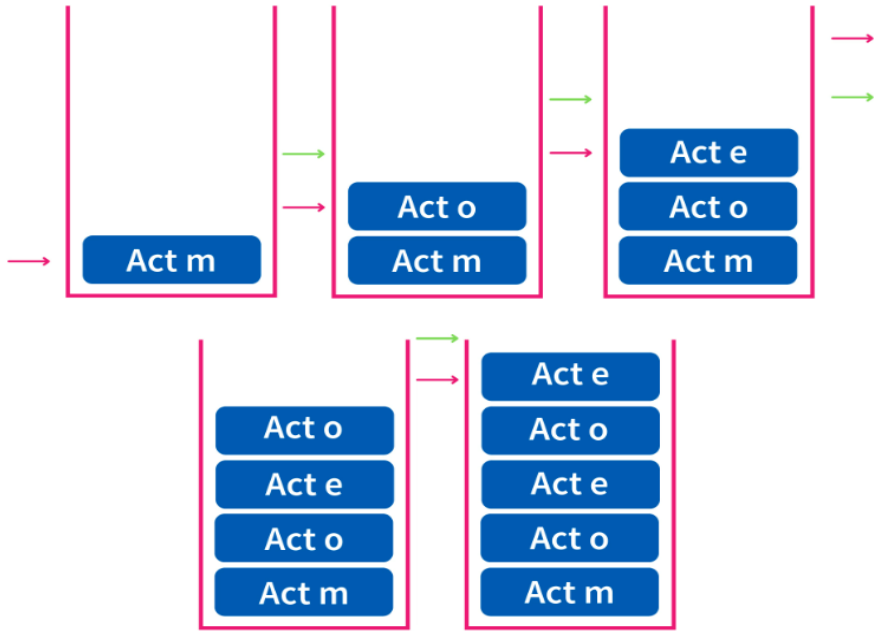
Then the value of n is incremented by 1(it becomes odd, and the odd() function is called. This indirect recursion goes on until the if condition inside both functions becomes unsatisfied. At last, we have the main() function inside, which we call the odd() function as the first number handle is 1, which is odd.

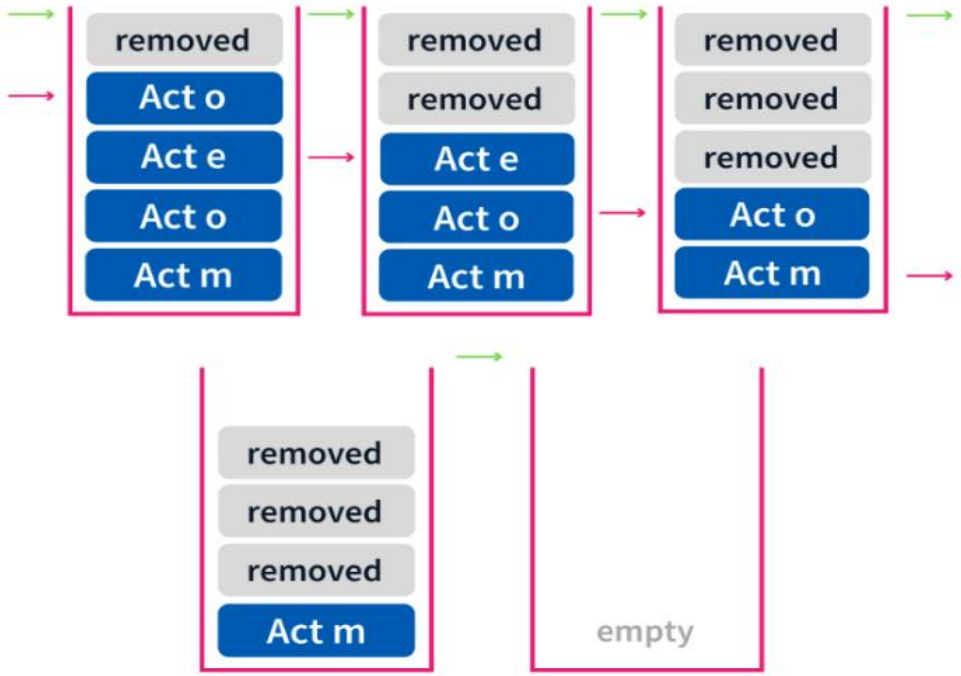
Now let's simulate this program using **stack** and the concept called **activation record** by which we could track program logic in respect of program stack.

In the following images:

- Act means "Activation Record"
- o means odd()
- e means even()
- m means main()

Act = Activation Record
o = odd ( )
e = even ( )
m = main ( )





Here, the activation record is denoted by Act, odd() is represented by o, even() by e, and main() is represented by m. Upon executing the program at first, the main() function is executed, which causes the activation record Act m to be stored in the stack. The main function calls the odd() function, so the activation record **Act o** is next added to the stack. Now inside odd() there is a call for even() so the activation record **Act e** is added in the stack and this process goes on until the base conditions inside the odd() and even() function are reached.

Now that the base conditions are met, the activation records are removed from the stack, and the value inside that activation record is returned, but in the above example, functions are void. They don't return any value.

Let's take a few more examples of recursion to understand it in a better way.

C Program to show infinite recursive function

Code:

```c
#include<stdio.h>

int main()
{
    printf("Scaler");
    main();

    return 0;
}
```

- In this program, there is a call for main() function from inside main() function. But we have not given exit conditions for the program. Therefore, the program will print 'Scaler' infinite times as output. Hence, this happens when you run a program without a base case.

C program to calculate the factorial of a number using recursion

Code:

```c
#include<stdio.h>

int factorial_01(int n)
{
    if(n == 0)
        return 1;
    else
        return (factorial_01(n-1)*n);
}

int main()
{
    int a fact;

    printf("Enter a number to calculate factorial: ");
    scanf("%d",&a);

    fact = factorial_01(a);

    printf("Factorial of %d = %d",a,fact);
    return 0;
}
```

Output:

```
Enter a number to calculate factorial: 4
Factorial of 4 = 24
```

In the above C program, we are calculating the factorial using recursion. Here we declare variable n inside which the number is stored whose factorial is to be found. The function factorial_01 calculates the factorial of that number. In the factorial_01 function, if the value of n=0, then it returns 1, which is the base condition of the function. Else factorial(n-1) is recursively computed first and then multiplied to n. The factorial value is stored inside the fact that we print in the end.

Sum of Natural Numbers Using Recursion

Code:

```c
#include <stdio.h>
int sum(int a);

int main() {
    int num, x;

    printf("Enter a number: ");
    scanf("%d", &num);

    x = sum(num);

    printf("sum of natural number = %d", x);
    return 0;
}

int sum(int a) {
    if (a != 0)

        return a + sum(a-1); //sum() calls itself
    else
        return a;
}
```

Output:

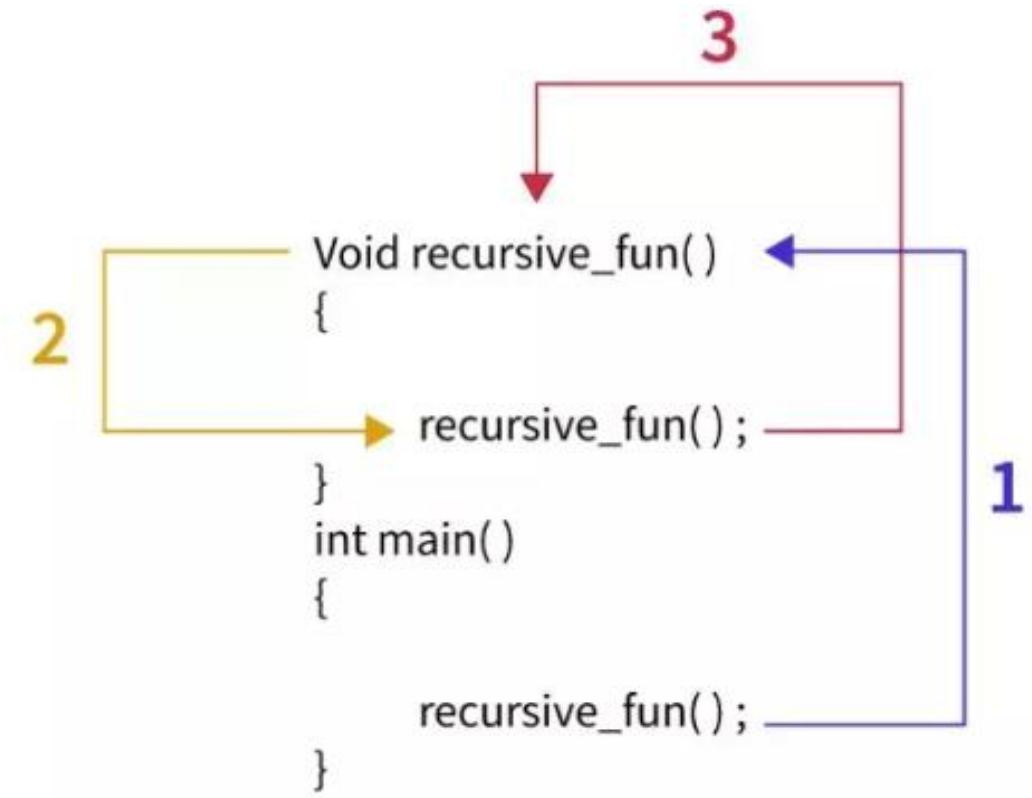
```
Enter a number: 8
sum of natural number = 36
```

Explanation: In the above program, the sum() function is invoked from the main() function in which the integer value is passed as an argument. In the sum() function, we pass an integer variable 'a' and if it is non-zero, then it returns an expression with a recursive call to the sum(a-1) function, and it continues until the value of a is equal to 0. When a is zero, the condition if sum() fails and returns the value of 'a'.

- **For example,** if we start with sum(3). As a=3 is not equal to 0 then the sum(3) function will return 3+sum(2) by calling sum(2) recursively, as a=2 not equal to 0 sum(2) will return 2+sum(1) by calling sum(1) recursively, as a=1 not equal to 0 sum(1) will return 1+sum(0) and as a==0 became true sum(0) will return 0. As sum(1)=1+sum(0) it will became 1, sum(2)=2+sum(1) it will became 3, sum(3)=3+sum(2) it will became 6. As a result sum(3) return 6 as a result of sum of first 3 natural numbers.

Recursive Function

- The recursive function is a function that repeats its execution by calling itself again and again directly or indirectly until its base case is reached. The recursive function contains a **recursive call**, which is present inside that function and calls that function. After each recursive call, the copy of that function with all the variables with the value passed in it is stored in memory, and after the function reaches the base case, the recursive calls are stopped, and the copies in the memory start to return all their values. After all the values are returned, the recursive function is terminated.



- In the above figure the recursive_fun() is the recursive function recursive_fun(); inside recursive_fun() is a recursive call.

Memory Allocation of Recursive Method

- Since recursion is a repetition of a particular process and has so much complexity, the stack is maintained in memory to store the occurrence of each recursive call. Each recursive call creates an activation record(copy of that method) in the stack inside the memory when recursion occurs. Once something is returned or a base case is reached, that activation record is de-allocated from the stack, and that stack gets destroyed. Each recursive call whose copy is stored in a stack stores a different copy of local variables declared inside that recursive function. Let's consider a C program to demonstrate the memory allocation of the recursive method.

Code:

```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
        printf("%d ",a);
        return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main

   return 0;
}
```

Output:

```
5 4 3 2 1
```

Explanation: In this C program rfunc() is a recursive function. When entering a digit, the function subtracts 1 with each recursive call from that digit and prints it until it encounters 0, and if it encounters 0, it terminates the function immediately.

Advantages of Recursion

1. The code becomes shorter and reduces the unnecessary calling to functions.
2. Useful for solving formula-based problems and complex algorithms.
3. Useful in Graph and Tree traversal as they are inherently recursive.
4. Recursion helps to divide the problem into sub-problems and then solve them, essentially divide and conquer.

Disadvantages of Recursion

1. The code becomes hard to understand and analyse.
2. A lot of memory is used to hold the copies of recursive functions in the memory.
3. Time and Space complexity is increased.
4. Recursion is generally slower than iteration.

Difference Between Recursion and Iteration

Recursion	Iteration
It is used with functions.	It is used generally with loops.
In each function call in recursion, extra Space is required in stack memory.	Here, in the case of each iteration, we do not require any space.
It terminates when the base condition is met.	It terminates when a condition becomes false.
Code size becomes smaller with recursion.	Code size is large in the case of iteration.

THANK YOU

hitbullseye