CS 2001
Programming Fundamentals

Lab 09b
Recursion

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2023

AIMS AND OBJECTIVES

This lab aims to make you familiar with the concept and use of recursion. After completing this lab you must be able to: -

1. Define what are recursive functions.
2. Identify a recursive function.
3. Identify a problem requiring a recursive function.
4. Recursively define a solution to a problem.
5. Write a recursive function.

INTRODUCTION

For a very interesting introduction to recursion go through the video in the playlist: https://youtube.com/playlist?list=PLaxvM12Ynpsr2j3izyHwy1kbG_Swg03hd&si=Nu6F-50IYO0aT9pE .

Recursion is a very fascinating way of solving a problem. It relieves you of being overburdened by solving a very complex and very big problem by asking you to solve a smaller version of the same problem. You keep dividing the problem into smaller and smaller problems until you arrive at the base step for which the solution is known. Now, the solution builds up from this base step back to the original problem solving it along the way.

To find out what is recursion go to this link: Recursion

DEFINITION

Before proceeding, make sure you understand what is function definition and function call.

Recursion is a thinking process, a concept, an approach to solving problems without the use of iterations. A recursive function, always make a call to themselves. In other words, inside the body of a recursive function there is a call to that same function. There are 2 parts of a recursive function.

1. Base Case (Stopping Condition)
   In base case, there is no call to the same function directly or indirectly. It must return something.
2. Recursive Case
   Here the recursion occurs. A call is made to the same function.

Functions in C are recursive, which means that they can call themselves. Recursion tends to be less efficient than iterative code (i.e., code that uses loop-constructs), but in some cases may facilitate more elegant, easier to read code.

THE RECURSION STORY

Once upon a time, in a secret lab, there was a scientist named Professor Lazybones. He had a boring job - writing numbers from 1 to 100. Being a clever but very lazy fellow, he had an idea. He made a clone of himself and gave the clone a sheet of paper with the number "1" written on it. "You write the numbers from 2 to 100," Lazybones said.

The clone obeyed but being as lazy as the professor, he also made a clone. This new clone continued the task from 3 to 100. Again, this new clone lazier than the previous, also made a clone, wrote 3 on the paper and told the new clone, “You write the numbers from 4 to 100”.

And so, it went on, clone after clone, each one lazier than the last.

This cloning saga continued with each successive clone passing the task down the line. As they approached the final leg of their numerical journey, the 99th clone found himself at the helm. But, much like the others, he was also afflicted by the same inherent laziness. He realized that there was only one number left to write: 100. His job was done, and he hurried back to the 98th Clone proud of accomplishing the task so quickly and writing only one number on the paper, “100”. The 98th clone grabbed the paper and, with a mischievous smile, shot the 99th clone.

All the clones got shot down by the one who created them. Finally, the paper reached Professor Lazybones with 1 to 100 written on it. Guess what, the Professor, having no other task for his loyal clone shoots him.

Professor Lazybones, who had no idea about the clone shenanigans, unwittingly got his numbers written in a most unusual way. And that's how they learned about recursion - when a task keeps getting passed down the line until something unexpected happens.

EXAMPLE: LAZYBONE(N);

Here is an example of the function lazybone to write numbers from 1 to 100.

```
01  #include <stdio.h>
02
03  void lazybone(int n){
04      //The base case
05      if (n == 100){
06              printf("100\n");
07      }
08      //Recursive case
09      else {
10              printf("%d, ",n);
11              lazybone(n+1);//Recursive Call
12              //Making the clone and asking it
13              //to write from n+1 onwards
14      }
15  }
16
17  int main(){
18      lazybone(1);
19      return 0;
```

```
20 }
```

EXAMPLE: PRINT1TO(N)

A better utilization will be to write a function to print from 1 to n.

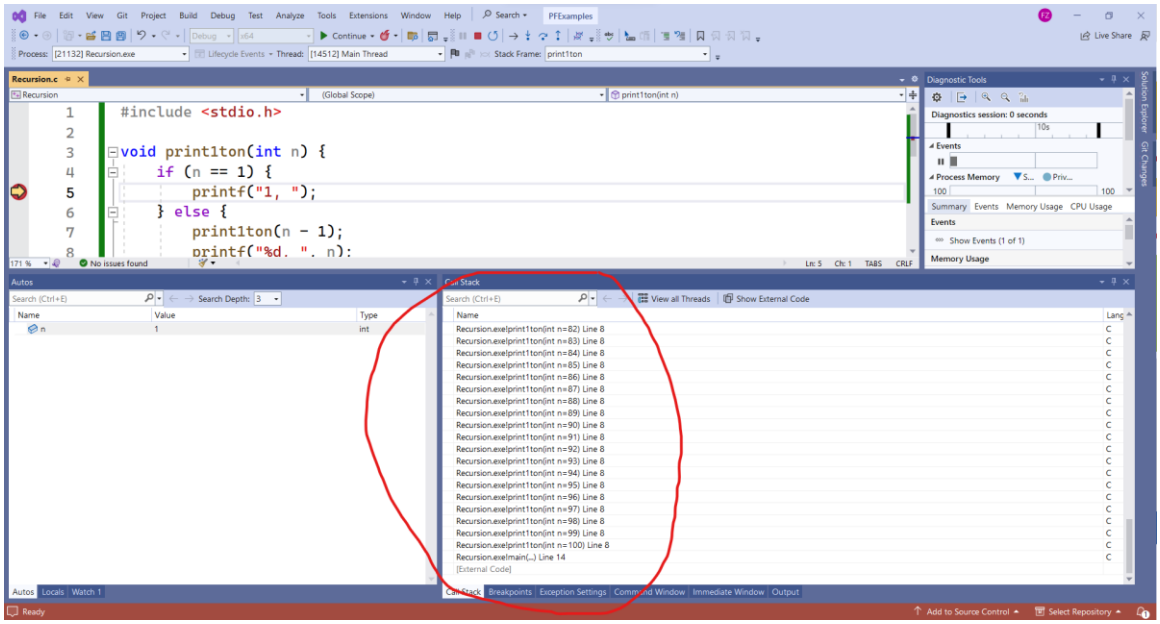
```
01   #include <stdio.h>
02
03 void print1ton(int n){
04      if (n == 1){
05              printf("1, ");
06      }
07      else {
08              print1ton(n-1);
09              printf("%d, ",n);
10      }
11 }
12
13 int main(){
14      print1ton(100);
15      return 0;
16 }
```

What happens when you swap line 08 to line 09. I.e.

```
08              printf("%d, ",n);
09              print1ton(n-1);
```

HOW IS THIS WORKING?

To understand this, appreciate that when a function call is made, the program control that started from the main function is transferred to the called function. This is done by maintaining a stack. A stack is a programming construct much like a stack of plates at a buffet. A plate can only be inserted on top and removed from top. When a function call is made the name of that function with its arguments is placed on top of this stack. When a function returns it is removed from this stack. So, when recursive calls are made, the function name (same name) gets placed on the stack until the base case is reached. Upon which the functions keep getting popped out from the stack (shot in our clone story). If you are using some advance IDE like visual code or similar you can use debug and actually see the call stack.



The calls go all the way to the stopping case i.e., n = 1.

SOLVING PROBLEMS RECURSIVELY

Although it will take time to develop the recursive thinking, here is an example of finding out factorials of integers. Recall that factorial of any number N is obtained by multiplying all the numbers from 1 to N.

E.g., 4! = 4 x 3 x 2 x 1 = 24.

What is 3!?

3! = 3 x 2 x 1 = 6.

Wait a minute. So if I have to calculate 4! and I have some way of obtaining 3!, I don't need to care how, all I need to do is multiply the answer of 3! by 4. Extending on this logic if I have answer to 2! I only need to multiply it by 3 to get 3!. Wait, wait, wait… to get 2! I only need multiply 2 into 1! which everyone knows is simply 1. The base case is therefore 1 for which we only return the number 1. Do you see what Professor Lazybones would do here? So, to get factorial for any number n simply find out the factorial for n-1 and multiply it by n, and factorial for 1 is 1.

1. Base Case
   If finding factorial of 1 return 1
2. Recursive case
   For finding factorial of N do this: N x factorial of (N-1)

TASKS TO DO:

Task 1: Write a function for finding factorial of any integer N using recursion.

Task 2: Write a function for finding product of 2 numbers without using multiplication operator and loops. Use recursion and addition only.

Task 3: Euclid gave a recursive description of GCD a very long time ago. Here is the simplified version of it.

```
01  If b is 0, then the GCD of a and b is a. This is the base case.
02  If b is not 0, then the GCD of a and b is the same as the GCD of b and the
    remainder when a is divided by b.
```

Mathematically, GCD(a, b) = GCD(b, a % b), where "%" denotes the remainder when a is divided by b, and GCD(a, 0) is a.

Write a function for finding GCD first using loops and then using recursion.

Task 4: Find out about Fibonacci series here.
https://www.mathsisfun.com/numbers/fibonacci-sequence.html
Write recursive function to print n Fibonacci numbers.
Hint: You need to make two recursive calls in the recursive case.

RECURSION

To really understand recursion, you must find out what is recursion from here.