CL1002 ***Programming Fundamentals Lab***	Lab 10 Recursion & Introduction to Structures

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2024

Author: Mr. Muhammad Nouman Hanif

AIMS AND OBJECTIVES

Aim:

To understand and implement recursion (types & stack calling), constants variables for fixed values, static variables for state retention, and structures for organizing related data in C programming.

Objectives:

- Explore recursive functions and understand the process of function self-calling for solving complex problems.
- Understand the use of constants and static variables to manage variable scope and immutability.
- Learn how to define and use structures to store and manage complex data types.

Introduction

Recursion is a powerful programming technique where a function calls itself to solve smaller instances of a problem until it reaches a **base case**. In recursion, a base case is a condition that stops the recursion from continuing indefinitely. Recursion may be a bit difficult to understand and write but problems like factorials, Fibonacci sequences, and others that have naturally recursive characteristics which often easily to implement.

Constants in C are values that do not change during program execution. They provide stability and are defined using the const keyword. **Static variables** retain their value across function calls, with scope confined to the function they're declared in, helping manage data that should persist through multiple function executions.

In C programming, **structures** provide a way to combine multiple data types into a single unit, allowing complex data handling beyond primitive types. Structures are used to represent real-world entities with multiple attributes, such as storing details for a student (name, ID, grades).

Section 1: Recursion

Recursion is the process by which a function calls itself. C language allows writing of such functions which call itself to solve complicated problems by breaking them down into simple and easy problems. These functions are known as **recursive functions**. 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.

Syntax of Recursion:

```
01 void recursive_function(){
02    recursion();   // function calls itself
03 }
04
05 int main(){
06    recursive_function();
07 }
```

Types of Recursion:

- **Direct Recursion: A** function calls itself directly.
- **Indirect Recursion:** A function calls another function, which in turn calls the original function.
- **Tail Recursion:** The recursive call is the last operation in the function.
- **Non-Tail Recursion:** The recursive call isn't the last operation, requiring additional operations after the call.

Stack Calling of Recursive Function:

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

In other words, each recursive function call is added to the stack, which maintains the execution of program. The stack allows the program to "remember" each state of the function's variables before the next recursive call. The base case pops all calls off the stack, unwinding back through the stored contexts.

Example 1: Factorial Using Recursion.

```
#include <stdio.h>

int factorial(int n) {

    if (n <= 1) {

        return 1;

    } else {

        return n * factorial(n - 1);

    }

}

int main() {

    int num = 5;

    printf("Factorial of %d is %d\n", num, factorial(num));

    return 0;

}
```

Output: *Factorial of 5 is 120*

Example 2: Sum of Natural Numbers Recursively.

```
#include <stdio.h>

int sum(int n) {

    if (n == 0) return 0;

    else return n + sum(n - 1); }

int main() {

    int num = 5;

    printf("Sum of numbers from 1 to %d is %d\n", num, sum(num));

    return 0;}
```

Output: *Sum of numbers from 1 to 5 is 15*

Example 3: Fibonacci Sequence using Recursion.

```
#include <stdio.h>

int fibonacci(int n) {

    if (n <= 1) return n;

    else return fibonacci(n - 1) + fibonacci(n - 2); }

int main() {

    int n = 5;

    for (int i = 0; i < n; i++) {

        printf("%d ", fibonacci(i));  }

    return 0; }
```

Output: 0 1 1 2 3

Problems:

1. Write a recursive function that calculates the sum of digits of a number. For example, if the input is 123, the output should be 6.
2. Write a recursive function that takes a string as input and returns the reversed string.

Section 2: Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct** keyword is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations like arrays. The difference between an array and a structure is that an array is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.

Defining a Structure:

To define a structure, we use the **struct** keyword, followed by the **structure name** and the **members** enclosed in curly braces.

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
};
```

Declaring Structure Variables:

Once a structure is defined, we can create variables of that type in the main(). Alternatively, we can declare the variables before the semi-colon (;)

```
struct Student {
    int id;
    char name[50];
    float marks;
}student1, student2;

----------------OR------------------

struct Student student1, student2;
```

Initialization of Structures:

We can initialize a structure at the time of declaration.

```
struct Student student1 = {1, " Haseeb Ahmed", 85.5};
```

Accessing Structure Members:

We can access the members of a structure using the **dot (.)** operator.

```
student1.id = 1;
strcpy(student1.name, "Haseeb Ahmed");
student1.marks = 85.5;
```

Example 1: A structure that represents a student data.

```
#include <stdio.h>

struct Student {

    int id;

    char name[50];

    float marks;

};

void printStudent(struct Student s) {

    printf("ID: %d, Name: %s, Marks: %.2f\n", s.id, s.name, s.marks);

}

int main() {

    struct Student student1 = {1, "Haseeb Ahmed", 85.5};

    printStudent(student1);

    return 0;

}
```

Output: *ID: 1, Name: Haseeb Ahmed, Marks: 85.50*

Constant and Static Variables:

Constants are used to define immutable values in C. This ensures that values remain unchanged throughout the program's execution, improving readability and reducing errors. They are defined using #define or const. They help maintain consistency and prevent unwanted changes.

- **const:** Used to declare immutable values for variables.
- **#define:** A preprocessor directive that defines constants globally.

```
01 #define PI 3.14 // global constant
02 const float PI = 3.14;
```

A **static** variable in a function retains its value between calls. Static variables have scope limited to the function but lifetime for the program's duration. They retain their value between function calls. They are initialized only once and remain in memory for the program's life.

```
01 static int sum = 0;
```

Example 2: A Chocolate structure to collect and display details (name, weight, calories, price, expiry date) for three chocolates entered by the user.

```
#include <stdio.h>

struct Chocolate {

    char Name[50];

    float Weight;

    int Calories;

    float Price;

    char ExpiryDate[20];

};

int main() {

    struct Chocolate myFavChocolates[3];
```

```
    for (int i = 0; i < 3; i++) {
        printf("Enter chocolate name: ");
        scanf("%s", myFavChocolates[i].Name);
        printf("Enter chocolate weight: ");
        scanf("%f", &myFavChocolates[i].Weight);
        printf("Enter chocolate calories: ");
        scanf("%d", &myFavChocolates[i].Calories);
        printf("Enter chocolate price: ");
        scanf("%f", &myFavChocolates[i].Price);
        printf("Enter chocolate expiry date: ");
        scanf("%s", myFavChocolates[i].ExpiryDate); }
    for (int i = 0; i < 3; i++) {
        printf("\nChocolate %d:\n", i + 1);
        printf("Name: %s\n", myFavChocolates[i].Name);
        printf("Weight: %.2f grams\n", myFavChocolates[i].Weight);
        printf("Calories: %d\n", myFavChocolates[i].Calories);
        printf("Price: $%.2f\n", myFavChocolates[i].Price);
        printf("Expiry Date: %s\n", myFavChocolates[i].ExpiryDate);
    } return 0; }
```

More problems :'D

1. Write a recursive function that takes an array and its size as input and prints all the elements.

2. Define a structure to represent a point in 2D space with x and y coordinates. Implement functions to calculate the distance between two points and to check if a point lies within a specific rectangular boundary.

3. Create a program with a constant that defines the maximum allowable temperature (in Celsius). Write a function to compare input temperatures and use a static variable to count how many times temperatures exceeded the limit.

4. Define a Book structure with fields like title, author, publication_year, and price. Create an array of books and write functions to:
   1. Display all books in the library
   2. Search for a book by title
   3. List book by a specific author

5. Write a recursive function bubbleSort that takes an array and its size. It performs the bubble sort algorithm by repeatedly comparing adjacent elements and swapping them if they are in the wrong order.

6. Design a structure to store information about travel packages, including package name, destination, duration, cost, and number of seats available. Write a program that allows users to book a travel package and display available packages.

7. Defne a date structure with variables day, month and year. Write functions to
   1. Calculate the number of days between two days.
   2. Find the day of the week for a given date
   3. Check if a date is valid accounting for leap years.

8. Write a recursive function linearSearch that takes an array, its size, the target element to search for, and the current index. It checks if the target is at the current index and continues searching in the subsequent indices until it either finds the target or exhausts the array.