BORG AL ARAB TECHNOLOGICAL UNIVERSITY

CODE: ITS501

ADVANCED PROGRAMMING IN C

Third Year –Information Technology Program Fall 2026

LECTURE 4

UNDERSTANDING RECURSION IN C PROGRAMMING

WHAT IS RECURSION?

- **Definition:** Recursion is a programming technique where a function calls itself to solve a problem.

- **Key Concept:** A recursive function must have a base case to prevent infinite loops.

COMPONENTS OF A RECURSIVE FUNCTION

Base Case

- **Definition:** The condition under which the recursion stops.
- **Purpose:** Prevents infinite recursion and eventually returns a value.

```
if (n == 0) return 1; // Base case for factorial
```

COMPONENTS OF A RECURSIVE FUNCTION

Recursive Case

- **Definition:** The part of the function where it calls itself.
- **Purpose:** Breaks down the problem into smaller sub-problems.

```
return n * factorial(n - 1); // Recursive case for factorial
```

GENERAL STRUCTURE OF A RECURSIVE FUNCTION

```
return_type function_name(parameters) {
    if (base_condition) {
        // Base case action
    } else {
        // Recursive case action
        return function_name(modified_parameters);
    }
}
```



FACTORIAL FUNCTION

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n . It's defined as:

- $n! = n \times (n-1)!$ for $n > 0$
- $0! = 1$ (base case)

- ***Explanation***: *5! = 5 × Fact(4) =5 × 4!=*
- *5× 4 × 3!=*
- *5× 4 × 3 × 2!=*
- *5× 4 × 3 × 2 × 1! =*
- *5× 4 × 3 × 2 × 1× 0! =*
- *5× 4 × 3 × 2 × 1× 1 = 120*

```
factorial(5)
    └─ 5 * factorial(4)
            └─ 4 * factorial(3)
                    └─ 3 * factorial(2)
                            └─ 2 * factorial(1)
                                    └─ 1 * factorial(0)
                                            └─ 1
```

EXAMPLE

```c
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case
    if (n == 0) {
        return 1;
    } else {
        // Recursive case
        return n * factorial(n - 1);
    }
}

int main() {
    int number = 5; // Example input
    printf("Factorial of %d is %d\n", number, factorial(number));
    return 0;
}
```

Explanation:

Base case: factorial(0) returns 1.

Recursive case: n * factorial(n - 1) reduces the problem size.

```
factorial(5)
  └─ factorial(4)
       └─ factorial(3)
            └─ factorial(2)
                 └─ factorial(1)
                      └─ factorial(0) -> returns 1
```

HOW THE CALLS HAPPEN

- **Call: factorial(5)**
  - n is 5, so it goes to the recursive case.
  - Calls factorial(4).
- **Call: factorial(4)**
  - n is 4, calls factorial(3).
- **Call: factorial(3)**
  - n is 3, calls factorial(2).
- **Call: factorial(2)**
  - n is 2, calls factorial(1).
- **Call: factorial(1)**
  - n is 1, calls factorial(0).
- **Call: factorial(0)**
  - Here, n is 0, which hits the base case.
  - Returns 1.

Now, we unwind the recursive calls:

7. **Return to factorial(1)**
   - Returns 1 * 1 = 1.
8. **Return to factorial(2)**
   - Returns 2 * 1 = 2.
9. **Return to factorial(3)**
   - Returns 3 * 2 = 6.
10. **Return to factorial(4)**
   - Returns 4 * 6 = 24.
11. **Return to factorial(5)**
   - Returns 5 * 24 = 120

The final output from factorial(5) is 120, as printed in the main function.

KEY POINTS

- **Base Case:** Prevents infinite recursion (stops the recursion).
- **Recursive Case:** Defines how to break the problem into smaller parts.
- **Call Stack:** Each function call waits for the result of the next until the base case is reached, and then results are returned in reverse order.

CALCULATING THE SUM OF FIRST N NATURAL NUMBERS USING RECURSION

- **Problem Statement:** How can we calculate the sum of the first n natural numbers?
- **Example:** For n=5, the sum is 1+2+3+4+5=15
  - Mathematical Formula:

$$\text{Sum}(n) = n + \text{Sum}(n-1)$$

  - Base Case:

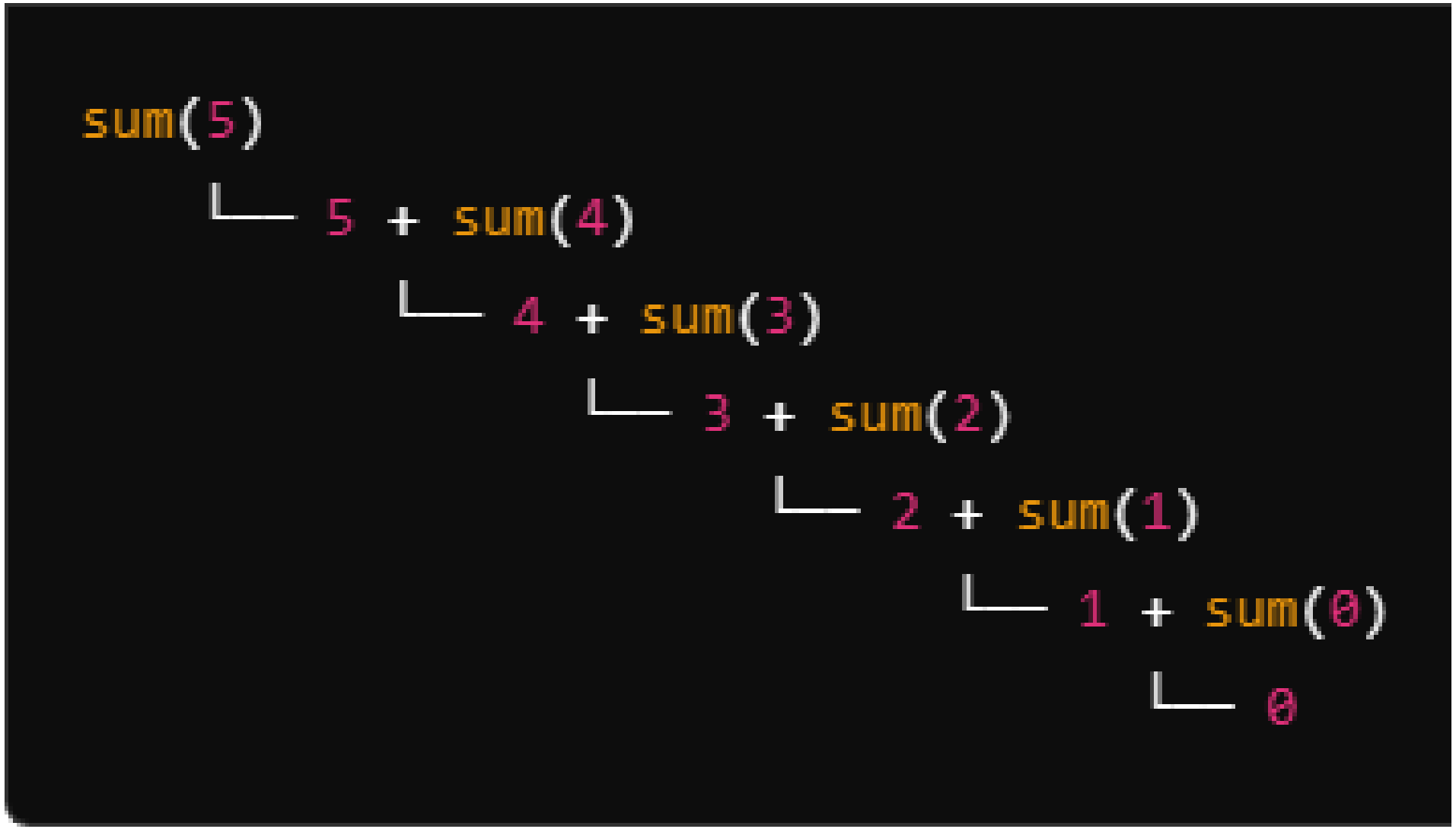
$$\text{Sum}(0) = 0$$

RECURSIVE SUM FUNCTION

```c
int sum(int n) {
    if (n == 0) return 0;  // Base case
    return n + sum(n - 1); // Recursive case
}
```

Explanation:
If n is 0, return 0.
Otherwise,
return n+sum(n−1).

VISUALIZING THE RECURSIVE CALLS



STEP-BY-STEP CALCULATION

- **Calculate:** sum(5)Calls sum(4)
- **Calculate:** sum(4)Calls sum(3)
- **Calculate:** sum(3)Calls sum(2)
- **Calculate:** sum(2)Calls sum(1)
- **Calculate:** sum(1)Calls sum(0)
- **Base Case:** sum(0) returns 0

- **Breakdown:**
- sum(1) returns 1+0=1
- sum(2) returns 2+1=3
- sum(3) returns 3+3=6
- sum(4) returns 4+6=10
- sum(5) returns 5+10=15

ADVANTAGES OF USING RECURSION

- **Simplicity:**
  - Easier to read and maintain than iterative solutions.
- **Natural Fit:**
  - Suitable for problems that can be defined in terms of smaller sub-problems.
- **Reduced Code Size:**
  - Recursive solutions can require fewer lines of code, making them easier to maintain and debug.
- **Easier to Conceptualize:**
  - Recursion closely mirrors the mathematical definition of problems, which can make the implementation more intuitive.

DISADVANTAGES OF RECURSION

- **Memory Usage:** Each recursive call consumes stack space.
- **Performance:** Can be slower than iterative approaches due to overhead.
- **Limited by Stack Size:** This makes it unsuitable for very large problems.
- **Difficult to Debug:** Debugging recursive functions can be more challenging due to multiple function calls and the complexity of the call stack.