SWEG2102 Fall 2026

Fundamentals of Programming II

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

Lecture 04

Modular Programming (Part III)

Recursion and Modular Design

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

01 Recursion concepts

02 Recursive Function

03 Modular Design and Core Principles

04 Header Files and Code Reuse

05 Common Design Mistake & best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Describe recursion concepts and its application.
- Define and use recursive functions.
- Explain and apply modular design principles.
- Create header files and implement code reusing.
- Identify common pitfalls and apply best practices.

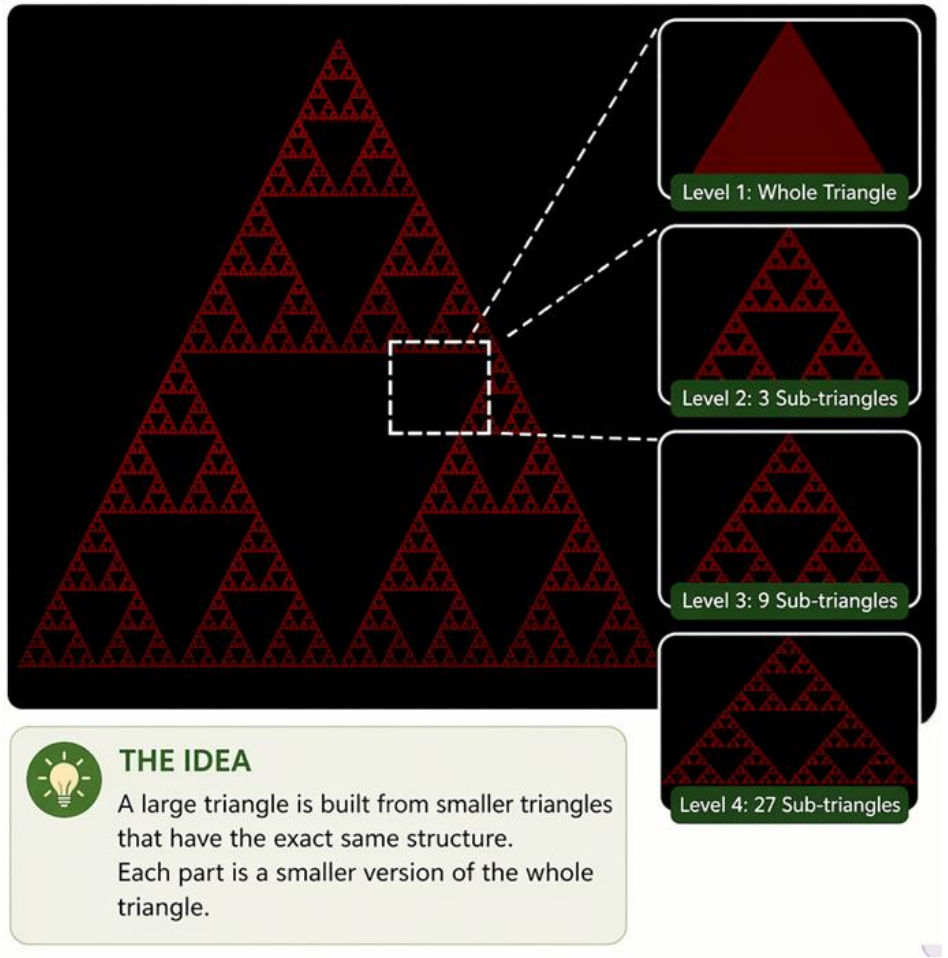
Recursive Concepts and Recursive Functions

1.1 Recursion Concepts

What is Recursion?

- Recursion is a **problem-solving technique** by breaking it down into **smaller instances, self-similar sub-problems**.
- The sub-problems become **simple enough** to solve directly.
- It is a **divide and conquer** method

Sierpiński Triangle:
the perfect example of recursion



Source: *AI-generated image (ChatGPT 3.5, 2026)*

Cont’d

Recursive Thinking

🍎 Case Study: Counting Apples in a Line

Problem:

- You want to count how many students in a long line are holding an apple.
- But the you can’t count everyone directly (too many students).

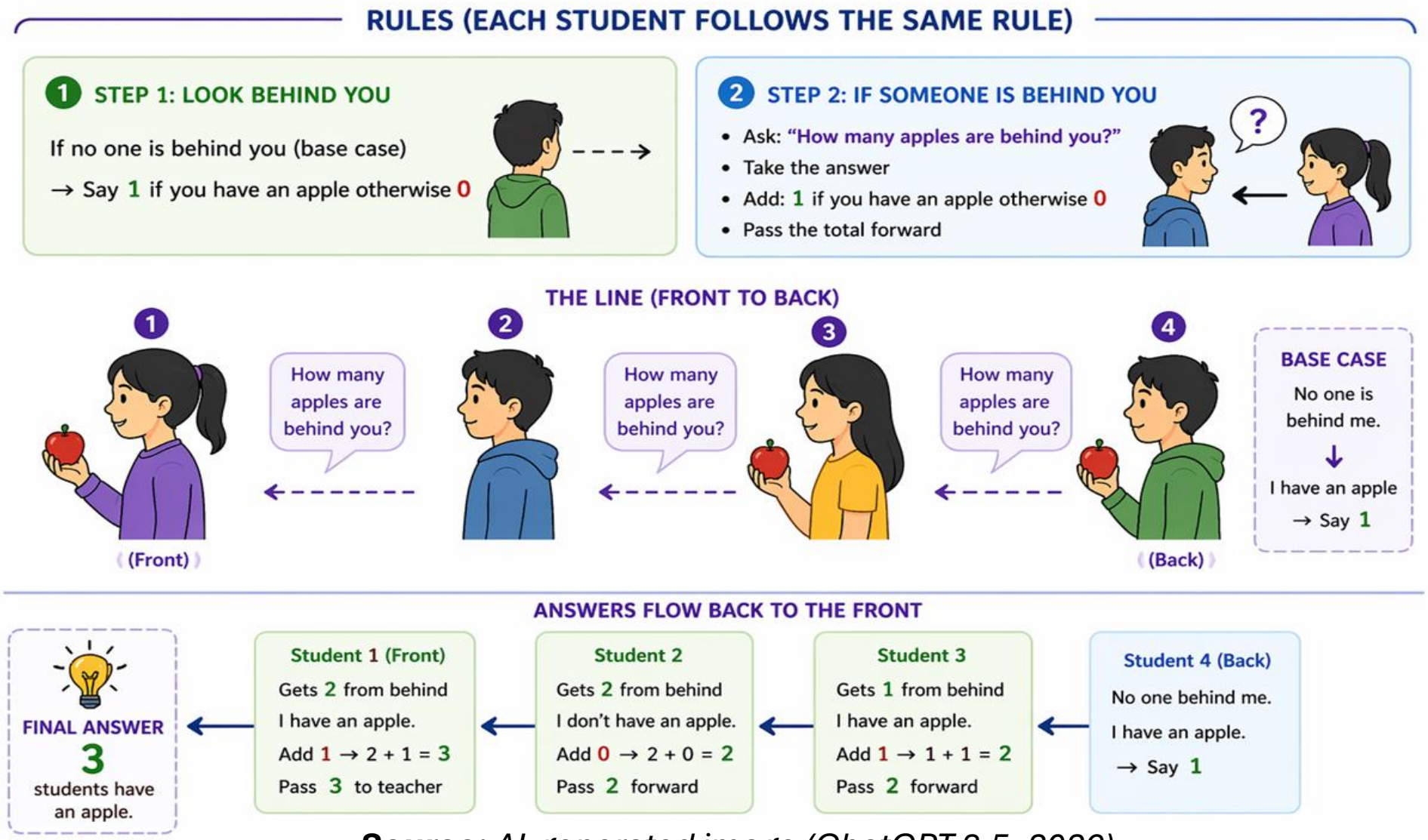
Recursive Thinking Approach:

- Let’s use a **line of students** to show how recursion works in real life.
- The **goal** is to count how many students are holding an apple by **passing a simple question along the line**, allowing each student to **add their own contribution** to the final answer step by step.

Cont’d

Solution:

Recursive Call



Source: *AI-generated image (ChatGPT 3.5, 2026)*

Cont'd

Key Components of Recursion

1. Base Case

- ✓ The **stopping condition** of recursion
- ✓ It defines when the **process should end** and prevents **infinite repetition**.
- ✓ It is the **simplest version** of the problem
- ✓ It **does not call itself** and just it **directly returns** a result/solution

Example: *A student with no one behind them simply returns 0 or 1.*

2. Recursive Case

- ✓ This is the part where the **problem is broken down** into smaller sub-parts.
- ✓ The process **calls itself** with a smaller or simpler version of the problem
- ✓ It moves **closer to the base case** each time
- ✓ It **combines the result** from the recursive call with the current step

Example:
A student asks the next student:
"How many apples are behind you?".

Cont’d

Key Components of Recursion

3. Recursive Call

- ✓ This is the actual **self-referencing** step
- ✓ The process **invokes itself**
- ✓ Each call works on a **reduced problem** size
- ✓ Think of it as **passing the same question** forward in a chain (line).

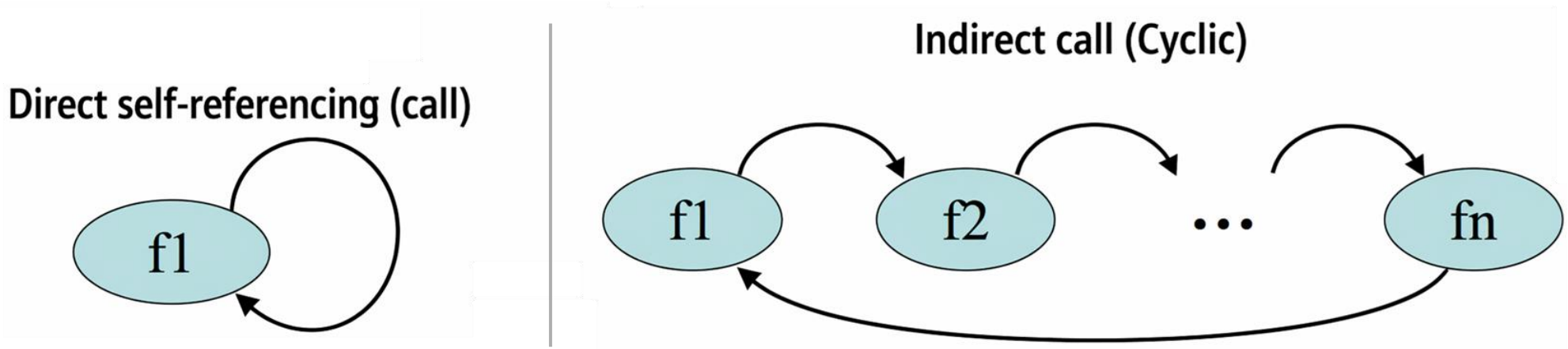
4. Stack / Call Stack Behavior

- ✓ Recursion **relies on memory** to keep track of unfinished work.
- ✓ Each call is **stored** until the base case is reached
- ✓ Once the **base case reached** and returns, results are passed back step by step
- ✓ This is why recursion “**returns backward**” after going forward.

1.2 Recursive Function

Definition

- A function that **calls itself** during execution directly or indirectly.
- Also called **self-referencing** functions
- Simply, it is a function that uses recursion
- And it comprising a **base case** and a **recursive case**



Cont'd

How a Recursive Function Works?

- A **recursive function** solves a problem by **calling itself** with a **smaller version of the same problem** until it reaches a condition where it stops.
- A recursive function typically works in **two phases**:

Phase 1: Going Down (Breaking the Problem)

- The function keeps calling itself
- Each call moves closer to the base case
- The problem size keeps shrinking

Phase 2: Coming Back Up (Building the Result)

- Once the base case is reached,
- it returns a value
- Each previous call receives the result
- The final answer is built step by step

Cont'd

Recursive Program Structure

```
recursiveFunc() {
  // ✅ Base Case (Stopping condition)
  if (simple_case) {
    // Solve directly (no recursion)
    return solution;  //Recursion stops here and returns results
  }

  // 🔄 Recursive Case
  else {
    // 1. Break problem into smaller sub-problems
    smaller_inputs = divide(problem);

    // 2. Recursion Call: solve each subproblem using recursion
    partial_results = recursiveFunc(smaller_inputs);

    // 3. Combine results
    return combine(partial_results);

     //Repeat: Function calls itself with smaller input
  }              until base case is reached
}
```

1. Base Case

- ✓ Stops recursion
- ✓ Direct solution
- ✓ No further function calls

2. Recursive Case

- ✓ Breaks problem into smaller unit
- ✓ Calls itself again
- ✓ Combines results on return

3. Execution Flow

- ✓ ↓ going down
  => **problem decomposition**
- ✓ ↑ coming up
  => **result aggregation**

Cont'd

Steps to Implement Recursion Function

Step1: Define a base case

- Identify the simplest (or base) case for which the solution is **known or trivial**.
- This prevents the function from infinitely calling itself.

Step2: Define a recursive case

- Define the problem in terms of smaller subproblems (breakdown into smaller unit).
- Call the function recursively to solve each subproblem.

Step3: Ensure the recursion terminates

- Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop

Step4: Combine the solutions

- Combine the solutions of the subproblems to solve the original problem.

Cont'd

Example 1: Digits Printing

- Implement a **recursive program** that print each digit of a given number,

  => Say, you given a number **2345**

Solutions:

Step 1: **Define Base Case**

- If the number (reminder) becomes **0**, stop recursion

Step 2: **Recursive Case**

- Break the number into separate digit:
  - Continuously divided remaining digits by 10

    → n / 10 and the last digit → n % 10

Step 3: **Ensure Termination**

- Each recursive call reduces the number:

  → 2345 → 234 → 23 → 2 → 0

  So it must eventually stop

Step 4: **Combine Results**

- First process smaller number
- Then print the last digit:

Cont'd

Recursive program to print digits of a number

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

void printDigits(int n) {
    if (n == 0)
        return; // Base Case

    printDigits(n / 10);    // Recursive Case

    cout << (n % 10) << "\n";    // Combine (print digit)
}

int main() {
    int number;

    cout << "Enter a number: ";
    cin >> number;

    // Handle zero case
    if (number == 0) {
        cout << 0;
    }
    else {
        cout << "\nOutput: ";
        printDigits(number);
    }

    return 0;
}
```

```
Enter a number: 2345

Output:
2
3
4
5
```

Phase 1: Going Down (Function Calls

The function keeps calling itself with n / 10:

```
printDigits(2345)
 └─ printDigits(234)
      └─ printDigits(23)
           └─ printDigits(2)
                └─ printDigits(0)  ← Base Case (stop)
```

Phase 2: Coming Back Up (Returning

```
printDigits(0)   → returns

printDigits(2)
 → prints: 2

printDigits(23)
 → prints: 3

printDigits(234)
 → prints: 4

printDigits(2345)
 → prints: 5
```

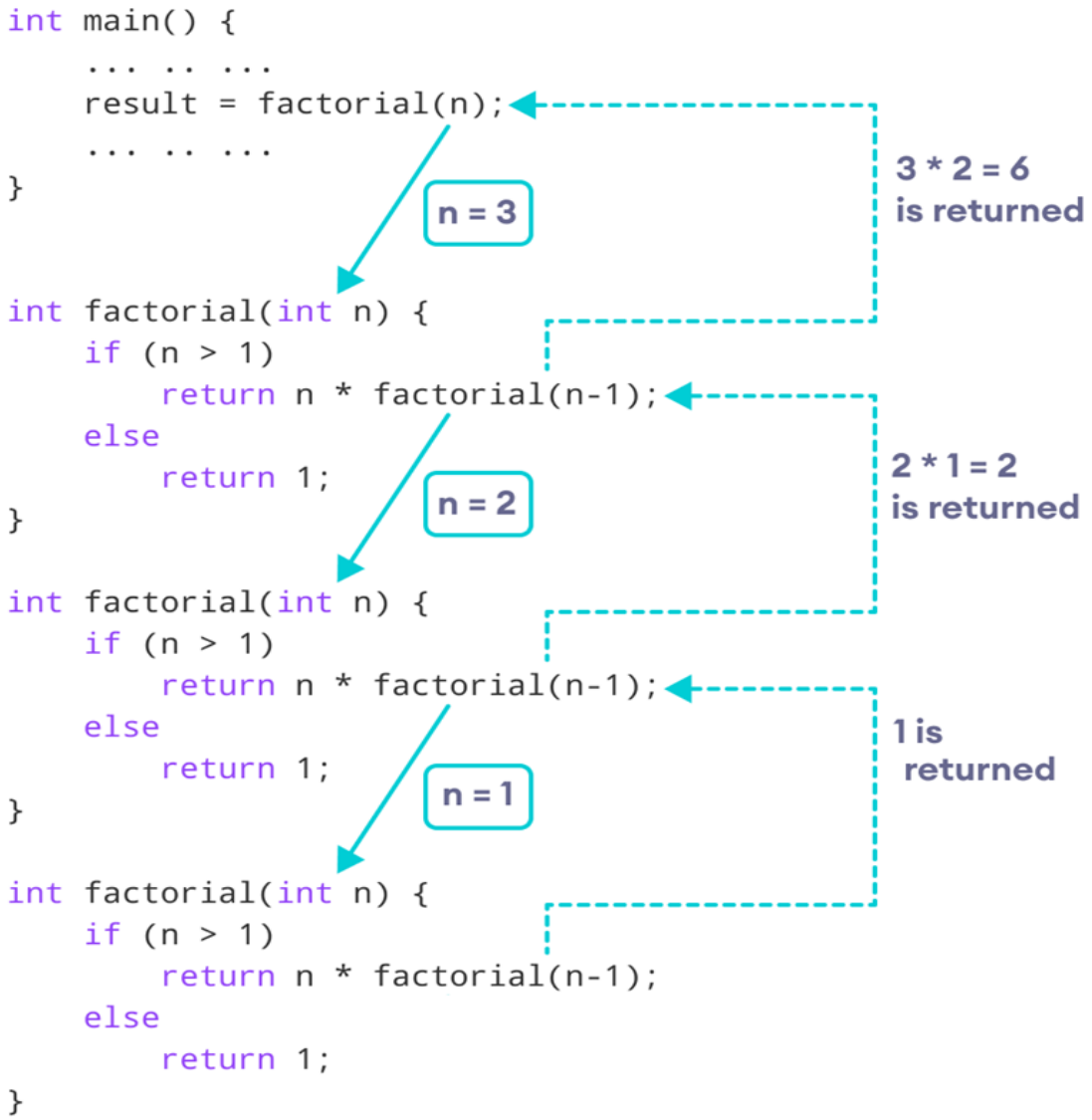
Cont'd

Example 2: Recursive Factorial

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

int factorial(int n) {

    if (n == 0 || n == 1) // Base case
        return 1;

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

int main() {
    int n;

    cout << "Enter a number: ";
    cin >> n;

    if (n < 0) {
        cout << "Negative number has factorial.";
    } else {
        cout << "Factorial of " << n << " = " << factorial(n);
    }

    return 0;
}
```

Output

```
Enter a number: 5
Factorial of 5 = 120
```



Cont'd

Example 3: Fibonacci Sequence

- Suppose the user wants to calculate the nth Fibonacci number.
- The Fibonacci sequence is: 0, 1, 1, 2, 3, 5,...
- Each number is the sum of the previous two numbers.

Solutions:

Step 1: Define Base Case

- If **n == 0** → return 0 & n == 1 → return 1
- These are the simplest known answers.

Step 2: Recursive Case

- The **nth** Fibonacci number is calculated recursively as follows:

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

- This formula allow us to break the problem into smaller subproblems:

Step 3: Ensure Termination

- Each recursive call reduces **n**. e.g. **n = 4**

4 → fib(3) & fib(2)

→ (fib(2) + fib(1)) + fib(2))

→ ((fib(1) + fib(0)) + fib(1) + (fib(1) + fib(0))

So recursion eventually stops.

Cont’d

Step 4: Combine Results

→ fib(2) = (fib(1) + fib(0)

= 1 + 0 = 1

→ fib(3) = fib(2) + fib(1)

= 1 + 1 = 2

→ fib(4) = fib(3) + fib(2)

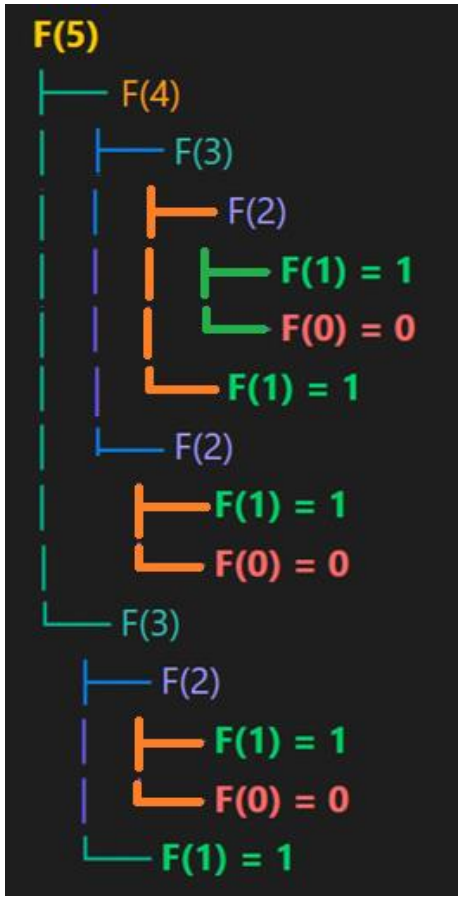
= 2 + 1 = 3

→ Answer: fib(4) = 3

Implementation in C++

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

int fibonacci(int n) {
    // Base cases
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n = 6;

    cout << "Fibonacci(" << n << ") = "
         << fibonacci(n);

    return 0;
}
```

Call Stack View



1.3 Use Cases

👉 Common Use Cases of **Recursion Function**:

✅ Data Structure Traversal

- It is the standard way to **navigate hierarchical or nested structures** like Binary Search Trees (BST), File System Operations, etc.

✅ Backtracking and Search

- Essential for exploring all possible solutions to a puzzle or decision tree and "backtracking" when a dead end is reached

✅ Divide-and-Conquer Algorithms

- Sorting algorithms like **Merge Sort and Quick Sort** rely on recursive partitioning.

✅ Mathematical Computations:

- Many mathematical problems like Factorials, Fibonacci Sequence etc. are inherently recursive, making them straightforward to implement this way.

1.4 Common Recursion Pitfalls

✕ Missing or Broken Base Cases

- ✓ No termination condition
- ✓ Forgetting a return particularly in the base case
- ✓ Leads to **infinite recursion** and stack overflow

✕ No Progress / Moving Out of Bounds

- ✓ Recursive calls don't move toward base
  e.g. **n+1** instead of **n-1**
- ✓ Same parameters passed in recursive call
- ✓ Also, leads to **infinite recursion** and stack overflow

✕ Wrong Returns

- ✓ Incorrect return values or combinations
- ✓ Wrong base case value or incorrect combine logic

✕ Side Effects

- ✓ Modifying external or global state inside recursive function
- ✓ Make debugging difficult and recursive call become hard to trace

✕ Stack Overflow

- ✓ Deep recursion logic consumes too much stack memory
- ✓ Cause program to crash when stack limit exceeds

Quick Check: Pause & Predict

1. What will happen when this function is called?

```
int fun(int n) {
    cout << n << " ";
    return fun(n - 1);
}
```

Answer: Error

→ *infinite recursion because no base cases.*

→ *The program will crash*

2. What happens if a recursive function has no base case?

A) Faster execution
B) Infinite recursion
C) Better memory usage
D) Compilation error

Answer: B

→ *The function keeps calling itself until the program crashes due to stack overflow.*

3. What happens first in a recursive function?

A) Base case is checked
B) Stack is emptied
C) Program ends
D) Return value is printed

Answer: A

→ *The stopping condition should be checked before making recursive calls.*

Modular Design and Its Core Principles

2.1. Modular Design Concepts

Definition

Modular design is a **software design technique** where a program is organized into well-structured, **independent, self-contained modules**, each handling a specific part of the system.

- *The core concepts is dividing a **complex system** into manageable parts that can **work independently.***

Goal: Manage complexity through structure

Emphasis: Design before coding

Why Modular Design?

A well-written code (function/module)
≠
a well-designed program

Modular design helps developers:

- Reduce program complexity
- Improve code readability
- Simplify debugging and testing
- Enhance maintainability
- Promote code reuse
- Enable team collaboration

Cont'd

Module Design Strategies

1. Top-Down Design Approach:

The design starts with the **whole system** and gradually breaks it into smaller modules.

- *Focuses on the big picture first*
  → high-level overview
- *Break down into sub-system/components*
  → Gradually adds detail
- *Move from general to specific*

2. Bottom-up Design Approach :

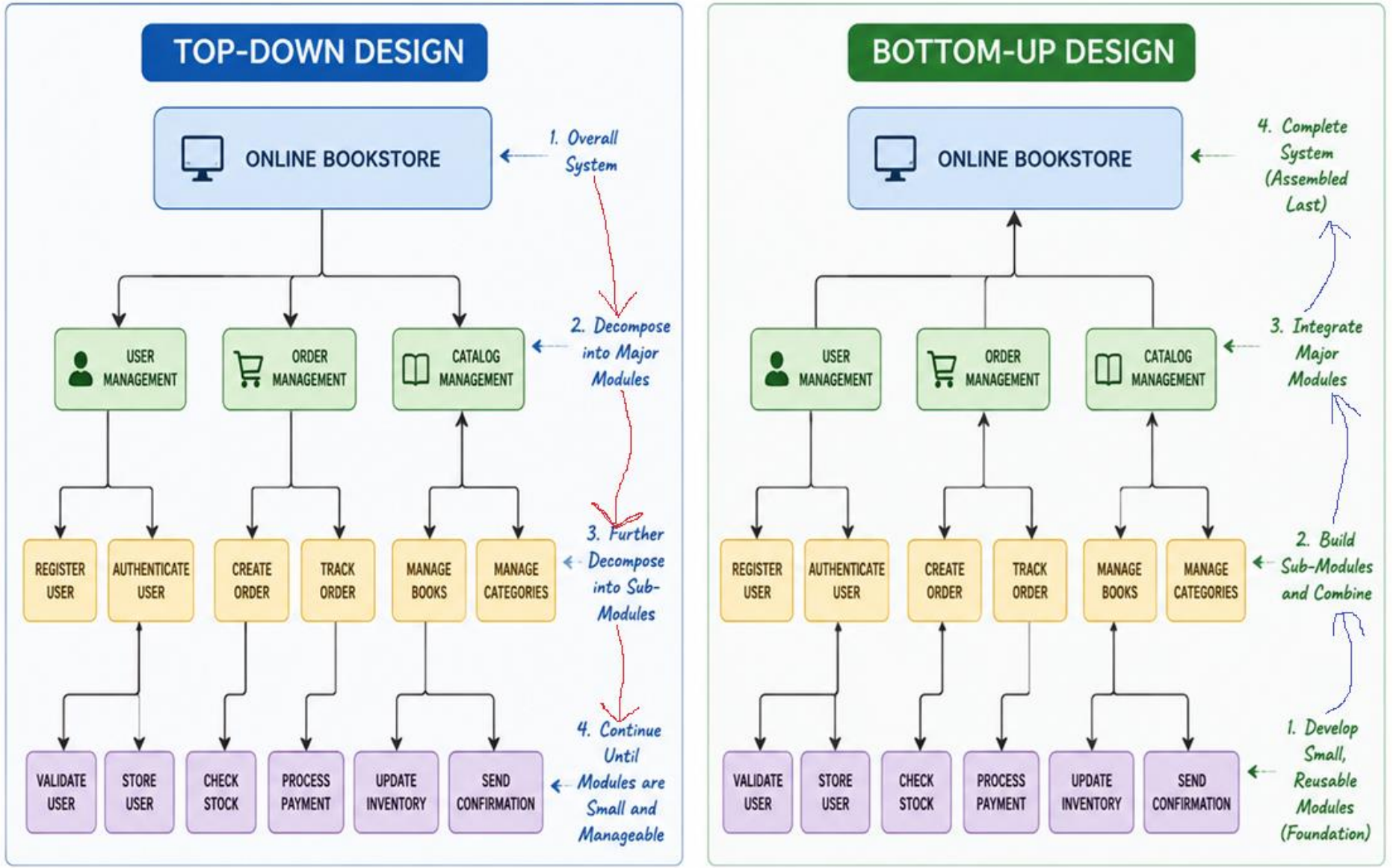
This approach starts by **creating small reusable modules/components** first and then combining them to form larger systems.

- *Build low-level modules first*
- *Test each modules separately*
- *Integrate individual modules step by step into higher-level components*
- *And eventually assemble the complete system*

Efficient for object-oriented systems.

Cont'd

Visual illustration



Source: *AI-generated image (ChatGPT 5.3)*

2.2 Core Principles of Modular Design

1. Decomposition:

- Break a large problem into smaller, manageable modules.
- Reduces complexity.
- Top-down or bottom-up approach.

2. Abstraction:

- Hiding unnecessary details.
- Focus on **what** a module does rather than **how** it does it.

3. Information Hiding (Encapsulation):

- Restricting direct access to module internals data.
- Bundling data & related operations together.

4. High Cohesion:

- Each module should perform one well-defined task.
- Easier testing, easier maintenance and better readability

5. Low Coupling:

- There should be have minimal dependencies between modules.
- A module should be able to perform its task with minimal interaction with other modules.

6. Reusability:

- Modules should be designed so that they can be reused in other systems.

Cont’d

Code Reuse (Reusability)

It is refers to the **ability to use existing software modules, or components** in multiple applications or different parts of a system without need to rewrite or redefine them.

Example:

A **discount calculation module** that used in an e-commerce system can also be reused in:

- *Supermarket software*
- *Hotel booking systems*
- *Restaurant billing systems*

Why it is important?

- Saves development time
- Reduces code duplication
- Improves consistency
- Simplify maintenance
- Increases productivity

Code reuse is more effective when software modules are designed with high cohesion and low coupling.

2.3 Header Files

Definition:

👉 **Header files** are **special files** in programming (mainly in C and C++) that contain *declarations of functions, classes definition, constants and macros, and type definition* which can be shared across multiple source files.

👉 In C/C++, header files usually have the extension:

```
</> C / C++
.h or .hpp
```

Why Header Files?

- ➤ Separate declaration from implementation
- ➤ Support modular programming
- ➤ Enable code reuse across multiple files
- ➤ Avoid writing the same code repeatedly
- ➤ Improve code organization
- ➤ Make programs easier to maintain

Cont'd

Example of a Header File

👉 **Create a reusable header file**

math_utils.h

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Function declarations
// (reusable interface)

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
float divide(int a, int b);

#endif
```

👉 **Implement the functions**

math_utils.cpp

```
#include "math_utils.h"

// Function definitions
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}

float divide(int a, int b) {
    if (b == 0) return 0;
    return (float)a / b;
}
```

👉 **Reuse in Program 1** (main_1.cpp)

```
#include <iostream>
#include "math_utils.h"

// Function usage in main program
int main() {
    cout<<"Sum: "<<add(10, 5)<<endl;
    cout<<"Product: "<<multiply(4, 6)<<endl;

    return 0;
}
```

👉 **Reuse in Program 2** (main_2.cpp)

```
#include <iostream>
#include "math_utils.h"

// Function usage in main program
int main() {
    cout<<"Difference: "<<subtract(10, 5)<<endl;

    return 0;
}
```

Practical Exercises

Practical Exercise:

Course Enrollment System Design

Objective

To design a College Course Enrollment System using **modular design principles** by decomposing the system into **input, processing**, and **output** modules.

The system should allow:

- *Allow students to select one or more courses.*
- *Verify prerequisite requirements.*
- *Check course capacity.*
- *Validate maximum credit-hour limits.*
- *Register eligible students.*
- *Display registration results and any validation errors.*

Your Task

Decompose the system into smaller modules by identifying:

1. Input Modules:

- *What data must be captured from users?*

2. Input Modules:

- What validation must be performed?
- What operations are required?

3. Output Modules:

- What information should be displayed?

Cont’d

Design Requirements

Abstraction:

- *What unnecessary details to hide?*

Encapsulation:

- *What internal details of a module to hide?*

Cohesion:

- *Identify the cohesion of each module.*

Coupling:

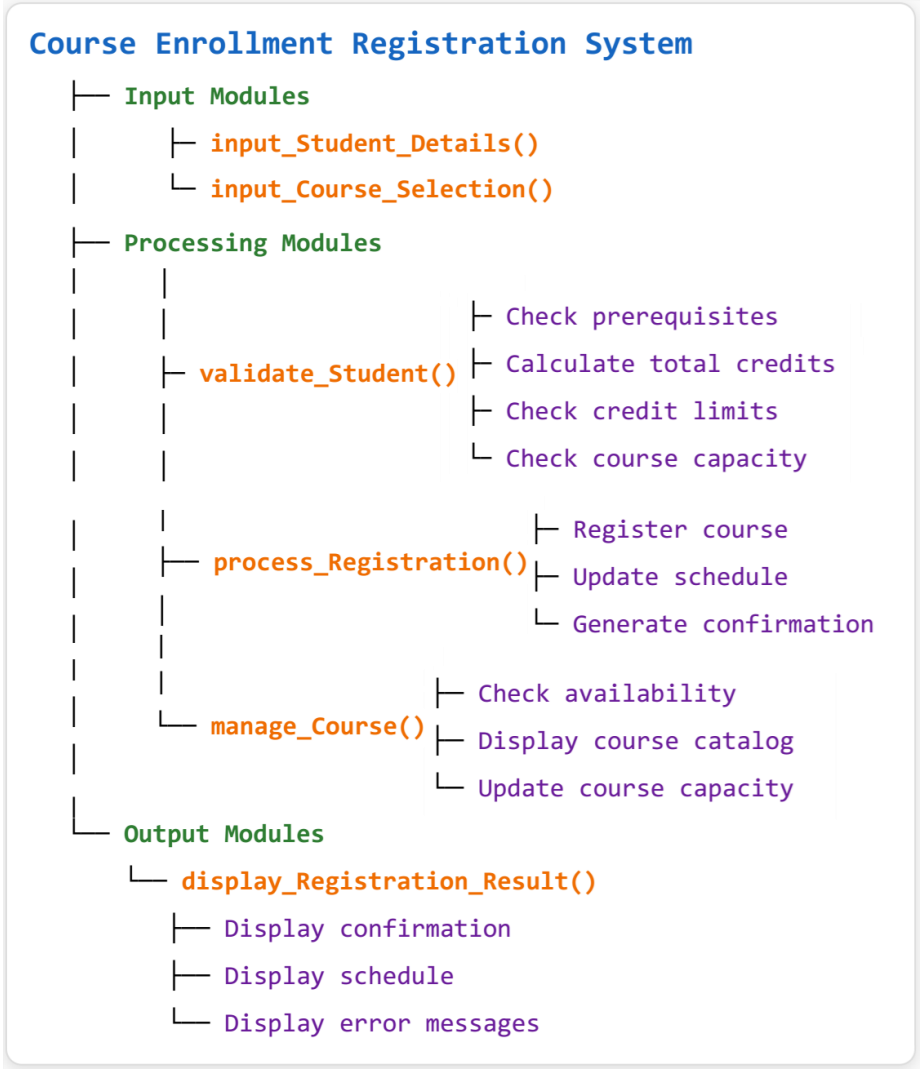
- *Identify any coupling between modules.*

Suggestion:

- *What improvements are required to achieve high cohesion and low coupling.*

System Decomposition & Module Identification

Solution



Cont'd

Design Issues:

COHESION PROBLEMS

1. **validate_Student()** performs several unrelated validation tasks.
2. **manage_Course()** mixes processing and display responsibilities.
3. **display_Registration_Result()** handles multiple output responsibilities.

COUPLING PROBLEMS

1. **process_Registration()** depends heavily on **validate_Student()**.
2. **process_Registration()** relies on **manage_Course()**.
3. Changes in one processing module can affect several others.

Solution

High-Cohesion:

- *Ensure each module performs a single, well-defined responsibility.*

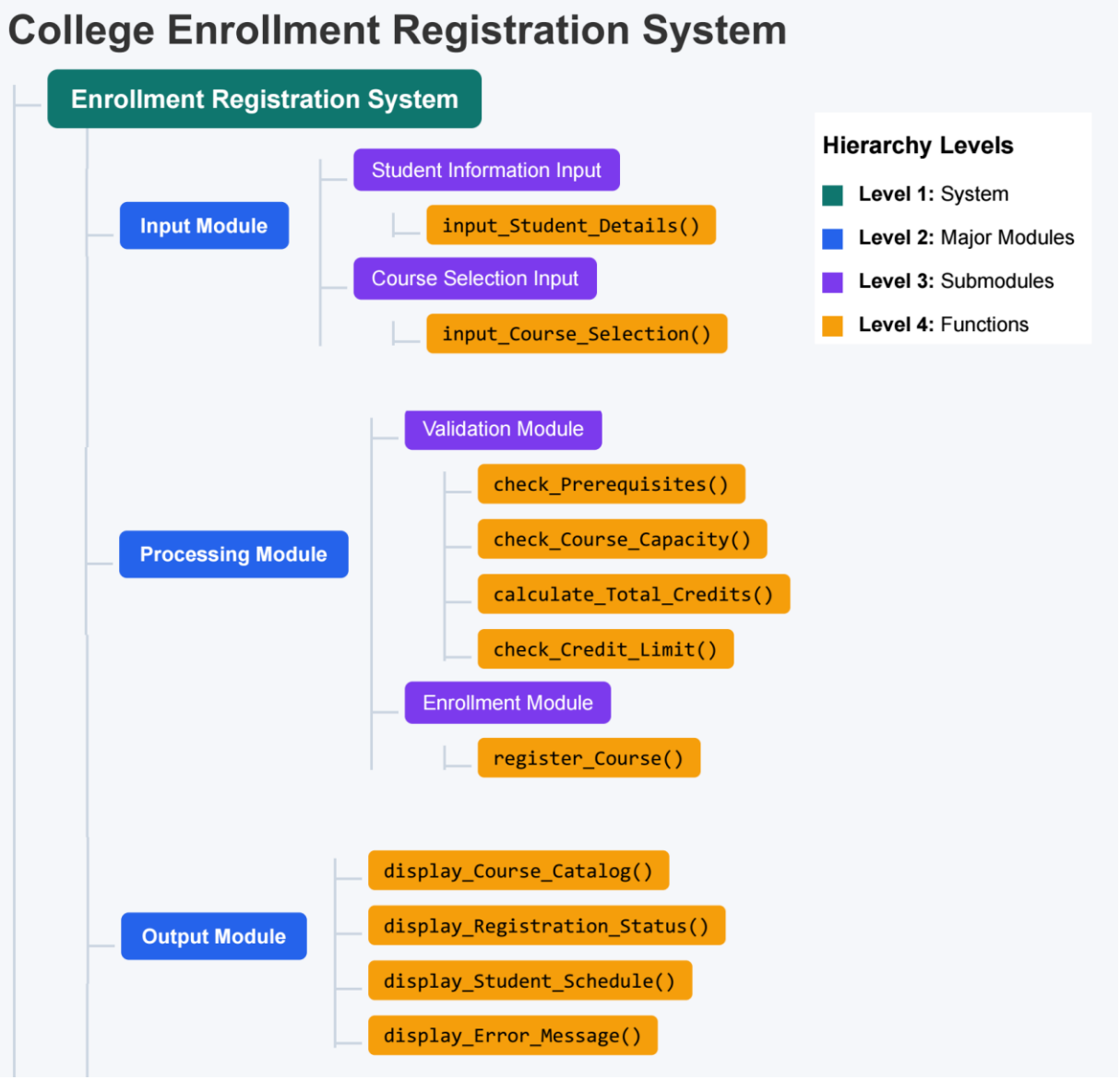
Low-Coupling:

- *Reduce heavy dependency among modules.*
- *Make modules interact through parameters and return values rather than sharing internal data.*

Cont’d

Refined

System Decomposition



High-Cohesion:

- *Each module focuses on a single responsibility.*

Low-Coupling:

- *Minimized dependency between modules*
- *Modules interact through parameters and return values*

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia