FEU INSTITUTE OF TECHNOLOGY

Elevator Problem

A C++ Program for Simulating Optimal Elevator Movement

Submitted for Compliance of the Subject,
CS0051 — Parallel and Distributed Computing
Under **Dr. Hadji Tejuco**

As Submitted By
Yap, John Angelo
Caranto, Emerson Ruter

Submitted On
January 26, 2025

Part I: Introduction

Purpose

The purpose of this project is **to simulate a multi-elevator system within a building**. The project demonstrates a practical application of algorithms to solve real-world problems, such as minimizing wait times and prioritizing elevator usage efficiently.

Elevators are an integral part of multi-story buildings, and inefficient handling can lead to longer wait times, user frustration, and energy waste. This project tackles these challenges with a dynamic simulation that allows configurable setups for floors and elevators, enabling a versatile and customizable environment.

Objectives

The key objectives of the project are as follows:

- **Optimize Elevator Operations**. Ensure elevators are dispatched based on their current position and direction, reducing wait times for users.
- **Learn and Apply Parallel Programming**. Use multi-threading in C++ to simulate real-time elevator movements. This project serves as an opportunity to learn how parallel programming can be applied in solving logistical problems in the real world, such as resource allocation, scheduling, and dynamic system updates.
- **Dynamic Visualization**. Provide a real-time display of elevator movement, allowing users to monitor progress.
- **Error Handling**. Validate user inputs to prevent incorrect floor numbers or direction mismatches, ensuring robustness.

Scope

The scope of the project defines the boundaries of what is in the simulation, including the following:

- Simulates real-time movement of elevators between floors.
- Accepts user inputs for current floor, desired direction, and destination floor.
- Visualizes elevator positions in a simple console-based interface.
- Handles up to 15 floors and up to 6 elevators.

Conversely, the project will not include any of the following limitations:

- Advanced GUI or graphical rendering.
- Real-world constraints, such as weight limits or door mechanics.
- Complex algorithms for handling simultaneous requests in large-scale systems.

Part II: Project Overview

Problem Statement

In high-rise buildings, multiple elevators operate simultaneously, but their movements are not always efficient. Poor elevator management can result in unnecessary delays and wasted energy.

This project aims to address these issues by simulating a system that dynamically assigns elevators based on:

- **Proximity**. Choosing the nearest elevator to the user's floor.
- **Direction**. Ensuring the selected elevator is already moving in the same direction as the user's request.
- **Availability**. Prioritizing idle elevators if they are closer than active ones.

Key Features

The project introduces several features to tackle the problem effectively:

- **Dynamic Elevator Assignment**. The system calculates which elevator is best suited for each request based on proximity and direction.
- **Real-Time Visualization**. Users can see the positions of all elevators and how they move after each request.
- **Error Handling**. Invalid inputs, such as out-of-range floors or mismatched directions, are detected and handled gracefully.
- **Configurable Environment**. Users can set the number of floors and elevators at the start, allowing the simulation to adapt to different building setups.

Part III: Requirements Analysis

Functional and Non-Functional Requirements

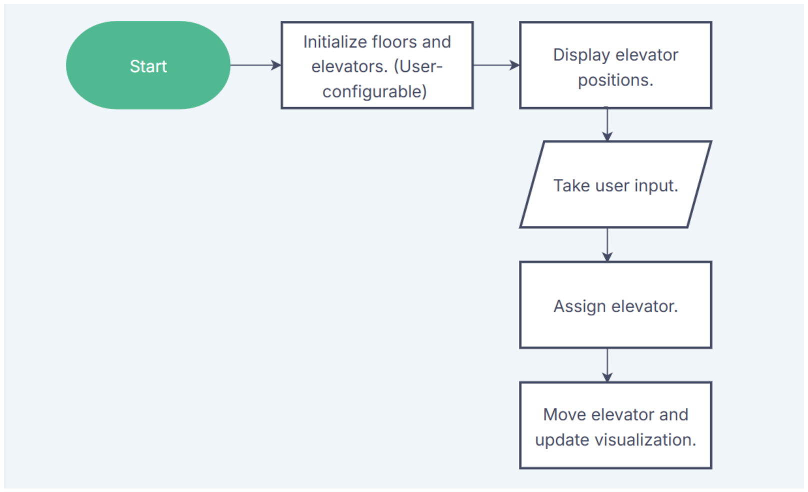
The system has several functional and non-functional requirements to ensure it operates effectively and provides a seamless user experience.

Functional requirements focus on what the system must do. The system should allow users to input their current floor, desired direction (up or down), and destination floor. These inputs must be validated to ensure correctness, such as checking that floors are within the specified range and that the direction matches the logical movement (e.g., selecting “up” while moving to a lower floor should trigger an error). Elevators should move one floor at a time, with their positions dynamically updated and displayed after each step. The program must also assign elevators optimally, prioritizing proximity and matching direction while considering idle elevators first. Additionally, users should have the flexibility to configure the number of floors and elevators during the system’s initialization.

Beyond functionality, the system must meet several ***non-functional requirements*** to ensure performance, scalability, and usability. Performance is critical as the system must process inputs and update elevator positions without noticeable delays. Scalability is another essential aspect, as the system must handle various configurations of floors and elevators efficiently. Usability is addressed through clear prompts, intuitive error messages, and an easily navigable console interface.

Part IV: System Design

Flowchart


Start
Initialize floors and elevators. (User-configurable)
Display elevator positions.
Take user input.
Assign elevator.
Move elevator and update visualization.

Part V: Implementation

Technologies Used

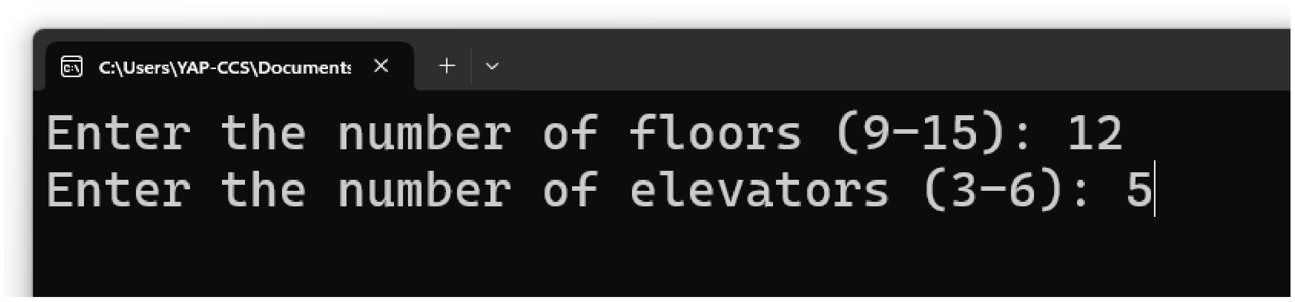
For programming language, C++ was chosen for its availability based on the course's requirements, performance, and concurrency capabilities.

For libraries, the developers have used the following:

- **<thread>** and **<chrono>** to simulate real-time elevator movement.
- **<vector>** for managing elevator states and positions.
- **<mutex>** for thread synchronization.

Screenshots

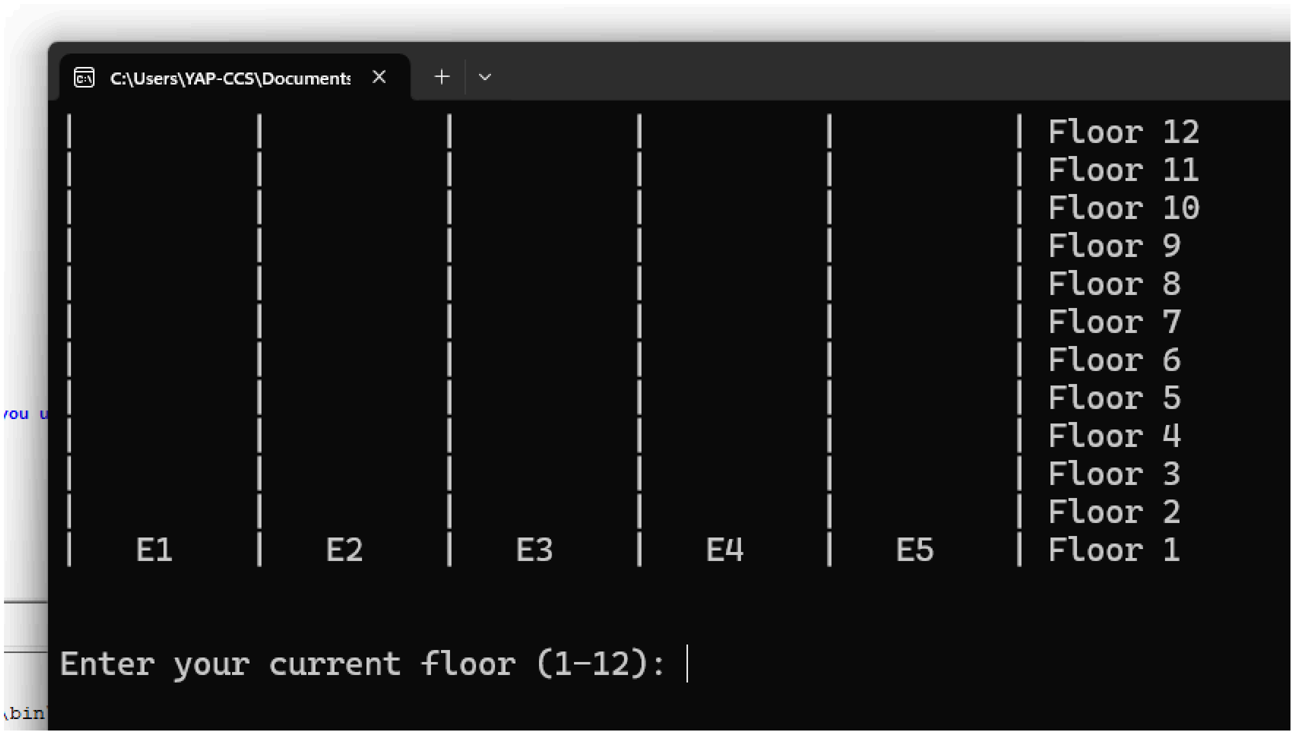
The image below shows **the initialization of elevators** through user configuration:



C:\Users\YAP-CCS\Documents

```
Enter the number of floors (9-15): 12
Enter the number of elevators (3-6): 5
```

The next image shows **the default position of the elevators after initialization**:

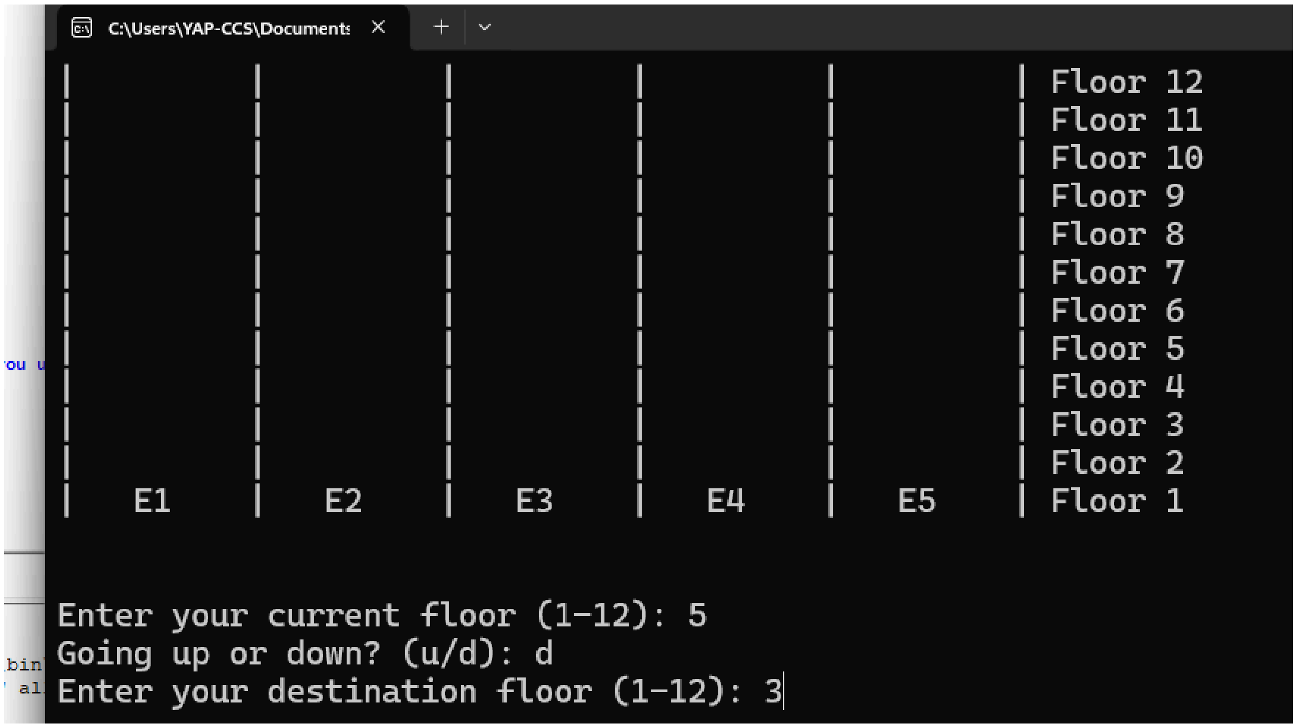


C:\Users\YAP-CCS\Documents

```
|         |         |         |         |         | Floor 12
|         |         |         |         |         | Floor 11
|         |         |         |         |         | Floor 10
|         |         |         |         |         | Floor 9
|         |         |         |         |         | Floor 8
|         |         |         |         |         | Floor 7
|         |         |         |         |         | Floor 6
|         |         |         |         |         | Floor 5
|         |         |         |         |         | Floor 4
|         |         |         |         |         | Floor 3
|         |         |         |         |         | Floor 2
|   E1    |   E2    |   E3    |   E4    |   E5    | Floor 1

Enter your current floor (1-12):
```

The next image shows **user input**:

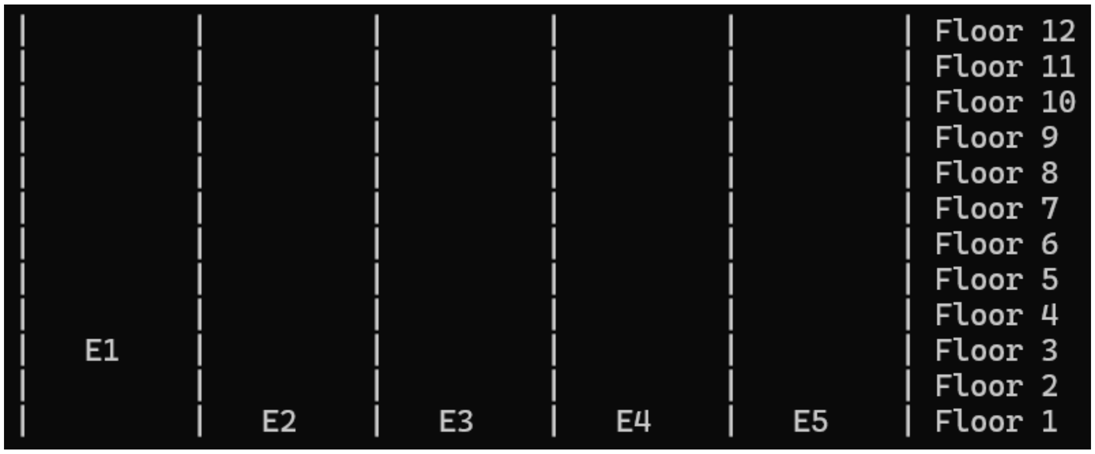


C:\Users\YAP-CCS\Documents

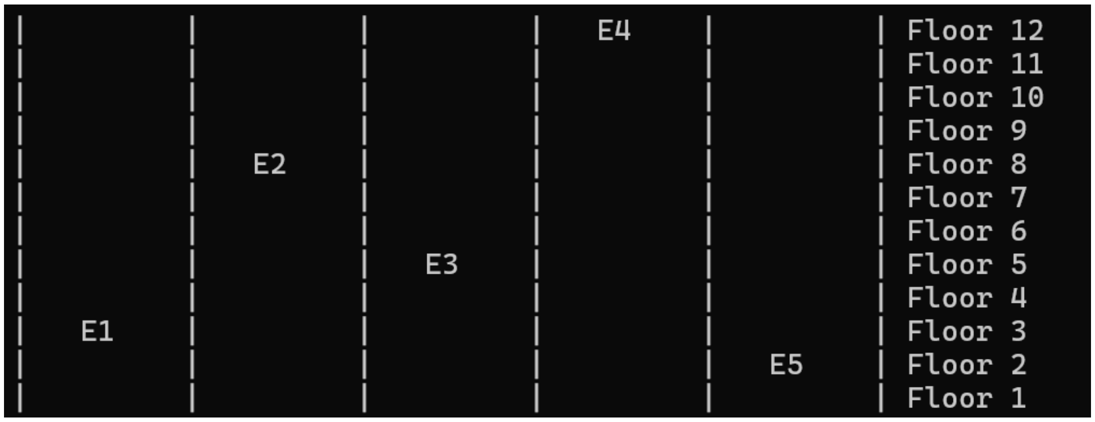
```
|         |         |         |         |         | Floor 12
|         |         |         |         |         | Floor 11
|         |         |         |         |         | Floor 10
|         |         |         |         |         | Floor 9
|         |         |         |         |         | Floor 8
|         |         |         |         |         | Floor 7
|         |         |         |         |         | Floor 6
|         |         |         |         |         | Floor 5
|         |         |         |         |         | Floor 4
|         |         |         |         |         | Floor 3
|         |         |         |         |         | Floor 2
|   E1    |   E2    |   E3    |   E4    |   E5    | Floor 1


Enter your current floor (1-12): 5
Going up or down? (u/d): d
Enter your destination floor (1-12): 3
```

The next images show **the updated position of elevators after the initial request** and after several requests:



```
|         |         |         |         |         | Floor 12
|         |         |         |         |         | Floor 11
|         |         |         |         |         | Floor 10
|         |         |         |         |         | Floor 9
|         |         |         |         |         | Floor 8
|         |         |         |         |         | Floor 7
|         |         |         |         |         | Floor 6
|         |         |         |         |         | Floor 5
|         |         |         |         |         | Floor 4
|   E1    |         |         |         |         | Floor 3
|         |         |         |         |         | Floor 2
|         |   E2    |   E3    |   E4    |   E5    | Floor 1
```



```
|         |         |         |   E4    |         | Floor 12
|         |         |         |         |         | Floor 11
|         |         |         |         |         | Floor 10
|         |         |         |         |         | Floor 9
|         |   E2    |         |         |         | Floor 8
|         |         |         |         |         | Floor 7
|         |         |         |         |         | Floor 6
|         |         |   E3    |         |         | Floor 5
|         |         |         |         |         | Floor 4
|   E1    |         |         |         |         | Floor 3
|         |         |         |         |   E5    | Floor 2
|         |         |         |         |         | Floor 1
```

Part VI: Testing

Test Cases and Results

Test Case	Input	Expected Result	Outcome
Valid Input	3, "up", 7	Elevator assigned, moves to floor 3 and then 7	**PASS**
Invalid Floor Input	17	Error message, re-prompt user	**PASS**
Invalid Direction (down or up)	Current: 4, Dest: 2, "up"	Error message, re-prompt user	**PASS**
Invalid Character Input	Non-numeric values for current and destination	Error message, re-prompt user	**PASS**
Zero Input	Input value of 0 for building floors, number of elevator, or destination	Error message, re-prompt user	**PASS**

Part VII: User Manual

Usage Instructions

A. **Initialization**. At the start, you will be asked to configure the system by entering:
    a. The number of floors (minimum: 9, maximum: 15).
    b. The number of elevators (minimum: 3, maximum: 6).
B. **Input Requests**. For every elevator request:
    a. Enter your current floor (1 to the maximum number of floors).
    b. Enter your desired direction (u for up or d for down).
    c. Enter your destination floor.
C. **Error Handling**. If any input is invalid (e.g., entering a floor out of range or an incorrect direction), the program will prompt you to re-enter valid information.
D. **View Real-Time Visualization**. After each request, the system will update and display the current position of all elevators.
E. **Exit the Program**. After completing all requests, press Ctrl + C to terminate the program.

Part VIII: Challenges, Solutions, and Future Enhancements

Challenges Faced & Solutions

- **Simulating Real-Time Movement**. Simulating the elevators moving one floor at a time while updating the visualization dynamically.
    - **Solution**: Used std::this_thread::sleep_for to create delays and mimic real-time movement. Updated elevator positions in a synchronized manner.
- **Dynamic Elevator Assignment**. Determining which elevator should handle a request based on proximity and direction.

    - **Solution**: Implemented an algorithm that calculates the most optimal elevator by comparing idle status, proximity, and direction.
- **Input Validation**. Ensuring user inputs are valid and logical, especially handling mismatched directions (e.g., pressing "up" but selecting a lower destination floor).
    - **Solution**: Added robust error-checking functions to validate all inputs.

Future Enhancements

- Replace the console-based visualization with a more interactive GUI for better user experience.
- Implement weight limits for elevators, rejecting new requests when an elevator reaches capacity.
- Add logic to prioritize requests based on urgency or waiting time.
- Use machine learning or optimization algorithms to further enhance elevator assignment efficiency.

Part IX: Conclusion

Summary

This project successfully simulates a multi-elevator optimization system that dynamically assigns and manages elevator requests. The system:

- Minimizes wait times through optimized elevator assignment.
- Ensures robust error handling and dynamic updates.
- Allows for user-configurable settings for floors and elevators.

Lessons Learned

By creating this application, the developers learned how to design and implement algorithms for dynamic optimization, gained practical experience with multi-threading to simulate real-time movement, understood the importance of strong input validation to ensure smooth functionality, and realized the value of clear instructions and visual feedback for user satisfaction.

Part X: References

Citations (APA)

GeeksforGeeks. (2019, December 27). *Parallel processing in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/parallel-processing-in-python/

GeeksforGeeks. (2024, May 16). *Concurrency in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp-concurrency/

GeeksforGeeks. (2025, January 11). *Multithreading in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/multithreading-in-cpp/

Zhu, R. (2024). Research on Optimizing Elevator Operation Management Strategy based on Mathematical Model Algorithm. *SHS Web of Conferences*, *200*, 02010. https://doi.org/10.1051/shsconf/202420002010

Part XI: Appendix

Complete Code

```
#include <iostream>
#include <vector>
#include <string>
#include <thread>
#include <chrono>
#include <mutex>

std::mutex elevatorMutex;
std::vector<int> elevators;
std::vector<std::string> elevatorStates;
int numFloors;
int numElevators;

void displayElevators() {
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    system("cls");
    for (int floor = numFloors; floor >= 1; --floor) {
        for (int i = 0; i < numElevators; ++i) {
            if (elevators[i] == floor) {
                std::cout << "|   E" << (i + 1) << "    ";
            } else {
                std::cout << "|         ";
            }
        }
        std::cout << "| Floor " << floor << std::endl;
    }
    for (int i = 0; i < numElevators; ++i) {
        std::cout << "==========";
    }
    std::cout << "=" << std::endl;
}

void moveElevator(int elevatorIndex, int targetFloor) {
    while (elevators[elevatorIndex] != targetFloor) {
        std::this_thread::sleep_for(std::chrono::milliseconds(500));

        {
            std::lock_guard<std::mutex> lock(elevatorMutex);
            if (elevators[elevatorIndex] < targetFloor) elevators[elevatorIndex]++;
            else if (elevators[elevatorIndex] > targetFloor) elevators[elevatorIndex]--;
        }

        displayElevators();
    }
}

void handleRequest(int currentFloor, int destination) {
    int bestElevator = -1;
    int minDistance = numFloors + 1;

    for (int i = 0; i < numElevators; ++i) {
        int distance = std::abs(elevators[i] - currentFloor);
        {
            std::lock_guard<std::mutex> lock(elevatorMutex);
            if ((elevatorStates[i] == "idle" ||
                 (elevatorStates[i] == "up" && destination > elevators[i] && currentFloor >=
elevators[i]) ||
                 (elevatorStates[i] == "down" && destination < elevators[i] && currentFloor <=
elevators[i])) &&
                distance < minDistance) {
                bestElevator = i;
                minDistance = distance;
            }
        }
    }

    if (bestElevator != -1) {
        {
            std::lock_guard<std::mutex> lock(elevatorMutex);
            elevatorStates[bestElevator] = (destination > currentFloor) ? "up" : "down";
        }

        moveElevator(bestElevator, currentFloor);
        moveElevator(bestElevator, destination);

        {
            std::lock_guard<std::mutex> lock(elevatorMutex);
            elevatorStates[bestElevator] = "idle";
        }
    }
}

bool validateInput(int floor, char direction, int destination) {
    if (floor < 1 || floor > numFloors || destination < 1 || destination > numFloors) {
        std::cout << "Invalid floor. Please enter a floor between 1 and " << numFloors << ".\n";
        return false;
    }
    if (direction == 'u' && destination <= floor) {
        std::cout << "You pressed UP but selected a lower floor. Try again.\n";
        return false;
    }
    if (direction == 'd' && destination >= floor) {
        std::cout << "You pressed DOWN but selected a higher floor. Try again.\n";
        return false;
    }
    return true;
}

int main() {
    std::cout << "Enter the number of floors (9-15): ";
    std::cin >> numFloors;
    while (numFloors < 9 || numFloors > 15) {
        std::cout << "Invalid input. Enter a number between 9 and 15: ";
        std::cin >> numFloors;
    }

    std::cout << "Enter the number of elevators (3-6): ";
    std::cin >> numElevators;
    while (numElevators < 3 || numElevators > 6) {
        std::cout << "Invalid input. Enter a number between 3 and 6: ";
        std::cin >> numElevators;
    }
```

```cpp
    elevators = std::vector<int>(numElevators, 1);
    elevatorStates = std::vector<std::string>(numElevators, "idle");

    while (true) {
        displayElevators();

        int currentFloor, destination;
        char direction;

        std::cout << "Enter your current floor (1-" << numFloors << "): ";
        std::cin >> currentFloor;
        std::cout << "Going up or down? (u/d): ";
        std::cin >> direction;
        std::cout << "Enter your destination floor (1-" << numFloors << "): ";
        std::cin >> destination;

        if (!validateInput(currentFloor, direction, destination)) {
            continue;
        }

        std::thread requestThread(handleRequest, currentFloor, destination);
        requestThread.detach();
    }

    return 0;
}
```