LIFT MANAGEMENT SYSTEM

SRM University – AP, Andhra Pradesh

Bachelor of Technology/Master of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Under the Guidance of

(Karnena Kavitha Rani)

Submitted by

NAME	REG NO	BRANCH & CLASS
KIRAN	AP23110010256	Btech CSE E
ABHISHEK	AP23110010304	Btech CSE E
ANUTEJASWINI	AP23110010330	Btech CSE E
HASHWITH	AP23110010280	Btech CSE E

SRM University–AP
Neerukonda, Mangalagiri,
GunturAndhra Pradesh –
522240
[November,**2024**]

Certificate

Date: 15-November-2024

This is to certify that the work present in this Project entitled **"Lift Management System"** has been carried out by our team and our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP forthe award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

Prof. Kavitha Rani Karnena

Assistance Professor.

Department of Computer Science.

Acknowledgements

This C++ Acknowledgments section sample has been created especially for this Lift Management System. In this section, contributors, tools, and resources that were crucial to the project's development are acknowledged.

Table of Contents

Problem Statement

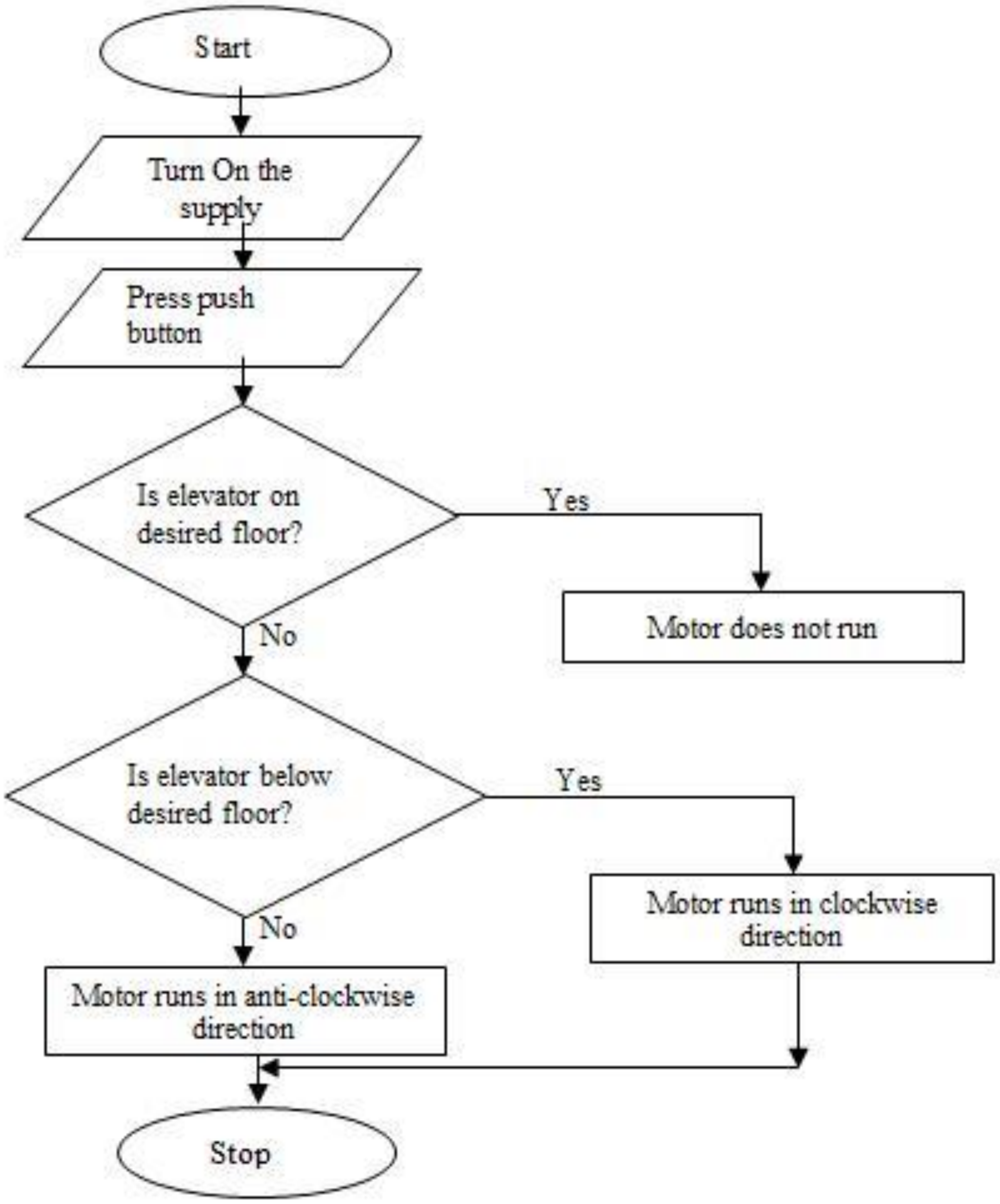
How does an elevator works with principles of OOPs with C++990

Abstract

The system includes:

1. **Lift Class**: Manages each lift's status, current and target floors, direction, and capacity.
2. **Floor Class**: Handles requests to go up or down from specific floors, tracking waiting passengers.
3. **Request Class**: Represents each request with origin and destination floors, timestamp, and direction.
4. **Lift Controller Class**: Dispatches lifts to requests, optimizing efficiency by balancing load, distance, and direction.
5. **LMS Exception Class**: Manages exceptions for errors, such as overloads or technical malfunctions.

List of figures


Start
Turn On the supply
Press push button
Is elevator on desired floor?
Yes
Motor does not run
No
Is elevator below desired floor?
Yes
Motor runs in clockwise direction
No
Motor runs in anti-clockwise direction
Stop

1.Introduction

The Lift Management System (LMS) is a tool to make elevators in buildings work better. It reduces waiting times, keeps passengers safe, and uses lifts efficiently. Building managers can use LMS to improve user experience, save energy, and reduce maintenance costs.

Built using C++, the system is organized and easy to update. It manages lift requests, plans stops, balances loads, handles emergencies, and sets up maintenance alerts. With simple error handling and an easy-to-use design, LMS ensures smooth and reliable operation. It's fast, flexible, and great for handling real-time tasks.

a) Inheritance and Core Classes

- LMS uses classes to represent essential components in elevator management, with inheritance capturing common characteristics and functions. Core classes include Lift, Floor, Request, and Maintenance.

b)Lift Operation and Maintenance

LMS includes components for managing lift movements and organizing maintenance.

Lift Operation Class

- This class controls lift movement, determining optimal paths based on requests, prioritizing nearby floors, and minimizing wait times. Functions like `move To Floor()` and `handle Request()` manage lift paths and stop schedules.

Maintenance Class

- Manages routine and emergency maintenance. It tracks issues, assigns maintenance staff, and temporarily restricts access to lifts needing repairs.

c) Error Handling

- The LMS Exception class, which inherits from `std::runtime error`, manages errors specific to LMS, such as overloaded lifts, unavailable lifts, and invalid floor requests. This custom exception provides detailed error messages to guide users.

2.Methodology

a) System Design

Object-Oriented Design:

- To represent real-world components and relationships in an elevator system, the LMS was built using OOP design principles.The key elements of this system were:

- Lift, Floor, Request, and Maintenance
- Each of these elements was represented by a class, with specific methods to handle the data and operations relevant to that component. For example:

Inheritance:

- Shared features across classes were implemented through inheritance. For example, both the `Floor` and `Maintenance` classes inherited common attributes (like `id` and `name`) from a base class called `Entity`.

Encapsulation:

- Each class encapsulated its data, keeping implementation details hidden from the user and offering specific methods to interact with that data. This allowed each part of the system to securely manage its own data while providing clear interfaces for interaction.

Polymorphism:

- Virtual functions enabled polymorphism, allowing different types of `Entity` objects (such as `Lift` and `Floor`) to display information in their own unique way depending on their role

c) Implementation

- **C++ Programming:**

- The LMS was developed in C++ to provide a strong and scalable structure, using features like classes, vectors (for flexible data storage), and exception handling.

- The system used vectors to store data about lifts, floor requests, and maintenance records, enabling easy expansion as the number of records grew.
- Core functionalities were implemented to support tasks like adding floors, managing requests, coordinating lift movements, and handling maintenance schedules.

d) Exception Handling

Custom exception handling (LMS Exception) was added to manage errors effectively and ensure the system could gracefully handle invalid inputs or operations without crashing.

3.Discussion

➢ The Lift Management System (LMS) built in C++ offers a robust solution for overseeing essential elevator functions, such as handling passenger requests, managing lift scheduling, tracking maintenance, and monitoring operational status.

➢ Object-oriented programming principles drive the system’s design, making it modular, reusable, and scalable.

➢ Dynamic memory management is achieved through vectors, allowing efficient access and organization of data as it grows.

➢ Request management supports scheduling by linking floors, lift requests, and tracking response times for efficient lift allocation.

- Key operations like lift movement, load tracking, and maintenance alerts are managed through specific functions to ensure smooth and reliable performance.

- The console-based interface is functional and interactive, although a graphical interface could provide a more modern, user-friendly experience.

- Basic exception handling via the LMS Exception class enables meaningful error reporting and ensures the system can manage invalid inputs or operations gracefully.

4. Concluding Remarks

The Lift Management System (LMS) built in C++ provides an effective and straightforward approach to managing key elevator functions. By leveraging object-oriented programming principles such as inheritance, polymorphism, and encapsulation, the system is structured to handle lift management in a modular, organized, and scalable way. This foundation effectively supports real-world elevator operations by managing lift requests, floor scheduling, maintenance, and status monitoring

a) Data Persistence:

The system currently lacks permanent data storage, which could be improved by integrating a database or file-based storage. This enhancement would allow the system to retain data even after closing, making it suitable for continuous, long-term use.

b) Advanced Features:

The LMS could be expanded with advanced capabilities, such as predictive scheduling, remote monitoring, energy efficiency tracking, and real-time maintenance alerts, making it more comprehensive.

5. Future Work

- The current Lift Management System (LMS) in C++ provides essential functionalities for managing elevator operations, such as handling passenger requests, scheduling lifts, tracking maintenance, and monitoring basic system statuses.

- Looking ahead, potential improvements include integrating a database for permanent data storage and faster data retrieval, along with developing a user-friendly Graphical User Interface (GUI) to make it easier for operators and maintenance staff to interact with the system.

- CODE FOR LMS:

```
#include <bits/stdc++.h>
using namespace std;

class Elevator {
private:
    int currentFloor;
    bool powerOn;
    bool fireEmergency;

public:
    Elevator() : currentFloor(0), powerOn(false), fireEmergency(false) {}

    void setPower(bool state) {
        if (fireEmergency) {
            cout << "Cannot turn on power. Fire emergency is active!" <<
```

```
endl;
        } else {
            powerOn = state;
            if (!powerOn) {
                cout << "Power is off. Elevator is moving to the nearest floor..."
<< endl;
                moveToNearestFloor();
            } else {
                cout << "Power is on. Elevator is ready to be used." << endl;
            }
        }
    }

    void setFireEmergency(bool state) {
        fireEmergency = state;
        if (fireEmergency) {
            cout << "Fire emergency! Elevator is moving to the ground floor
and opening doors..." << endl;


            moveToGroundFloor();
            powerOn = false;
        } else {
            cout << "Fire emergency cleared. Please turn on the power to use
the elevator." << endl;
        }
    }

    void moveToNearestFloor() {
        if (powerOn) {
            cout << "Elevator is at floor " << currentFloor << endl;
        } else {
            while (currentFloor > 0) {
                currentFloor--;
                cout << "Moving to floor " << currentFloor << endl;
            }
            cout << "Doors opening..." << endl;
        }
    }

    void moveToGroundFloor() {
        while (currentFloor > 0) {
            currentFloor--;
            cout << "Moving to floor " << currentFloor << endl;
        }
        cout << "Doors opening..." << endl;
    }

    void pressButtonOutside(string direction, int targetFloor) {
        if (!powerOn) {
            cout << "Elevator cannot be used. Power is off." << endl;
            return;
        }




        if (fireEmergency) {
            cout << "Elevator is in fire emergency mode." << endl;
            return;
        }

        if (targetFloor < 0 || targetFloor > 15) {
            cout << "Invalid floor. Please choose a floor between 0 and 15." <<
endl;
            return;
        }

        if (targetFloor == currentFloor) {
            cout << "You are already at floor " << targetFloor << ". Doors
opening..." << endl;
            return;
        }

        if (direction == "up") {
            if (currentFloor == 15) {
                cout << "Elevator is at the top floor (15). Cannot move up." <<
endl;
                return;
            } else if (currentFloor < targetFloor) {
                moveUp(targetFloor);
            } else {
                cout << "Elevator is moving down. Please press the down
button." << endl;
            }
        } else if (direction == "down") {
            if (currentFloor == 0) {
                cout << "Elevator is at the ground floor (0). Cannot move
down." << endl;
                return;
            } else if (currentFloor > targetFloor) {
                moveDown(targetFloor);
            }

else {
                cout << "Elevator is moving up. Please press the up button." <<
endl;
            }
        } else {
            cout << "Invalid button press. Use 'up' or 'down'." << endl;
        }
    }

    void moveUp(int targetFloor) {
        while (currentFloor < targetFloor) {
            currentFloor++;
            cout << "Moving up to floor " << currentFloor << endl;
        }
        cout << "Reached floor " << currentFloor << ". Doors opening..." <<
endl;
    }

    void moveDown(int targetFloor) {
        while (currentFloor > targetFloor) {
            currentFloor--;
            cout << "Moving down to floor " << currentFloor << endl;
        }
        cout << "Reached floor " << currentFloor << ". Doors opening..." <<
endl;
    }

    bool isPowerOn() const {
        return powerOn;
    }

    bool isFireEmergency() const {
        return fireEmergency;
    }



    int getCurrentFloor() {
        return currentFloor;
    }
};

int main() {
    Elevator elevator;
    int option, targetFloor;
    string direction;

    while (true) {
        if (elevator.isFireEmergency()) {
            cout << "\nFire emergency active! Only options to turn off fire
emergency or exit are available.\n";
            cout << "1. Turn off fire emergency\n2. Exit\n";
        } else if (elevator.isPowerOn()) {
            cout << "\n1. Turn power off\n2. Trigger fire emergency\n3.
Press outside button\n4. Exit\n";
        } else {
            cout << "\n1. Turn power on\n2. Trigger fire emergency\n3.
Exit\n";
        }
        cin >> option;

        if (elevator.isFireEmergency()) {
            if (option == 1) {
                elevator.setFireEmergency(false);
            } else if (option == 2) {
                cout << "Exiting..." << endl;
                return 0;
            } else {
                cout << "Invalid option. Fire emergency is active!" << endl;}
        } else if (!elevator.isPowerOn()) {
            if (option == 1) {
                elevator.setPower(true);
            } else if (option == 2) {

                elevator.setFireEmergency(true);
            } else if (option == 3) {
                cout << "Exiting..." << endl;
                return 0;
            } else {
                cout << "Power is off. No other options are available." << endl;
                }
            } else {
                switch (option) {
                    case 1:
                        elevator.setPower(false);
                        break;
                    case 2:
                        elevator.setFireEmergency(true);
                        break;
                    case 3:
                        if (elevator.getCurrentFloor() == 15) {
                            direction = "down";
                        } else if (elevator.getCurrentFloor() == 0) {
                            direction = "up";
                        } else {
                            cout << "Press 'up' or 'down': ";
                            cin >> direction;
                        }

                        cout << "Which floor would you like to go to? (0-15): ";
                        cin >> targetFloor;
                        elevator.pressButtonOutside(direction, targetFloor);
                        break;
                    case 4:
                        cout << "Exiting..." << endl;
                        return 0;
                    default:
                        cout << "Invalid option. Try again." << endl; } }  }
                  return 0;
}
```

CODE EXPLANATION:

Here's a function-by-function explanation of the elevator simulation code:

Class Constructor: Elevator()
Purpose: Initializes the elevator object with default values:
  -current Floor = 0 (starts at the ground floor).
  - power On = false (elevator is off initially).
  - fire Emergency = false` (no emergency by default).

void set Power(bool state)
- Purpose: Turns the elevator power on or off.
- Logic:
  - If a fire emergency is active, power cannot be turned on.

  - If turning power on:
    - Sets `power On` to true and prints a message indicating the elevator is ready.
  - If turning power off:
    - Moves the elevator to the nearest floor and opens the doors.
    - Ensures the elevator is idle and doors are open when power is off.

void setFireEmergency(bool state)
- Purpose: Activates or deactivates fire emergency mode.
- Logic:
  - If fire emergency is activated:
    - The elevator moves to the ground floor, opens its doors, and turns off the power for safety.
  - If fire emergency is deactivated:
    - Prompts the user to turn the power back on for normal operation.

void moveToNearestFloor()
- Purpose: Moves the elevator to the nearest floor (ground floor in this simulation).

Logic:

  - If the power is on, it displays the current floor.
  - If the power is off, the elevator simulates descending to the ground floor floor-by-floor and opens the doors.

voidmoveToGroundFloor()`

-Purpose: Moves the elevator to the ground floor (used in emergencies).

- Logic:
  - Decreases `currentFloor` step-by-step until it reaches 0, simulating the elevator moving down.
  - Once at the ground floor, the doors open.

void pressButtonOutside(string direction, int targetFloor)`

- Purpose: Handles requests from outside the elevator.
- Logic:
  - Checks if power is off or fire emergency is active; if so, the request is rejected.
  - Validates the target floor to ensure it is between 0 and 15.
  - If the target floor is the same as the current floor, it simply opens the doors.
  - Depending on the `direction`:
    - If `up` and the target floor is above the current floor, calls `moveUp()`.
    - If `down` and the target floor is below the current floor, calls `moveDown()`.
    - Prints appropriate messages for invalid cases (e.g., trying to go "up" when already at the top floor).

`void moveUp(int targetFloor)`

- Purpose: Moves the elevator up to the specified floor.
- Logic:
  - Increments the `currentFloor` step-by-step until it reaches `targetFloor`.
  - Displays each floor as the elevator moves up.
  - Opens the doors upon reaching the target.

void moveDown(int targetFloor)

- Purpose: Moves the elevator down to the specified floor.

- Logic:
  - Decrements the `currentFloor` step-by-step until it reaches `targetFloor`.
  - Displays each floor as the elevator moves down.
  - Opens the doors upon reaching the target.

`bool isPowerOn() const`
- Purpose: Returns whether the power is currently on.
- Logic:
  - Simply returns the value of `powerOn`.

`bool isFireEmergency() const`
- Purpose: Returns whether a fire emergency is active.
- Logic:
  - Simply returns the value of `fireEmergency`.

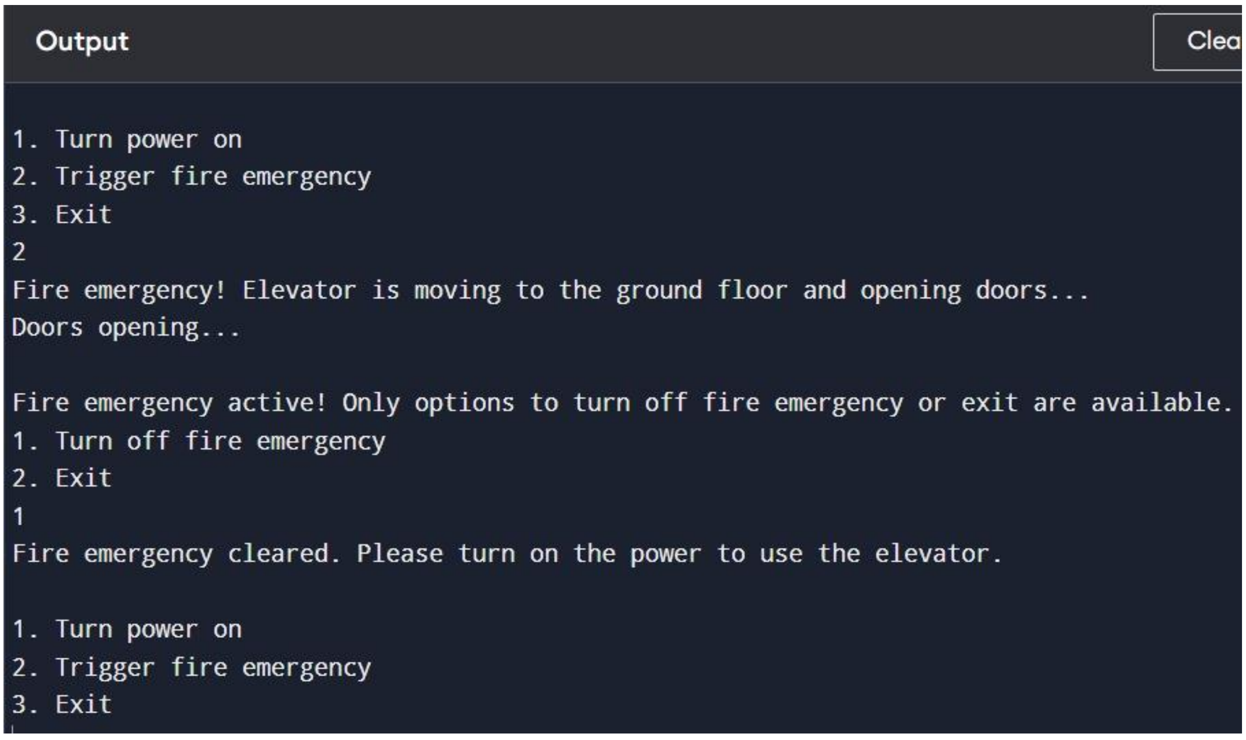
`int getCurrentFloor()`
- Purpose: Returns the current floor of the elevator.
- Logic:
  - Simply returns the value of `currentFloor`.

`main()`
- Purpose: Provides the user interface and controls the elevator simulation.
- Logic:
  - Uses a loop to display a dynamic menu based on the elevator's state:
    - If fire emergency is active:
      - Only allows turning off the fire emergency or exiting.
    - If power is off:
      - Only allows turning the power on, activating a fire emergency, or exiting.
    - If power is on:
      - Allows turning off the power, triggering a fire emergency, pressing buttons for floor requests, or exiting.
  - Handles the user's input and calls appropriate functions (e.g.,

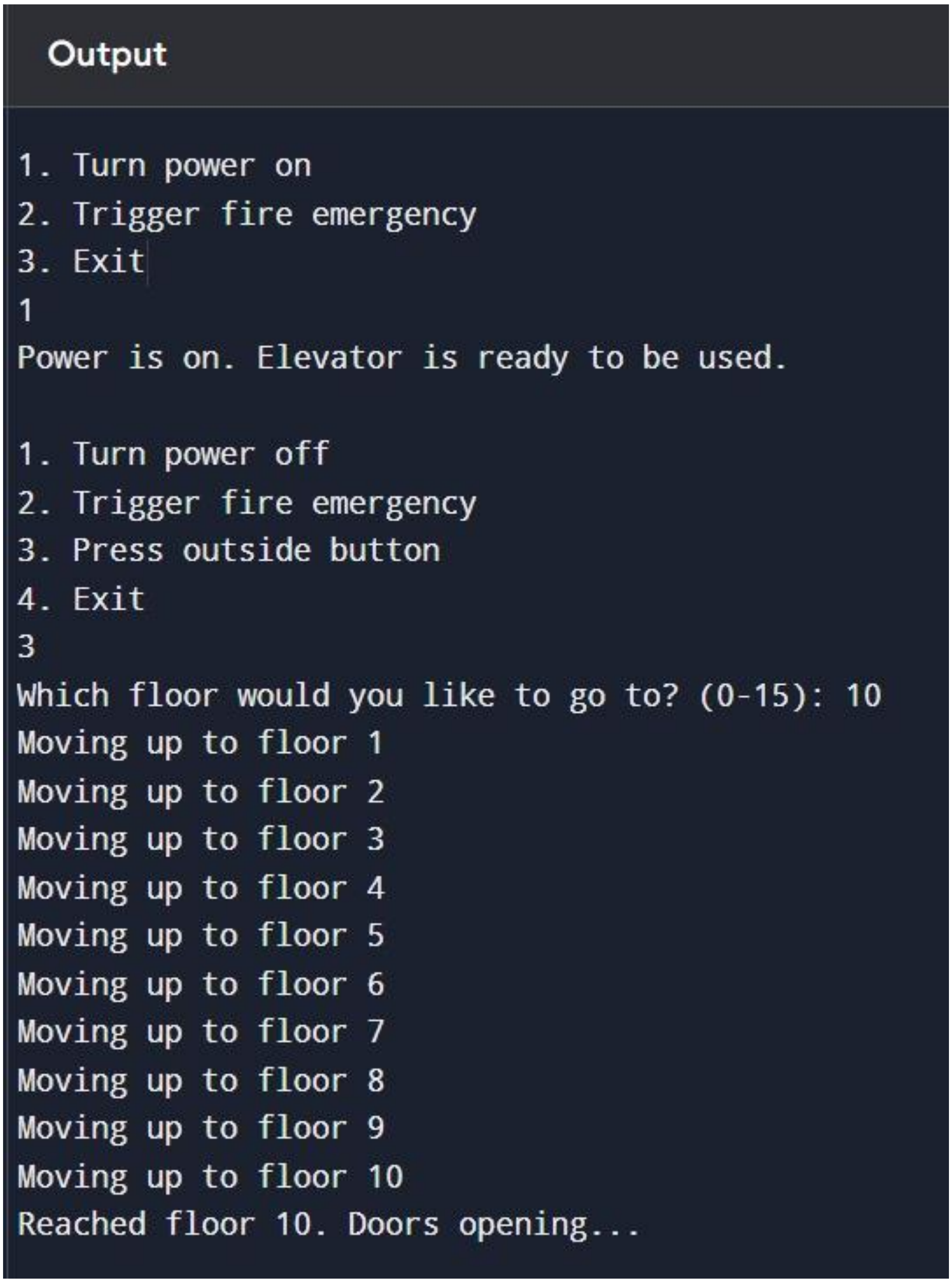
`setPower`, `setFireEmergency`, or `pressButtonOutside`).

Flow Control Example
- Power Off:
  - User turns the power on using `setPower(true)`.
- Power On:
  - User presses a button (e.g., "up" to floor 5).
  - The program validates the request and calls `moveUp(5)`.
- Fire Emergency:
  - User triggers a fire emergency using `setFireEmergency(true)`.
  - Elevator moves to the ground floor and turns off power.

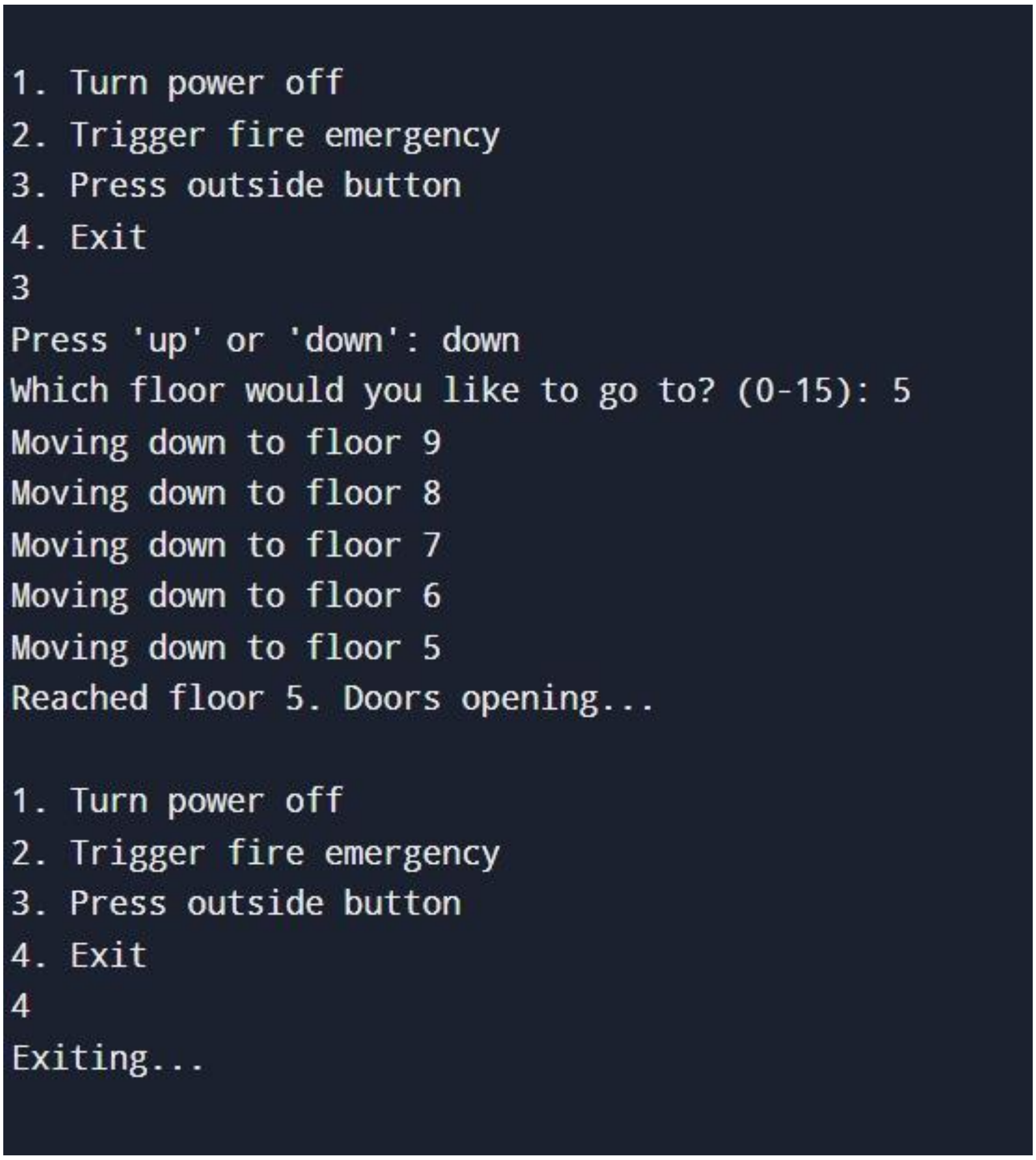
CODE OUTPUTS:



```
1. Turn power on
2. Trigger fire emergency
3. Exit
2
Fire emergency! Elevator is moving to the ground floor and opening doors...
Doors opening...

Fire emergency active! Only options to turn off fire emergency or exit are available.
1. Turn off fire emergency
2. Exit
1
Fire emergency cleared. Please turn on the power to use the elevator.

1. Turn power on
2. Trigger fire emergency
3. Exit
```


Output

```
1. Turn power on
2. Trigger fire emergency
3. Exit
1
Power is on. Elevator is ready to be used.

1. Turn power off
2. Trigger fire emergency
3. Press outside button
4. Exit
3
Which floor would you like to go to? (0-15): 10
Moving up to floor 1
Moving up to floor 2
Moving up to floor 3
Moving up to floor 4
Moving up to floor 5
Moving up to floor 6
Moving up to floor 7
Moving up to floor 8
Moving up to floor 9
Moving up to floor 10
Reached floor 10. Doors opening...
```



```
1. Turn power off
2. Trigger fire emergency
3. Press outside button
4. Exit
3
Press 'up' or 'down': down
Which floor would you like to go to? (0-15): 5
Moving down to floor 9
Moving down to floor 8
Moving down to floor 7
Moving down to floor 6
Moving down to floor 5
Reached floor 5. Doors opening...

1. Turn power off
2. Trigger fire emergency
3. Press outside button
4. Exit
4
Exiting...
```

Reference:

- ✓ **"Thinking in C++"**: Object-oriented design.
- ✓ **"Object-oriented programming in C++"**: Effective C++ concepts.
- ✓ **"STL Pocket Reference"**: Using STL containers like std::vector and algorithms like sort.