Report Template: Object-Oriented Programming and Design

# Student Name:

Olesia Mykhailyshyn

# Group:

ОПД 2

# Task:

Assignment 1

The provided code implements an airplane seat booking system. Here's a brief description of its components:

1. Airplane class (Airplane.h and Airplane.cpp):
   * Manages flight information such as flight date, flight number, seat prices, and bookings.
   * Allows booking, checking seat availability, returning tickets, and displaying flight details (like available seats and pricing).
   * Includes functionality to assign unique booking IDs and handle seat validations.
2. Ticket class (Ticket.h):
   * Represents a booked ticket containing details like the username, seat number, booking ID, and price.
3. BookingManager class (BookingManager.h and BookingManager.cpp):
   * Manages viewing and querying bookings by booking ID, username, or specific flight date and number across multiple flights.
4. FileReader class (FileReader.h and FileReader.cpp):
   * Responsible for reading flight data from a configuration file and parsing it to create Airplane objects with seat pricing information using RAII for safe file handling.
5. RAII class (RAII.h and RAII.cpp):
   * Handles safe file operations using WinAPI, ensuring that file handles are properly managed and closed after use.
6. Main program (Main.cpp):
   * Provides a command-line interface to interact with the system, allowing users to check flight details, book seats, return tickets, and view booking information.

The task implemented is a complete booking management system for flights, with functionality for handling seat reservations, ticket management, and user interactions.

# System Model:

# Зображення, що містить текст, почерк, папір, Паперовий виріб Автоматично згенерований описGitHub Link:

# <https://github.com/olesia-mykhailyshyn/Assignment_1_OOP.git>

# Solution Description:

# The system was designed with a focus on efficiency and ease of use, relying on appropriate data structures like vector and unordered\_map.

Data Structures and Why They Were Used

1. vector (Dynamic Array):

Used in:

* + - Storing flights (vector<Airplane>).
    - Storing price ranges for flights (vector<PriceRange>).
  + Why:
    - The vector container is ideal for situations where elements need to be stored sequentially and frequently accessed by index, which is common when managing multiple flights or price ranges.
    - In both the cases of flights and price ranges, elements are primarily accessed by iterating or retrieving specific entries, which vector supports efficiently.
  + Time Complexity:
    - Access: O(1) for direct access by index.
    - Insertion/Deletion at the end: O(1), but O(n) for insertion/deletion in the middle.

1. unordered\_map (Hash Map):
   * Used in:
     + Storing bookings in Airplane (unordered\_map<string, Ticket>), where the key is the seat number and the value is the corresponding Ticket object.
     + Managing booking IDs in BookingManager (unordered\_map<int, Ticket\*>).
     + Mapping users to their bookings in BookingManager (unordered\_map<string, vector<Ticket\*>>).
   * Why:
     + unordered\_map provides average O(1) time complexity for insertions, lookups, and deletions, making it ideal for managing bookings where constant-time retrieval by seat number, booking ID, or username is critical.
     + Bookings are accessed frequently by seat number and booking ID, so fast lookups are essential.
   * Time Complexity:
     + Insertion/Lookup: O(1) on average.
     + Deletion: O(1) on average.
     + Worst-case for all operations can degrade to O(n) if hash collisions occur.
2. regex (Regular Expressions):
   * Used in:
     + Validating and extracting data from input strings (flight date, seat format).
     + Parsing the flight data file in FileReader.
   * Why:
     + Regular expressions provide a flexible and powerful tool for pattern matching and validation of inputs like dates and seat formats. They simplify input validation significantly.
   * Time Complexity:
     + Matching/Searching: O(n), where n is the length of the string being matched.

Time Complexities of Major Operations

1. Check Flight Information and Seat Availability (check command)

* Relevant Operations:
  + Searching for a flight by date and flight number.
  + Displaying flight info and available seats.
* Time Complexity:
  + Searching for a flight: O(n), where n is the number of flights.
  + Displaying flight information: O(1).
  + Displaying seat availability: O(r × c), where r is the number of rows in the flight and c is the number of seats per row (constant for each row).
* Overall Complexity: O(n + r × c)

2. Book a Seat (book command)

* Relevant Operations:
  + Searching for the flight by date and flight number.
  + Checking seat availability and booking the seat.
* Time Complexity:
  + Searching for a flight: O(n), where n is the number of flights.
  + Checking seat availability: O(1) (since it uses an unordered\_map to check if the seat is already booked).
  + Booking the seat: O(1) for inserting the booking into the unordered\_map.

3. Return a Ticket (return command)

* Relevant Operations:
  + Searching for the flight that contains the booking with the specified booking ID.
  + Removing the ticket from the booking list.
* Time Complexity:
  + Searching through all flights and their bookings: O(n × b), where n is the number of flights, and b is the average number of bookings per flight.
  + Removing a booking: O(1) (since unordered\_map::erase is O(1)).

4. View Booking by Booking ID (view ID command)

* Relevant Operations:
  + Searching through all flights and bookings to find the specified booking ID.
* Time Complexity: O(n × b), where n is the number of flights, and b is the average number of bookings per flight.

5. View Booking by Username (view username command)

* Relevant Operations:
  + Searching through all flights and their bookings for matching usernames.
* Time Complexity: O(n × b), where n is the number of flights, and b is the average number of bookings per flight.

6. View Bookings by Flight (Date + Flight Number) (view flight command)

* Relevant Operations:
  + Searching for the flight by date and flight number.
  + Displaying all bookings for that flight.
* Time Complexity:
  + Searching for the flight: O(n).
  + Displaying bookings for the flight: O(b), where b is the number of bookings for the flight.
* Overall Complexity: O(n + b)

# Testing:

- Steps:

Testing the program began with reading and parsing the file. Then we checked whether the validation works correctly when entering a command. Then I tested all the commands sequentially, checking certain cases to verify the validation. I tested these commands both independently and after each other to check how the information is stored in my structure. Finally, I checked the output of the program if the file path is not specified correctly.

# Зображення, що містить знімок екрана, текст, Прямокутник, мистецтво Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, Шрифт Автоматично згенерований опис Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення Автоматично згенерований опис Зображення, що містить текст, програмне забезпечення, Мультимедійне програмне забезпечення, знімок екрана Автоматично згенерований опис

# Conclusions:

#### What Was Implemented:

The project successfully implemented a complete airplane seat booking system, with key functionalities for booking seats, returning tickets, viewing bookings, and handling multiple flights. The system allows users to interact through a command-line interface (CLI), offering commands like check, book, return, and view. It uses a modular structure where each class (e.g., Airplane, Ticket, BookingManager) is defined in its own file. The system ensures that flight data is safely parsed from a file, while seat availability and booking details are managed efficiently using appropriate data structures.

The use of & and const ensures that data is passed efficiently (by reference) and remains immutable where needed, improving performance and reducing memory overhead. This enhances code safety by preventing unintended modifications and allows for better optimization.

Each class was placed in its own header and source file, which is standard C++ practice, and file inclusion was managed using #pragma once for simplicity and to avoid issues like circular dependencies, making the code clean and scalable.

#### Differences from the Proposed Model:

One notable difference from the proposed model was in file parsing. During file parsing, lines with multiple consecutive spaces were not processed, and an exception was thrown to handle the error. This behavior was introduced to ensure the integrity of the parsed data and prevent potential errors due to formatting inconsistencies in the file.

Additionally, instead of using the standard C++ file handling library fstream, the RAII class was developed using the WinAPI functions like CreateFileW, ReadFile, and CloseHandle. This decision was made to implement custom file handling while also adhering to RAII principles. This approach provided more control over file operations and helped ensure proper resource management.

In the RAII implementation, Windows-specific data types like DWORD were encapsulated within the RAII class, preventing them from leaking into the rest of the code. This decision was based on the idea of maintaining cross-platform compatibility, as the code might need to be ported to Linux in the future.

Lastly, the logic for opening a file (via CreateFileW) was moved out of the initializer list of the constructor into the body of the constructor, following a more conventional approach to reduce complexity in the initializer list.

#### Additional Notes:

* A key challenge in file parsing was ensuring that lines with formatting issues (e.g., extra spaces) were caught and handled properly. Exceptions were thrown to handle such cases and inform the user about the error.
* RAII was implemented without relying on C++'s fstream, providing greater flexibility and control over file operations using native Windows API functions.

# Appendices:

<https://www.youtube.com/watch?v=7X8eleYIkUU> – RAII

<https://www.youtube.com/watch?v=gZFERZfsMGs&t=7s> – RAII

<https://www.geeksforgeeks.org/bus-reservation-system-in-cpp/> -- Example of Bus reservation app

<https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew> -- WinApi commands

<https://stackoverflow.com/questions/26965672/using-regex-for-input-validation> -- regex for validation parsing data