Introduction

## Swimming Pool Membership and Lesson Booking System ##

A club's swim school currently has five swimming instructors, Jeff, Anna, Peter, Michael, and Kerry. All the members in the club can book as many private swimming lessons as they like with any of the swimming instructors through the staff in the club. The swimming lessons are available from 9am to 5pm on working days, and the duration of a swimming lesson is one hour.

The program will be used by the staff in the club to manage the swimming lessons booking. The program should have a main menu that allows the staff

* to book a swimming lesson for a club member

* to cancel a swimming lesson

* to display the current schedule of a swimming instructor (An example of current schedule is shown in Figure 1. In the current schedule, 'x' indicates that the time slot has been booked by another member.)

* to display all time slots that are available for booking (An example is shown in Figure 2. In the figure, 'x' means no swimming instructor is available in the time slot, 'J/A/P/M/K' means J (Jeff), A (Anna), P(Peter), M (Michael) and K(Kerry) are all available in the time slot, 'J/A/P' means J (Jeff), A (Anna), and P(Peter) are all available in the time slot, but M (Michael) and K(Kerry) are not available in the time slot.)

* to clear all the schedules

Design:

Justification of Selected Data Structures and Algorithms:

1. **2D Arrays for Schedule Representation:**
    - **Justification:** 2D arrays provide a straightforward representation of the schedule grid, with each cell representing a time slot for a specific day. This structure allows for constant time access and manipulation of individual schedule entries.
    - **Advantages:**
        - Efficient storage and retrieval of schedule information.
        - Simple indexing for accessing specific time slots and days.
        - Suitable for modelling a grid-like structure, which is intuitive for scheduling purposes.
2. **Boolean Arrays for Instructor Availability:**

- **Justification:** Using Boolean arrays enables quick checks for instructor availability at specific time slots and days, each Boolean value indicates whether an instructor is booked or available.
- **Advantages:**
    - Constant time complexity for checking availability.
    - Straightforward representation of binary states (booked or available).
    - Efficient utilisation of memory, as Boolean values require only one bit of storage.

3. **<u>String Arrays for Instructor Names:</u>**
    - **Justification:** String arrays are suitable for storing and managing instructor names and their corresponding abbreviations. This allows for easy retrieval and display of instructor information throughout the system.
    - **Advantages:**
        - Simple storage and retrieval of instructor names.
        - Facilitates human-readable representation of instructor data.
        - Enables consistent formatting and display of names across different parts of the system.

<u>Analysis of Algorithms Providing Key Functionality:</u>

1. **Booking and Cancelling Swimming Lessons:**
    - **Justification:** These algorithms involve simple checks and updates to the schedule arrays based on instructor availability. They have a time complexity of O(1) as they directly manipulate the schedule data structures.
    - **Advantages:**
        - Efficient handling of booking and cancelling operations.
        - Constant time complexity ensures fast performance regardless of the size of the schedule.
2. **<u>Displaying Instructor Schedules:</u>**
    - **Justification:** This algorithm iterates through the schedule arrays to display an instructor's schedule. It has a time complexity of O(n), where n is the total number of time slots and days in the schedule.
    - **Advantages:**
        - Straightforward implementation for displaying schedule information.

        - Linear time complexity ensures reasonable performance even for large schedules.

3. **Filling Overall Schedule:**
    - **Justification:** This algorithm updates the overall schedule based on the availability of all instructors at each time slot. It involves iterating through the schedule arrays and has a time complexity of O(n), similar to displaying schedules.
    - **Advantages:**
        - Provides a consolidated view of all available time slots.
        - Linear time complexity allows for efficient generation of the overall schedule.

In summary, the selected data structures and algorithms are well-suited for modelling and managing swimming lesson schedules efficiently. They provide fast access, manipulation, and display of schedule information while maintaining simplicity and ease of implementation.

Let's delve deeper into the analysis of the algorithms providing key functionality:

Booking a Swimming Lesson:

- **Algorithm Analysis:**
    - This algorithm checks if the selected instructor's schedule at a given day and time slot is available. If available, it marks the instructor's schedule as booked. Additionally, if all instructors are booked for that particular time slot, it updates the overall schedule.
    - **Time Complexity:** O(1) for checking individual instructor's availability. In the worst case, if all instructors are booked, updating the overall schedule takes O(1) as well.

Canceling a Swimming Lesson:

- **Algorithm Analysis:**
    - This algorithm checks if the selected instructor has a booked lesson at the specified time slot and day. If booked, it marks the instructor's schedule as available. It also checks if the overall schedule needs to be updated due to the cancellation.
    - **Time Complexity:** O(1) for cancelling an individual instructor's lesson. If the overall schedule update is required, it's also O(1).

Displaying an Instructor's Schedule:

- **Algorithm Analysis:**

    - This algorithm iterates through each time slot and day to display the schedule of a specific instructor. It simply retrieves the schedule information from the instructor's schedule array.
    - **Time Complexity:** O(n), where n is the total number of time slots multiplied by the number of days.

Filling Overall Schedule:

- **Algorithm Analysis:**
    - This algorithm updates the overall schedule based on the availability of all instructors at each time slot. It iterates through all instructors' schedules and counts the number of booked instructors for each time slot and day. If all instructors are booked, it marks the overall schedule accordingly.
    - **Time Complexity:** O(n), where n is the total number of time slots multiplied by the number of days.

Summary of Algorithm Analysis:

- **Booking and Canceling Lessons:** Both operations have constant time complexity (O(1)), ensuring fast performance regardless of the schedule size.
- **Displaying Instructor Schedules:** This operation has linear time complexity (O(n)), where n represents the total number of time slots and days. It provides a reasonable performance for displaying individual schedules.
- **Filling Overall Schedule:** Similar to displaying schedules, this operation also has linear time complexity (O(n)), ensuring efficient generation of the overall schedule.

In conclusion, the algorithms providing key functionality in the swimming lesson scheduling system offer efficient performance, with constant time complexity for booking and cancelling lessons, and linear time complexity for displaying individual schedules and generating the overall schedule.

Testing, including:

Statement of Testing Approach Used:

The testing approach for the swimming lesson scheduling system will involve both unit testing and integration testing.

1. **Unit Testing:**
    - Each individual method of the `Schedule` class will be tested in isolation to ensure that it behaves as expected.
    - Test cases will cover different scenarios such as booking/cancelling lessons, displaying schedules, and updating the overall schedule.

- Mock objects or stubs may be used to simulate dependencies such as input/output streams or external data sources.

2. **Integration Testing:**
    - After ensuring that individual methods work correctly, integration testing will be performed to test interactions between different components of the system.
    - Test cases will cover end-to-end scenarios, including booking lessons, cancelling lessons, and displaying schedules, to validate the system's overall behaviour.
    - Integration testing will ensure that all components work together seamlessly to provide the desired functionality.

Table of Test Cases:

Test Case Description	Input	Expected Output	Type
Booking a lesson successfully	Instructor: Jeff, Day: Monday, Time: 9:00 AM	Instructor's schedule updated	Unit
Canceling a booked lesson	Instructor: Anna, Day: Tuesday, Time: 11:00 AM	Instructor's schedule updated	Unit
Displaying an instructor's schedule	Instructor: Peter	Instructor's schedule displayed	Unit
Filling overall schedule	-	Overall schedule updated	Unit
Booking lessons concurrently	Instructor: Michael, Day: Wednesday, Time: 10 AM	Both instructors' schedules updated	Integration
Canceling lessons concurrently	Instructor: Kerry, Day: Thursday, Time: 3:00 PM	Both instructors' schedules updated	Integration
Displaying overall schedule	-	Overall schedule displayed	Integration
Invalid input validation	Instructor: John, Day: Saturday, Time: 7:00 AM	Error message indicating invalid input	Unit

Note:

- Unit tests focus on individual methods, ensuring their correctness and robustness.
- Integration tests verify that different parts of the system work together as expected, testing end-to-end scenarios.

- Test cases cover various scenarios, including valid and invalid inputs, edge cases, and concurrent operations, to ensure comprehensive testing coverage.

Conclusion:

Summary of Work Done:

- **Design and Implementation:** Developed a swimming lesson scheduling system in C++, including class structure, methods for booking/cancelling lessons, displaying schedules, and filling the overall schedule.
- **Analysis:** Analysed the selected data structures and algorithms, providing justifications for their suitability and evaluating their time complexities.
- **Testing:** Defined a testing approach and created a table of test cases to verify the correctness and robustness of the system.

Limitations and Critical Reflection:

Limitations:

1. **Memory Management:** The code lacks proper memory deallocation, potentially leading to memory leaks, especially for dynamically allocated arrays like `timeCount`.
2. **Input Validation:** Limited input validation is implemented in the code. It may not handle all edge cases or invalid inputs gracefully.
3. **Dependencies:** Dependencies like `conio.h` and `stdlib.h` are included but not utilized, potentially causing confusion for developers.

Critical Reflection:

1. **Memory Management Oversight:** Failure to include proper memory management can lead to memory leaks and degrade system performance over time.
2. **Input Validation:** Insufficient input validation may result in unexpected behaviour or crashes, compromising system reliability.
3. **Unused Dependencies:** Including unnecessary dependencies can clutter the codebase and make it harder to maintain or understand.

Future Approach:

Changes for Future Tasks:

1. **Improved Memory Management:** Ensure proper memory allocation and deallocation using smart pointers or RAII (Resource Acquisition Is Initialisation) to prevent memory leaks.
2. **Comprehensive Input Validation:** Implement thorough input validation to handle a wide range of input scenarios and provide meaningful error messages.
3. **Dependency Management:** Review and remove unnecessary dependencies to keep the codebase clean and reduce complexity.

By conveying these limitations and adopting a more organised approach to memory management, input validation, and dependency management, future tasks can benefit from improved reliability, performance, and maintainability. Moreover, adopting best practices such as modular design, thorough testing, and documentation can further enhance the quality of the software developed.