LAB-REPORT

KUNJ BHUVA 202201275

I have taken a 1500 lines of code on the topic of railway management system from github, the link of which is provided below:
https://github.com/iamayan2011/Projects/blob/main/Railway%20Managment%20System%20(OOPS)/railway.cpp

QUESTION 1:

1. How many errors are there in the program? Mention the errors you have identified.

Here are the identified errors:

Improper Main Function Declaration: The function is declared as `void main()`, which is incorrect in C++. The standard main function should be defined as `int main()`. This can lead to undefined behavior.

Missing Return Statement: The main function (when corrected) should return an integer value. If `int main()` is used, it should end with a return statement (e.g., `return 0;`).

Infinite Recursion on Login Failure: When the user enters the wrong Admin ID or Password, the program recursively calls `firstPage()`, which can lead to a stack overflow if the user repeatedly enters incorrect credentials.

File Handling Issues: When opening `user.txt`, there is no check to verify if the file was opened successfully. This could result in reading from a non-existent or inaccessible file.

End-of-File Check on File Read: The loop using `while (!f1.eof())` can lead to accessing invalid data since the end-of-file (EOF) condition is checked after attempting to read. It would be better to check the read operation's success directly.

Uninitialized Variables: In the case of user login, if the user fails to log in, the variables `cuid` and `cupass` will still hold whatever values were in memory prior to initialization, which could lead to unexpected behavior.

Potential Infinite Loop: The program may not exit gracefully if the user chooses to return to the first page after entering wrong credentials; there's no way to exit the program cleanly if the admin ID or password is incorrect.

Password Handling: The program uses `cin` to input passwords, which exposes sensitive information on the console. This is a security risk.

Lack of Input Validation: There is no check on user input for the number of passengers, customer IDs, or other inputs, which could lead to unexpected behavior or crashes if invalid data is entered.

Hardcoded Train Information: While hardcoding is acceptable for a demo, consider making the train data dynamic or loading from an external source for maintainability and flexibility.

Redundant Code: The booking logic is very repetitive. Using functions to handle booking and charge assignment could reduce redundancy and improve maintainability.

User Experience Improvements: The message prompts and structure could be improved for clarity and to enhance user interaction.

Using `mainMenu()` Without Definition: The function `mainMenu()` is called multiple times, but it is not defined in the provided code. This will result in a linker error.

Static Array Size: The arrays `name`, `gender`, `bp`, `age`, and `cId` are declared with a fixed size of 6. If `n` exceeds 6, it will lead to out-of-bounds access, which is undefined behavior.

Input Validation for Passenger Count: The program only checks if `n > 6` but does not handle the case when a user inputs a negative number or zero. This can lead to an infinite loop or garbage output when accessing the arrays.

Random Number Generation: The random number generation for `pnr` uses `srand(time(NULL))` every time `information()` is called. This can lead to the same value being generated if the function is called multiple times in quick succession. It would be better to call `srand()` once, typically at the start of the program.

No Return Type for `information()`: The `information()` function does not specify a return type. Since it's used within the class, it should ideally return a type (like `void`), which is implicitly understood, but explicit declaration is a good practice.

Misleading Message for Charges: After booking a ticket, the program prints "YOU CAN GO BACK TO MENU AND TAKE THE TICKET" without actually providing a way to take the ticket or go back. The message can be confusing.

Inconsistent Data Input: The user input for `gender`, `bp`, and `age` does not include any validation, meaning users could input incorrect data types or values. For example, age should be an integer, but the code does not check if the input is indeed an integer.

Misleading Message for Charges: After booking a ticket, the program prints "YOU CAN GO BACK TO MENU AND TAKE THE TICKET" without actually providing a way to take the ticket or go back. The message can be confusing.

Inconsistent Train Listing: In the Nagpur section, the listing of train numbers for PAT-345 is incorrectly labeled as "1" instead of "2". This can confuse users when making selections.

Incorrect Use of `eof()`: In the `dispBill()` method, using `while (!ifs.eof())` can lead to reading the last line twice if it does not end with a newline character. It's safer to read the lines within the loop condition itself.

Redundant Use of `flush`: The use of `flush` in `temp << cpnr << " " << ccid << " " << cname << " " << cgen << " " << cdest << " " << ccharges << endl << flush;` is unnecessary as the `endl` already flushes the stream.

Variable Naming Convention: The variable names `cpnr`, `ccid`, `ccharges`, etc., do not follow consistent naming conventions. It would improve readability to adopt a consistent style (e.g., `customerPNR`, `customerID`, etc.).

Potential Memory Leak: The `char filename[]="foodr.txt";` in `foodOptions()` is defined within the function scope. If the program is extended to dynamically allocate memory, a memory leak could occur if it is not freed properly.

No Handling for Empty Food Options: In `foodOptions()`, there's no check or feedback for when the menu is empty or when a user tries to order food without a valid selection.

Infinite Recursion Risk: The recursive call to `getDetails()` when the PNR does not match could potentially lead to a stack overflow if the user continuously enters an invalid PNR. A loop is a better approach for retrying user input.

Invalid Switch Case Statement: In the `displayMenu()` function, the case for item 3 mistakenly lists "Rs. 210" when the price should actually be "Rs. 240". This leads to inconsistency in pricing.

Use of `goto` Statement: The `goto tryagain;` statement is generally discouraged in structured programming due to making code harder to read and maintain. A while loop would be a cleaner alternative.

Not Checking File Open Success: The `fstream` objects for `f2` and `f3` are opened without checking if the files opened successfully, which can lead to issues if the files cannot be created.

Missing Menu Option Handling: There is no handling for when the user enters a number outside the valid range of menu options in `displayMenu()`. It could be beneficial to implement input validation for this.

2. Which category of program inspection would you find more effective?

Category F: Interface Errors is particularly effective because it can highlight issues related to the number and types of parameters being passed between functions and the potential mismatch in expectations between modules (e.g., user input and file handling).

Category D: Data Errors is particularly effective here because many issues arise from the use of arrays and user input without validation. Inspecting how data is handled (like the passenger information) can reveal potential issues with out-of-bounds access, data integrity, and type mismatches.

Category A: Control Flow Errors would be beneficial here, as many of the identified issues revolve around how the flow of information (like file writing and passenger data) is handled, and how choices are managed

Category B: Data Handling Errors is appropriate here, as the issues primarily relate to how data is read from files, handled, and modified..

3. Which type of error were you not able to identify using program inspection?

- **Run-time Errors**: Specific errors like file access issues (file not found, file permission issues) cannot be detected during inspection, as these only arise during the execution of the program.
- **Logical Errors**: Issues like infinite loops or logic that leads to unhandled scenarios (like failing multiple times to log in) can only be detected through dynamic testing or debugging, as they depend on user interactions.
- **File I/O Errors**: Errors related to file permissions, file existence, or any disk-related issues cannot be identified without running the program.

- **Concurrency Issues**: If multiple instances of this program were run simultaneously, issues related to file access and modification could arise, which are not apparent through static inspection.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worthwhile. It has helped identify critical issues such as the improper function signature, file handling problems, and logical errors in the code that may lead to stack overflow or infinite recursion. However, it should be complemented with debugging practices to capture run-time errors and observe actual program behavior.