DAY - 10 Assignment and LABs

Day 10 [Assignment 1 Time - 9 : 11].

1. File Processing:Design a base class File with a virtual function readData() that has an empty body.Create derived classes like TextFile and ImageFile inheriting from File and overriding readData() with their specific reading procedures. Implement a function that takes a pointer to File as input, attempts to read the data using the readData() function, and handles potential errors based on the actual derived class type (e.g., different file formats).

```
#include <iostream>

#include <fstream>

#include <stdexcept>


// Base class File

class File {

public:

    virtual void readData() = 0;

    virtual void processData() = 0;

};


// Derived class TextFile

class TextFile : public File {

private:

    std::string filePath;
```

```
public:
    TextFile(const std::string& path) : filePath(path) {}

    void readData() override {
        std::ifstream file(filePath);
        if (!file) {
            throw std::runtime_error("Failed to open text file");
        }

        std::string line;
        while (std::getline(file, line)) {
            std::cout << line << std::endl;
        }

        file.close();
    }

    void processData() override {
        std::cout << "Processing text data..." << std::endl;
        // Add text processing logic here
    }
};

// Derived class ImageFile
class ImageFile : public File {
private:
    std::string filePath;
public:
    ImageFile(const std::string& path) : filePath(path) {}

    void readData() override {
        std::ifstream file(filePath, std::ios::binary);
        if (!file) {
            throw std::runtime_error("Failed to open image file");
        }

        // Assume image file format is BMP
        char buffer[2];
        file.read(buffer, 2);
        if (buffer[0] != 'B' || buffer[1] != 'M') {
            throw std::runtime_error("Invalid image file format");
        }

        std::cout << "Image file read successfully" << std::endl;

        file.close();
    }

    void processData() override {
        std::cout << "Processing image data..." << std::endl;
        // Add image processing logic here
    }
```

```
};

// Function to read and process file data
void readFileData(File* file) {
    try {
        file->readData();
        file->processData();
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
}

int main() {
    TextFile textFile("example.txt");
    ImageFile imageFile("image.bmp");

    readFileData(&textFile);
    readFileData(&imageFile);

    return 0;
}
```

2. Design an abstract factory class hierarchy to create different families of products (e.g., furniture). Use pointers and runtime polymorphism. Define an abstract base class FurnitureFactory with a virtual function createChair(). Create derived classes like ModernFurnitureFactory and ClassicFurnitureFactory

that override createChair() to return pointers to concrete chair objects specific to their style. Utilize the factory pattern with runtime polymorphism to allow for flexible furniture creation based on user choice.

```
#include <iostream>


// Abstract base class for furniture products
class Chair {
public:
    virtual void describe() = 0;
};


// Concrete chair classes
class ModernChair : public Chair {
public:
    void describe() override {
        std::cout << "Modern chair" << std::endl;
    }
};


class ClassicChair : public Chair {
public:
    void describe() override {
        std::cout << "Classic chair" << std::endl;
    }
};
```

```
// Abstract factory class
class FurnitureFactory {
public:
    virtual Chair* createChair() = 0;
};

// Derived factory classes
class ModernFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ModernChair();
    }
};

class ClassicFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ClassicChair();
    }
};

int main() {
    // User choice: 0 for modern, 1 for classic
    int choice;
    std::cout << "Enter your choice (0 for modern, 1 for classic): ";
    std::cin >> choice;


    FurnitureFactory* factory;
    if (choice == 0) {
        factory = new ModernFurnitureFactory();
    } else {
        factory = new ClassicFurnitureFactory();
    }


    Chair* chair = factory->createChair();
    chair->describe();


    delete chair;
    delete factory;


    return 0;
}
```

3.

Data Structures:

3. **Create a C++ structure named Flight to represent flight information, including:**

Flight number (string)

Departure and arrival airports (strings)

Departure and arrival date/time (strings or appropriate data types)

Number of available seats (integer)

Price per seat (float)

Consider creating another structure named Passenger (optional) to store passenger details if needed (name, passport information etc.).

Functions: Develop C++ functions to: Display a list of available flights based on user-specified origin and destination airports (consider searching by date range as well). Book a specific number of seats for a chosen flight (handle cases where insufficient seats are available). Cancel a booking for a specific flight and number of seats (ensure the user cancels the correct booking). Display a list of all booked flights for a specific user (if using Passenger structure). Implement error handling for invalid user input (e.g., trying to book negative seats). Include a function to add new flights to the system (consider adding flights dynamically if needed).

Develop C++ functions to: Display a list of available flights based on user-specified origin and destination airports (consider searching by date range as well). Book a specific number of seats for a chosen flight (handle cases where insufficient seats are available).Cancel a booking for a specific flight and number of seats (ensure the user cancels the correct booking).Display a list of all booked flights for a specific user (if using Passenger structure).

Implement error handling for invalid user input (e.g., trying to book negative seats).

Include a function to add new flights to the system (consider adding flights dynamically if needed).

```
#include <iostream>

#include <vector>

#include <string>

#include <unordered_map>

#include <iomanip>


struct Flight {

    std::string flightNumber;

    std::string departureAirport;

    std::string arrivalAirport;

    std::string departureDateTime;

    std::string arrivalDateTime;

    int availableSeats;
    float pricePerSeat;
};

struct Passenger {
    std::string name;
    std::string passportNumber;
    std::unordered_map<std::string, int> bookedFlights; // flightNumber -> number of seats booked
};

class FlightSystem {
private:
    std::vector<Flight> flights;
    std::unordered_map<std::string, Passenger> passengers;

public:
    void addFlight(const Flight& flight) {
        flights.push_back(flight);
    }

    void displayAvailableFlights(const std::string& origin, const std::string& destination, const
std::string& startDate, const std::string& endDate) {
        std::cout << "Available flights from " << origin << " to " << destination << " between " << startDate
<< " and " << endDate << ":\n";
        for (const auto& flight : flights) {
            if (flight.departureAirport == origin && flight.arrivalAirport == destination) {
                std::cout << "Flight Number: " << flight.flightNumber << "\n";
                std::cout << "Departure: " << flight.departureDateTime << "\n";
                std::cout << "Arrival: " << flight.arrivalDateTime << "\n";

                std::cout << "Available Seats: " << flight.availableSeats << "\n";

                std::cout << "Price per Seat: $" << std::fixed << std::setprecision(2) << flight.pricePerSeat <<
"\n";

                std::cout << "----------------------------------\n";

            }

        }

    }



    void bookSeats(const std::string& passengerName, const std::string& passportNumber, const
std::string& flightNumber, int seats) {

        for (auto& flight : flights) {

            if (flight.flightNumber == flightNumber) {

                if (flight.availableSeats >= seats) {

                    flight.availableSeats -= seats;

                    passengers[passportNumber].name = passengerName;

                    passengers[passportNumber].passportNumber = passportNumber;

                    passengers[passportNumber].bookedFlights[flightNumber] += seats;

                    std::cout << "Booking successful. " << seats << " seats booked on flight " << flightNumber
<< ".\n";

                } else {

                    std::cout << "Insufficient seats available.\n";

                }

                return;

            }

        }

        std::cout << "Flight not found.\n";

    }
    void cancelBooking(const std::string& passportNumber, const std::string& flightNumber, int seats) {
        if (passengers.find(passportNumber) != passengers.end() &&
passengers[passportNumber].bookedFlights.find(flightNumber) !=
passengers[passportNumber].bookedFlights.end()) {
            if (passengers[passportNumber].bookedFlights[flightNumber] >= seats) {
                passengers[passportNumber].bookedFlights[flightNumber] -= seats;
                if (passengers[passportNumber].bookedFlights[flightNumber] == 0) {
                    passengers[passportNumber].bookedFlights.erase(flightNumber);
                }
                for (auto& flight : flights) {
                    if (flight.flightNumber == flightNumber) {
                        flight.availableSeats += seats;
                        std::cout << "Booking cancelled. " << seats << " seats cancelled on flight " <<
flightNumber << ".\n";
                        return;
                    }
                }
            } else {
                std::cout << "You do not have that many seats booked.\n";
            }
        } else {
            std::cout << "Booking not found.\n";
        }
    }

    void displayBookedFlights(const std::string& passportNumber) {
        if (passengers.find(passportNumber) != passengers.end()) {
            std::cout << "Flights booked for passenger " << passengers[passportNumber].name << ":\n";
            for (const auto& booking : passengers[passportNumber].bookedFlights) {
                std::cout << "Flight Number: " << booking.first << " - Seats: " << booking.second << "\n";
            }
        } else {
            std::cout << "No bookings found for this passenger.\n";
        }
    }
};

int main() {
    FlightSystem system;

    // Add sample flights
    system.addFlight({"FL123", "JFK", "LAX", "2024-07-05 08:00", "2024-07-05 11:00", 100, 300.00});
    system.addFlight({"FL124", "JFK", "LAX", "2024-07-06 09:00", "2024-07-06 12:00", 50, 350.00});
    system.addFlight({"FL125", "JFK", "SFO", "2024-07-07 10:00", "2024-07-07 13:00", 75, 400.00});

    // Display available flights
    system.displayAvailableFlights("JFK", "LAX", "2024-07-04", "2024-07-08");

    // Book seats
    system.bookSeats("John Doe", "P123456", "FL123", 2);

    // Cancel booking
    system.cancelBooking("P123456", "FL123", 1);
    // Display booked flights
    system.displayBookedFlights("P123456");

    return 0;
}
```

5**. Practice Problem Statement:**

Scenario: You're working on a data analysis project where you need to filter a list of integers based on whether they are even or odd. You want to use a lambda expression to achieve this filtering.

Task:

Define a function named filter_even_odds that takes two arguments:

const std::vector<int>& numbers: The vector containing the integer values.

bool is_even: A flag indicating whether to filter even (true) or odd (false) numbers.

Inside the function, use a lambda expression to iterate through the numbers vector.

Within the lambda, check if the current number is even using the modulo operator (%).

If the even/odd condition matches the is_even flag, add the number to a new filtered vector.

Return the filtered vector from the filter_even_odds function.

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<int> filter_even_odds(const std::vector<int>& numbers, bool is_even) {
    std::vector<int> filtered_numbers;

    // Lambda expression to filter numbers
    std::for_each(numbers.begin(), numbers.end(), [&filtered_numbers, is_even](int num) {
        if ((num % 2 == 0) == is_even) {
            filtered_numbers.push_back(num);
        }
    });

    return filtered_numbers;
}

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "Original numbers: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    std::vector<int> even_numbers = filter_even_odds(numbers, true);
    std::vector<int> odd_numbers = filter_even_odds(numbers, false);

    std::cout << "Filtered even numbers: ";
    for (int num : even_numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    std::cout << "Filtered odd numbers: ";
    for (int num : odd_numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

2. Finding Maximum Value:

Scenario: You have a list of objects and want to find the object with the highest value based on a specific criterion.

Task:

Define a function named find_max that takes two arguments:

const std::vector<T>& objects: The vector containing the objects (can be any type T).

std::function<bool(const T& a, const T& b)> compare: A function object (e.g., a lambda) that defines the comparison logic for finding the maximum.

Inside the function, use a std::accumulate with a lambda expression to iterate through the objects vector.

Within the inner lambda, compare the current element with the current maximum using the provided compare function.

If the current element is greater (based on the comparison logic), return it as the new maximum.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <numeric>

template<typename T>
T find_max(const std::vector<T>& objects, std::function<bool(const T& a, const T& b)> compare) {
    if (objects.empty()) {
        throw std::runtime_error("The input vector is empty.");
    }

    return std::accumulate(objects.begin(), objects.end(), objects[0],
        [compare](const T& max_obj, const T& current_obj) {
            return compare(max_obj, current_obj) ? current_obj : max_obj;
        });
}

int main() {
    struct Item {
        int id;
        double value;
    };

    std::vector<Item> items = {
        {1, 10.5},
        {2, 20.3},
        {3, 15.8},
        {4, 25.1},
        {5, 18.6}
    };

    auto compare = [](const Item& a, const Item& b) {
        return a.value < b.value;
    };

    try {
        Item max_item = find_max(items, compare);
        std::cout << "Item with max value: ID = " << max_item.id << ", Value = " << max_item.value <<
std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```