TASK – 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).

A: #include <iostream>

#include <string>

using namespace std;

class File { // Base class File

public:

virtual void readData() = 0; // Pure virtual function

virtual void handleError() = 0; // Pure virtual function for handling errors

virtual ~File() {}

};

class TextFile : public File { // Derived class TextFile

public:

void readData() override { // Reading text file data

cout << "Reading data from text file." << endl;

if (/\* condition indicating error \*/ false) { // Simulate an error

handleError();

} else {

cout << "Text file read successfully." << endl; }

}

void handleError() override {

cout << "Error: Could not read text file." << endl; }

};

class ImageFile : public File { // Derived class ImageFile

public:

void readData() override { // Reading image file data

cout << "Reading data from image file." << endl;

if (/\* condition indicating error \*/ false) { // Simulate an error

handleError();

} else {

cout << "Image file read successfully." << endl; }

}

void handleError() override {

cout << "Error: Could not read image file." << endl; }

};

void readFile(File\* file) { // Function to read data from a file and handle errors

file->readData(); }

int main() {

TextFile txtFile;

ImageFile imgFile;

File\* file = &txtFile;

readFile(file); // Should read text file data

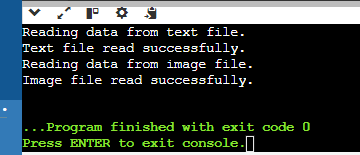
file = &imgFile;

readFile(file); // Should read image file data

return 0;

}

OUTPUT:



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

A: #include <iostream>

using namespace std;

class Chair { // Abstract product class Chair

public:

virtual void sitOn() = 0; // Pure virtual function

virtual ~Chair() {}

};

class ModernChair : public Chair { // Concrete product class ModernChair

public:

void sitOn() override {

cout << "Sitting on a modern chair." << endl; }

};

class ClassicChair : public Chair { // Concrete product class ClassicChair

public:

void sitOn() override {

cout << "Sitting on a classic chair." << endl; }

};

class FurnitureFactory {

public:

virtual Chair\* createChair() = 0; // Pure virtual function

virtual ~FurnitureFactory() {}

};

class ModernFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ModernChair(); }

};

class ClassicFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ClassicChair(); }

};

void createFurniture(FurnitureFactory\* factory) { // Function to create furniture

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

chair->sitOn();

delete chair; // Clean up

}

int main() {

FurnitureFactory\* factory = nullptr; // User choice: Modern or Classic

string choice;

cout << "Enter furniture style (modern/classic): ";

cin >> choice;

if (choice == "modern") {

factory = new ModernFurnitureFactory();

} else if (choice == "classic") {

factory = new ClassicFurnitureFactory();

} else {

cout << "Invalid choice." << endl;

return 1;

}

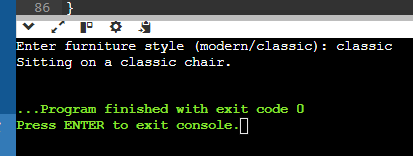
createFurniture(factory);

delete factory; // Clean up

return 0;

}

OUTPUT:



TASK – 2

Problem Statement: Flight Booking System (C++ Structures & Functions)

Data Structures:

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.).

A: #include <iostream>

#include <string>

using namespace std;

struct Flight { // Structure to represent flight information

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime;

string arrivalDateTime;

int availableSeats;

float pricePerSeat;

void display() const { // Function to display flight information

cout << "Flight Number: " << flightNumber << endl;

cout << "Departure Airport: " << departureAirport << endl;

cout << "Arrival Airport: " << arrivalAirport << endl;

cout << "Departure DateTime: " << departureDateTime << endl;

cout << "Arrival DateTime: " << arrivalDateTime << endl;

cout << "Available Seats: " << availableSeats << endl;

cout << "Price Per Seat: $" << pricePerSeat << endl; }

};

struct Passenger {

string name;

string passportNumber;

void display() const {

cout << "Name: " << name << endl;

cout << "Passport Number: " << passportNumber << endl; }

};

int main() {

Flight flight1 = {

"AB123",

"JFK",

"LAX",

"2024-08-01 10:00",

"2024-08-01 13:00",

150,

299.99

};

flight1.display();

Passenger passenger1 = {

"John Doe",

"X12345678"

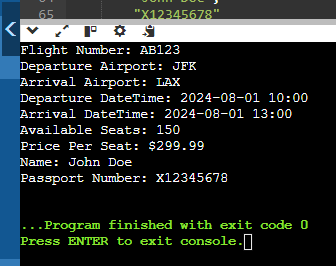
};

passenger1.display();

return 0;

}

OUTPUT:



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).

A: #include <iostream>

#include <vector>

#include <string>

using namespace std;

class Flight {

private:

string flightNumber;

string origin;

string destination;

string date;

int availableSeats;

public:

Flight(string flightNumber, string origin, string destination, string date, int availableSeats)

: flightNumber(flightNumber), origin(origin), destination(destination), date(date), availableSeats(availableSeats) {}

string getFlightNumber() const { return flightNumber; }

string getOrigin() const { return origin; }

string getDestination() const { return destination; }

string getDate() const { return date; }

int getAvailableSeats() const { return availableSeats; }

void displayFlightDetails() const {

cout << "Flight Number: " << flightNumber << endl;

cout << "Origin: " << origin << endl;

cout << "Destination: " << destination << endl;

cout << "Date: " << date << endl;

cout << "Available Seats: " << availableSeats << endl; }

void bookSeats(int numSeats) {

if (numSeats > 0 && numSeats <= availableSeats) {

availableSeats -= numSeats;

cout << numSeats << " seats booked successfully for flight " << flightNumber << endl;

} else {

cout << "Error: Insufficient seats available." << endl; }

}

void cancelBooking(int numSeats) {

if (numSeats > 0 && numSeats <= availableSeats) {

availableSeats += numSeats;

cout << numSeats << " seats cancelled successfully for flight " << flightNumber << endl;

} else {

cout << "Error: Invalid number of seats to cancel." << endl; } }

};

class FlightManager {

private:

vector<Flight> flights;

public:

void addFlight(const Flight& flight) {

flights.push\_back(flight); }

void displayAvailableFlights(const string& origin, const string& destination, const string& date) {

for (const auto& flight : flights) {

if (flight.getOrigin() == origin && flight.getDestination() == destination && flight.getDate() == date) {

flight.displayFlightDetails();

cout << endl; } }

}

void bookSeats(const string& flightNumber, int numSeats) {

for (auto& flight : flights) {

if (flight.getFlightNumber() == flightNumber) {

flight.bookSeats(numSeats);

return; }

}

cout << "Flight with number " << flightNumber << " not found." << endl;

}

void cancelBooking(const string& flightNumber, int numSeats) {

for (auto& flight : flights) {

if (flight.getFlightNumber() == flightNumber) {

flight.cancelBooking(numSeats);

return; }

}

cout << "Flight with number " << flightNumber << " not found." << endl; }

};

int main() {

FlightManager manager;

manager.addFlight(Flight("F001", "JFK", "LAX", "2024-07-05", 150));

manager.addFlight(Flight("F002", "LAX", "JFK", "2024-07-06", 200));

cout << "Available Flights from JFK to LAX on 2024-07-05:" << endl;

manager.displayAvailableFlights("JFK", "LAX", "2024-07-05");

cout << "Enter number of seats to book: ";

int seatsToBook;

cin >> seatsToBook;

manager.bookSeats("F001", seatsToBook);

cout << "Enter number of seats to cancel: ";

int seatsToCancel;

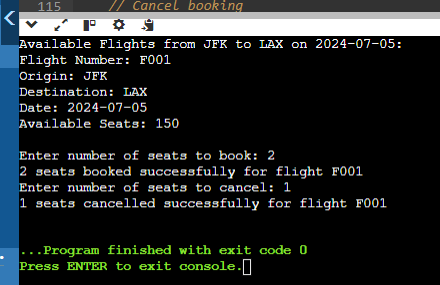
cin >> seatsToCancel;

manager.cancelBooking("F001", seatsToCancel);

return 0;

}

OUTPUT:



-------------------------------------------------------------------------------------------------------------------------------

TASK - 3

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.

A: #include <iostream>

#include <vector>

using namespace std;

vector<int> filter\_even\_odds(const vector<int>& numbers, bool is\_even) {

vector<int> filtered;

for (int num : numbers) {

if (is\_even == (num % 2 == 0)) {

filtered.push\_back(num);

}

}

return filtered;

}

int main() {

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

// Filter even numbers

vector<int> evens = filter\_even\_odds(numbers, true);

cout << "Even numbers: ";

for (int num : evens) {

cout << num << " ";

}

cout << endl;

// Filter odd numbers

vector<int> odds = filter\_even\_odds(numbers, false);

cout << "Odd numbers: ";

for (int num : odds) {

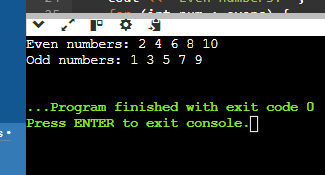
cout << num << " ";

}

cout << 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 <functional>

#include <algorithm>

using namespace std;

template <typename T, typename Compare>

T find\_max(const vector<T>& objects, Compare compare) {

return \*std::max\_element(objects.begin(), objects.end(), compare);

}

struct Object {

int value;

string name;

};

int main() {

vector<Object> objects = {{10, "Object1"}, {30, "Object2"}, {20, "Object3"}, {40, "Object4"}};

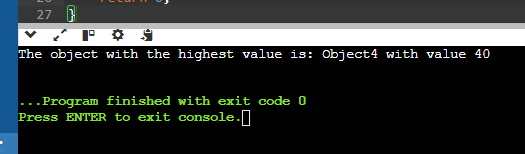
auto compare = [](const Object& a, const Object& b) { return a.value < b.value; };

Object max\_obj = find\_max(objects, compare);

cout << "The object with the highest value is: " << max\_obj.name << " with value " << max\_obj.value << endl;

return 0;

}

****