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 <exception>

class File {

public:

virtual void readData() = 0;

virtual ~File() = default;

};

class TextFile : public File {

public:

void readData() override {

std::cout << "Read data from text file." << std::endl;

throw std::runtime\_error("Text file read error.");

}

};

class ImageFile : public File {

public:

void readData() override {

std::cout << "Read data from image file." << std::endl;

throw std::runtime\_error("Image file read error.");

}

};

void processFile(File\* file) {

try {

file->readData();

} catch (const std::exception& e) {

std::cerr << "Error reading file: " << e.what() << std::endl;

}

}

int main() {

TextFile textFile;

ImageFile imageFile;

std::cout << "Process text file:" << std::endl;

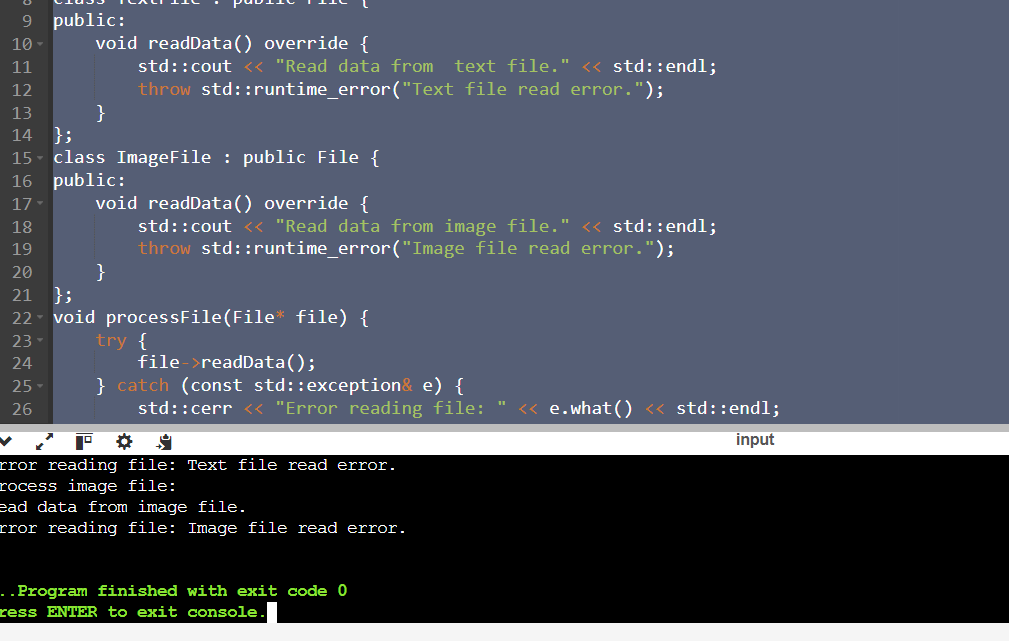
processFile(&textFile);

std::cout << "Process image file:" << std::endl;

processFile(&imageFile);

return 0;

}



1. 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>

#include <memory>

class Chair {

public:

virtual void sitOn() const = 0;

virtual ~Chair() = default;

};

class ModernChair : public Chair {

public:

void sitOn() const override {

std::cout << "Sitting on a modern chair." << std::endl;

}

};

class ClassicChair : public Chair {

public:

void sitOn() const override {

std::cout << "Sitting on a classic chair." << std::endl;

}

};

class FurnitureFactory {

public:

virtual Chair\* createChair() const = 0;

virtual ~FurnitureFactory() = default;

};

class ModernFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() const override {

return new ModernChair();

}

};

class ClassicFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() const override {

return new ClassicChair();

}

};

void createAndUseChair(const FurnitureFactory& factory) {

Chair\* chair = factory.createChair();

chair->sitOn();

delete chair;

}

int main() {

ModernFurnitureFactory modernFactory;

ClassicFurnitureFactory classicFactory;

std::cout << "Creat a modern furniture:" << std::endl;

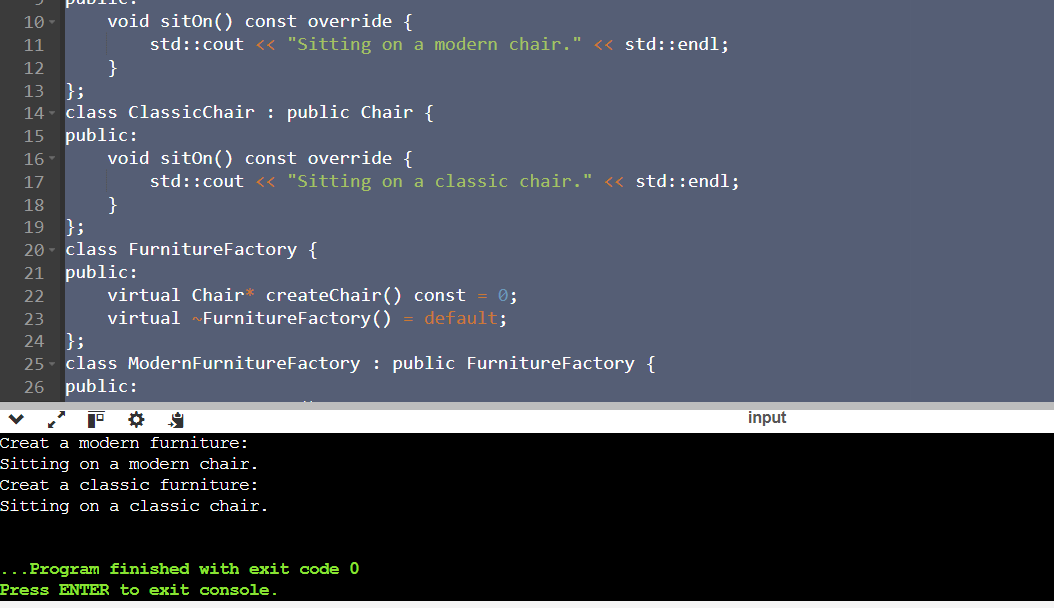
createAndUseChair(modernFactory);

std::cout << "Creat a classic furniture:" << std::endl;

createAndUseChair(classicFactory);

return 0;

}



1. 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.).

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).

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <limits>

using namespace std;

struct Flight {

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime;

string arrivalDateTime;

int availableSeats;

float pricePerSeat;

};

struct Passenger {

string name;

string passportNumber;

vector<string> bookedFlights;

};

void addFlight(vector<Flight>& flights);

void displayFlights(const vector<Flight>& flights);

void searchFlights(const vector<Flight>& flights);

void bookSeats(vector<Flight>& flights, Passenger& passenger);

void cancelBooking(vector<Flight>& flights, Passenger& passenger);

void displayBookedFlights(const Passenger& passenger);

void handleInvalidInput();

int main() {

vector<Flight> flights;

Passenger passenger;

int choice;

while (true) {

cout << "\nFlight Booking System\n";

cout << "1. Add Flight\n";

cout << "2. Display Flights\n";

cout << "3. Search Flights\n";

cout << "4. Book Seats\n";

cout << "5. Cancel Booking\n";

cout << "6. Display Booked Flights\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

if (cin.fail()) {

handleInvalidInput();

continue;

}

switch (choice) {

case 1:

addFlight(flights);

break;

case 2:

displayFlights(flights);

break;

case 3:

searchFlights(flights);

break;

case 4:

bookSeats(flights, passenger);

break;

case 5:

cancelBooking(flights, passenger);

break;

case 6:

displayBookedFlights(passenger);

break;

case 7:

cout << "Exiting...\n";

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

return 0;

}

void addFlight(vector<Flight>& flights) {

Flight newFlight;

cout << "Enter flight number: ";

cin >> newFlight.flightNumber;

cout << "Enter departure airport: ";

cin >> newFlight.departureAirport;

cout << "Enter arrival airport: ";

cin >> newFlight.arrivalAirport;

cout << "Enter departure date/time (YYYY-MM-DD HH:MM): ";

cin.ignore();

getline(cin, newFlight.departureDateTime);

cout << "Enter arrival date/time (YYYY-MM-DD HH:MM): ";

getline(cin, newFlight.arrivalDateTime);

cout << "Enter number of available seats: ";

cin >> newFlight.availableSeats;

cout << "Enter price per seat: ";

cin >> newFlight.pricePerSeat;

flights.push\_back(newFlight);

cout << "Flight added successfully!\n";

}

void displayFlights(const vector<Flight>& flights) {

if (flights.empty()) {

cout << "No flights available.\n";

return;

}

for (const auto& flight : flights) {

cout << "\nFlight Number: " << flight.flightNumber

<< "\nDeparture Airport: " << flight.departureAirport

<< "\nArrival Airport: " << flight.arrivalAirport

<< "\nDeparture Date/Time: " << flight.departureDateTime

<< "\nArrival Date/Time: " << flight.arrivalDateTime

<< "\nAvailable Seats: " << flight.availableSeats

<< "\nPrice per Seat: $" << flight.pricePerSeat << "\n";

}

}

void searchFlights(const vector<Flight>& flights) {

string origin, destination, startDate, endDate;

cout << "Enter origin airport: ";

cin >> origin;

cout << "Enter destination airport: ";

cin >> destination;

cout << "Enter start date: ";

cin.ignore();

getline(cin, startDate);

cout << "Enter end date: ";

getline(cin, endDate);

bool found = false;

for (const auto& flight : flights) {

if (flight.departureAirport == origin && flight.arrivalAirport == destination &&

flight.departureDateTime >= startDate && flight.departureDateTime <= endDate) {

cout << "\nFlight Number: " << flight.flightNumber

<< "\nDeparture Airport: " << flight.departureAirport

<< "\nArrival Airport: " << flight.arrivalAirport

<< "\nDeparture Date/Time: " << flight.departureDateTime

<< "\nArrival Date/Time: " << flight.arrivalDateTime

<< "\nAvailable Seats: " << flight.availableSeats

<< "\nPrice per Seat: $" << flight.pricePerSeat << "\n";

found = true;

}

}

if (found) {

cout << "No flights found for the given criteria.\n";

}

}

void bookSeats(vector<Flight>& flights, Passenger& passenger) {

string flightNumber;

int seatsToBook;

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

cin >> flightNumber;

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

cin >> seatsToBook;

if (cin.fail() || seatsToBook <= 0) {

handleInvalidInput();

return;

}

for (auto& flight : flights) {

if (flight.flightNumber == flightNumber) {

if (flight.availableSeats >= seatsToBook) {

flight.availableSeats -= seatsToBook;

passenger.bookedFlights.push\_back(flightNumber);

cout << "Seats booked successfully! Remaining seats: " << flight.availableSeats << "\n";

}

else {

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

}

return;

}

}

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

}

void cancelBooking(vector<Flight>& flights, Passenger& passenger) {

string flightNumber;

int seatsToCancel;

cout << "Enter flight number to cancel booking: ";

cin >> flightNumber;

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

cin >> seatsToCancel;

if (cin.fail() || seatsToCancel <= 0) {

handleInvalidInput();

return;

}

auto it = find(passenger.bookedFlights.begin(), passenger.bookedFlights.end(), flightNumber);

if (it != passenger.bookedFlights.end()) {

for (auto& flight : flights) {

if (flight.flightNumber == flightNumber) {

flight.availableSeats += seatsToCancel;

passenger.bookedFlights.erase(it);

cout << "Booking canceled successfully! Remaining seats: " << flight.availableSeats << "\n";

return;

}

}

}

else

{

cout << "No booking found for the specified flight.\n";

}

}

void displayBookedFlights(const Passenger& passenger) {

if (passenger.bookedFlights.empty()) {

cout << "No booked flights.\n";

return;

}

cout << "Booked Flights for " << passenger.name << ":\n";

for (const auto& flightNumber : passenger.bookedFlights) {

cout << flightNumber << "\n";

}

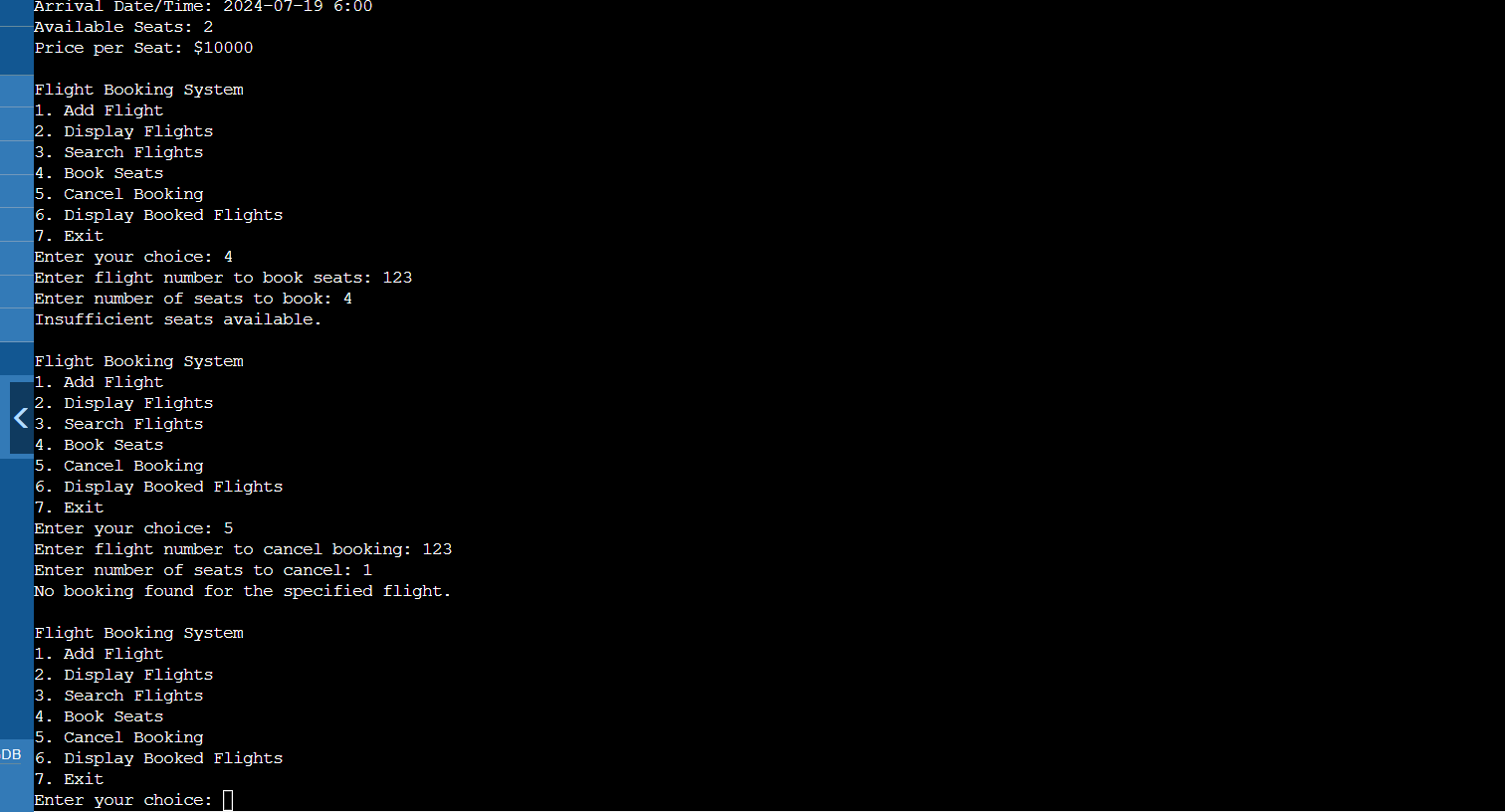
}

void handleInvalidInput() {

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cout << "Invalid input. Please try again.\n";

}

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;

auto filter = [is\_even](int num) {

return is\_even ? (num % 2 == 0) : (num % 2 != 0);

};

std::copy\_if(numbers.begin(), numbers.end(), std::back\_inserter(filtered\_numbers), filter);

return filtered\_numbers;

}

int main() {

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

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

std::cout << "Even numbers: ";

for (int num : evens) {

std::cout << num << " ";

}

std::cout << std::endl;

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

std::cout << "Odd numbers: ";

for (int num : odds) {

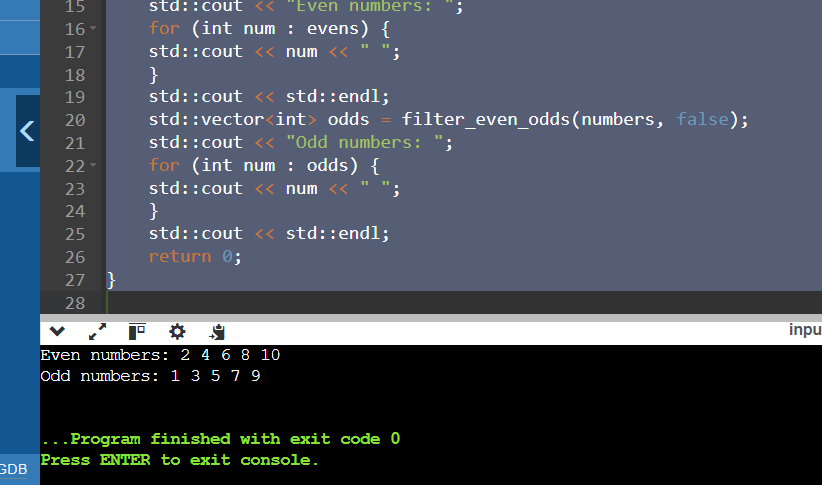
std::cout << num << " ";

}

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>

template <typename T, typename Compare>

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

if (objects.empty()) {

throw std::invalid\_argument("The objects vector is empty");

}

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

}

int main() {

std::vector<int> numbers = {1, 5, 3, 9, 2};

int max\_number = find\_max(numbers, [](const int& a, const int& b) {

return a < b;

});

std::cout << "Max number: " << max\_number << std::endl;

struct Person {

std::string name;

int age;

};

std::vector<Person> people = {{"ritik", 30}, {"Boby", 25}, {"georg", 35}};

Person oldest = find\_max(people, [](const Person& a, const Person& b) {

return a.age < b.age;

});

std::cout << "Oldest person: " << oldest.name << " (" << oldest.age << ")" << std::endl;

return 0;

}

