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>

#include <fstream>

#include <string>

// Base class

class File {

public:

virtual ~File() = default;

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

};

// Derived class for text files

class TextFile : public File {

private:

std::string filename;

public:

TextFile(const std::string& fname) : filename(fname) {}

void readData() override {

std::ifstream file(filename);

if (!file.is\_open()) {

throw std::runtime\_error("Could not open text file: " + filename);

}

std::string line;

while (std::getline(file, line)) {

std::cout << line << std::endl;

}

file.close();

}

};

// Derived class for image files

class ImageFile : public File {

private:

std::string filename;

public:

ImageFile(const std::string& fname) : filename(fname) {}

void readData() override {

// For simplicity, we'll just simulate reading an image file

std::ifstream file(filename, std::ios::binary);

if (!file.is\_open()) {

throw std::runtime\_error("Could not open image file: " + filename);

}

std::cout << "Reading image file: " + filename << std::endl;

// Simulated image processing code

file.close();

}

};

void processFile(File\* file) {

try {

file->readData();

} catch (const std::runtime\_error& e) {

std::cout << "Error: " << e.what() << std::endl;

}

}

int main() {

File\* textFile = new TextFile("example.txt");

File\* imageFile = new ImageFile("example.jpg");

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

processFile(textFile);

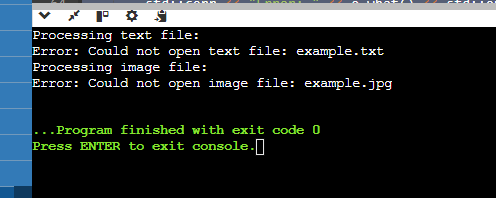
std::cout << "Processing image file:" << std::endl;

processFile(imageFile);

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

#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 with sleek design" << std::endl;

}

};

class ClassicChair : public Chair {

public:

void describe() override {

std::cout << "Classic chair with ornate details" << 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();

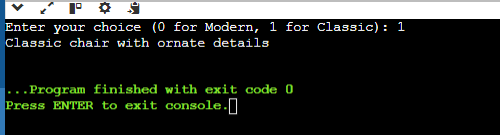
delete chair;

delete factory;

return 0;

}

Output:



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() {

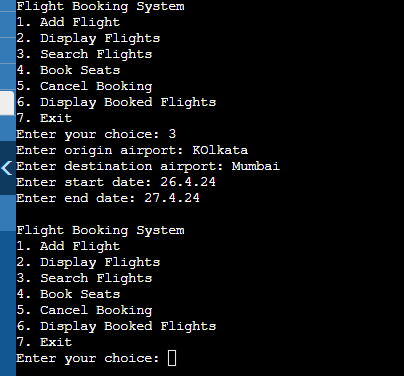
cin.clear();

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

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

}

Output:



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.

#define FILTER\_EVEN\_ODDS\_H

#include<iostream>

#include <vector>

#include <algorithm>

using namespace std;

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

vector<int> filtered\_numbers;

auto is\_match = [is\_even](int number) {

return (number % 2 == 0) == is\_even;

};

copy\_if(numbers.begin(), numbers.end(), back\_inserter(filtered\_numbers), is\_match);

return filtered\_numbers;

}

int main() {

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

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

cout << "Even numbers: ";

for (int num : even\_numbers) {

cout << num << " ";

}

cout << endl;

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

cout << "Odd numbers: ";

for (int num : odd\_numbers) {

cout << num << " ";

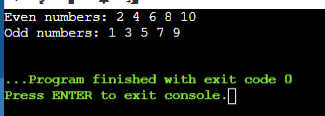
}

cout << endl;

return 0;

}

Output:



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.

define FIND\_MAX\_H

#include<iostream>

#include <vector>

#include <functional>

#include <numeric>

#include <stdexcept>

using namespace std;

template <typename T>

T find\_max(const vector<T>& objects, function<bool(const T& a, const T& b)> compare) {

if (objects.empty()) {

throw invalid\_argument("The vector is empty.");

}

return accumulate(objects.begin() + 1, objects.end(), objects[0],

[&compare](const T& max\_obj, const T& current\_obj) {

return compare(current\_obj, max\_obj) ? current\_obj : max\_obj;

});

}

struct Item {

int id;

double value;

Item(int i, double v) : id(i), value(v) {}

};

int main() {

vector<Item> items = {

Item(1, 10.5),

Item(2, 20.2),

Item(3, 15.3),

Item(4, 22.1),

Item(5, 18.7)

};

auto compare = [](const Item& a, const Item& b) {

return a.value > b.value;

};

try {

Item max\_item = find\_max (items, compare);

cout << "Item with max value: ID = " << max\_item.id << ", Value = " << max\_item.value << endl;

} catch (const invalid\_argument& e) {

cout << "Error: " << e.what() << endl;

}

return 0;

}

Output:

