**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 <string>

using namespace std;

// Base class File

class File {

public:

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

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

virtual ~File() {}

};

// Derived class TextFile

class TextFile : public File {

public:

void readData() override {

// Simulate reading text file data

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

// Simulate an error

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

handleError();

} else {

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

}

}

void handleError() override {

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

}

};

// Derived class ImageFile

class ImageFile : public File {

public:

void readData() override {

// Simulate reading image file data

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

// Simulate an error

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

handleError();

} else {

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

}

}

void handleError() override {

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

}

};

// Function to read data from a file and handle errors

void readFile(File\* file) {

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:**

**A computer screen shot of a black and white screen

Description automatically generated**

**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>

using namespace std;

// Abstract product class Chair

class Chair {

public:

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

virtual ~Chair() {}

};

// Concrete product class ModernChair

class ModernChair : public Chair {

public:

void sitOn() override {

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

}

};

// Concrete product class ClassicChair

class ClassicChair : public Chair {

public:

void sitOn() override {

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

}

};

// Abstract factory class FurnitureFactory

class FurnitureFactory {

public:

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

virtual ~FurnitureFactory() {}

};

// Concrete factory class ModernFurnitureFactory

class ModernFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ModernChair();

}

};

// Concrete factory class ClassicFurnitureFactory

class ClassicFurnitureFactory : public FurnitureFactory {

public:

Chair\* createChair() override {

return new ClassicChair();

}

};

// Function to create furniture based on user choice

void createFurniture(FurnitureFactory\* factory) {

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-1:**

**A computer screen with white text

Description automatically generated**

**Output-2:**

**A screen shot of a computer program

Description automatically generated**

**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.).**

#include <iostream>

#include <string>

using namespace std;

// Structure for storing Passenger details

struct Passenger {

string name;

string passportNumber;

// Add more fields as needed (e.g., contact information, seat number, etc.)

};

// Structure to represent Flight information

struct Flight {

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime; // Consider using appropriate date/time data type (e.g., std::chrono)

string arrivalDateTime; // Consider using appropriate date/time data type (e.g., std::chrono)

int availableSeats;

float pricePerSeat;

// Optional field for storing Passenger details

Passenger passengerInfo;

// Function to display flight details

void displayFlightDetails() {

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

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

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

cout << "Departure Date/Time: " << departureDateTime << endl;

cout << "Arrival Date/Time: " << arrivalDateTime << endl;

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

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

}

};

int main() {

// Example usage of Flight structure

Flight flight1;

flight1.flightNumber = "BA123";

flight1.departureAirport = "DEL";

flight1.arrivalAirport = "RJY";

flight1.departureDateTime = "2024-05-06 04:30";

flight1.arrivalDateTime = "2024-05-06 20:00";

flight1.availableSeats = 150;

flight1.pricePerSeat = 71.85;

// Display flight details

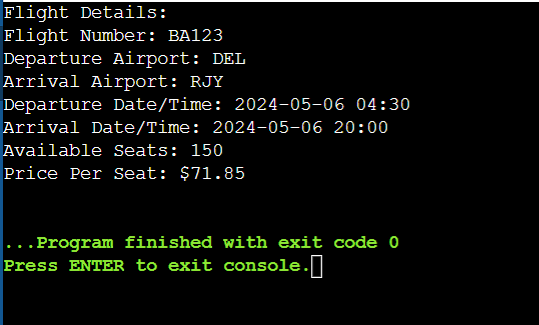
cout << "Flight Details:" << endl;

flight1.displayFlightDetails();

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).**

#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", "DEL", "RJY", "2024-06-05", 150));

manager.addFlight(Flight("F002", "RJY", "DEL", "2024-06-05", 200));

cout << "Available Flights from DEL to RJY on 2024-07-05:" << endl;

manager.displayAvailableFlights("DEL", "RJY", "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:**

**A screenshot of a computer screen

Description automatically generated**

**Use Case:**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

void assign(int& v)

{

static int n = 1; v = n++;

}

void print(int v)

{

cout<<v<<" ";

}

int main()

{

vector<int>vec(10);

for\_each(vec.begin(),vec.end(), print);

cout<<endl;

for\_each(vec.begin(),vec.end(), assign);

for\_each(vec.begin(),vec.end(), print);

return 0;

}

**Output:**

**A screen shot of a computer

Description automatically generated**

**Capture by value:**

#include <iostream>

using namespace std;

void lambda\_value\_capture() {

int value = 1;

// Capture value by value

auto copy\_value = [value]() {

return value;

};

value = 100; // Changing the original value does not affect the captured value in lambda

auto stored\_value = copy\_value(); // Call the lambda to get the captured value

cout << "stored\_value = " << stored\_value << endl; // This will print 1, not 100

}

int main() {

lambda\_value\_capture();

return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Capture by reference:**

#include <iostream>

using namespace std;

void lambda\_reference\_capture() {

int value = 1;

// Capture value by value

auto copy\_value = [&value]() {

return value;

};

value = 100; // Changing the original value does not affect the captured value in lambda

auto stored\_value = copy\_value(); // Call the lambda to get the captured value

cout << "stored\_value = " << stored\_value << endl; // This will print 1, not 100

}

int main() {

lambda\_reference\_capture();

return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Capture by both value and reference:**

#include <iostream>

using namespace std;

int main() {

int m = 0;

int n = 0;

[&,n](int a) mutable { m= ++n +a;}(4);

cout<<m<<" "<<endl;

}

**Output:**

**A screen shot of a computer

Description automatically generated**

**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>

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;

}

**Output:**

**A screen shot of a computer

Description automatically generated**

**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;

}

**Output:**

**A computer screen with text

Description automatically generated**