MOUNIKA Day10 c++

**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;

class File {

public:

virtual void readData() = 0;

virtual ~File() {}

};

class TextFile : public File {

public:

void readData() override {

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

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

handleError();

} else {

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

}

}

void handleError() {

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

}

};

class ImageFile : public File {

public:

void readData() override {

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

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

handleError();

} else {

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

}

}

void handleError() {

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

}

};

class FileFactory {

public:

virtual File\* createFile() = 0;

virtual ~FileFactory() {}

};

class TextFileFactory : public FileFactory {

public:

File\* createFile() override {

return new TextFile();

}

};

class ImageFileFactory : public FileFactory {

public:

File\* createFile() override {

return new ImageFile();

}

};

void readFile(FileFactory\* factory) {

File\* file = factory->createFile();

file->readData();

delete file;

}

int main() {

FileFactory\* factory = nullptr;

string choice;

cout << "Enter file type (text/image): ";

cin >> choice;

if (choice == "text") {

factory = new TextFileFactory();

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

factory = new ImageFileFactory();

} else {

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

return 1;

}

readFile(factory);

delete factory;

return 0;

}

OUTPUT:

Enter file type (text/image): image

Reading data from image file.

Image file read successfully.

**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:

Enter furniture style (modern/classic): modern

Sitting on a modern chair.

**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; // Using directive to avoid std:: prefixes

// Structure to represent flight information

struct Flight {

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime; // Can use appropriate date/time data types (e.g., std::chrono::time\_point)

string arrivalDateTime; // Can use appropriate date/time data types (e.g., std::chrono::time\_point)

int availableSeats;

float pricePerSeat;

// Constructor for initializing all members

Flight(const string& number, const string& depAirport, const string& arrAirport,

const string& depDateTime, const string& arrDateTime,

int seats, float price)

: flightNumber(number), departureAirport(depAirport), arrivalAirport(arrAirport),

departureDateTime(depDateTime), arrivalDateTime(arrDateTime),

availableSeats(seats), pricePerSeat(price) {}

// Function to display flight details

void displayFlightDetails() const {

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;

}

};

// Structure to represent passenger information

struct Passenger {

string name;

string passportNumber;

// Constructor for initializing passenger details

Passenger(const string& passengerName, const string& passport)

: name(passengerName), passportNumber(passport) {}

// Function to display passenger details

void displayPassengerDetails() const {

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

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

}

};

// Function to update the number of available seats

void updateAvailableSeats(Flight& flight, int newSeatsCount) {

flight.availableSeats = newSeatsCount;

}

// Function to calculate total price for given number of seats

float calculateTotalPrice(const Flight& flight, int numberOfSeats) {

return flight.pricePerSeat \* numberOfSeats;

}

// Function to update passport number

void updatePassportNumber(Passenger& passenger, const string& newPassportNumber) {

passenger.passportNumber = newPassportNumber;

}

int main() {

// Create a Flight object

Flight flight1("ABC123", "JFK", "LAX", "2024-07-05 12:00", "2024-07-05 15:00", 150, 350.0);

// Display flight details

cout << "Flight Information:" << endl;

flight1.displayFlightDetails();

cout << endl;

// Optional: Create a Passenger object

Passenger passenger1("John Doe", "ABC123456");

// Optional: Display passenger details

cout << "Passenger Information:" << endl;

passenger1.displayPassengerDetails();

cout << endl;

return 0;

}

OUTPUT:  
Flight Information:

Flight Number: ABC123

Departure Airport: JFK

Arrival Airport: LAX

Departure Date/Time: 2024-07-05 12:00

Arrival Date/Time: 2024-07-05 15:00

Available Seats: 150

Price per Seat: $350

Passenger Information:

Passenger Name: John Doe

Passport Number: ABC123456

**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", "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:

Available Flights from JFK to LAX on 2024-07-05:

Flight Number: F001

Origin: JFK

Destination: LAX

Date: 2024-07-05

Available Seats: 150

Enter number of seats to book: 2

2 seats booked successfully for flight F001

Enter number of seats to cancel: 1

1 seats cancelled successfully for flight F001

**LAMBDA FUNCTION:**

#include<iostream>

int multiply(int a,int b);

int main(){

std::cout << multiply(4, 5) << std::endl;

std::cout << [](int a, int b) {return a\*b; }(4, 5) << std::endl;

auto f = [](int a, int b) { return a\*b; };

std::cout << f(4, 5) << std::endl;

}

int multiply(int a, int b){

return a\*b;

}

OUTPUT:

20

20

20

**1.CAPTURE BY VALUE:**

#include <iostream>

void lambda\_value\_capture() {

int value = 1;

auto copy\_value = [value] {

return value;

};

value = 100;

auto stored\_value = copy\_value();

std::cout << "stored\_value = " << stored\_value << std::endl;

// At this moment, stored\_value == 1, and value == 100.

// Because copy\_value has copied when it was created.

}

int main() {

lambda\_value\_capture();

return 0;

}

OUTPUT:

stored\_value = 1

1. **REFRENCE CAPTURE:**

#include <iostream>

void lambda\_reference\_capture() {

int value = 1;

auto copy\_value = [&value] {

return value;

};

value = 100;

auto stored\_value = copy\_value();

std::cout << "stored\_value = " << stored\_value << std::endl;

// At this moment, stored\_value == 100, value == 100.

// Because copy\_value stores reference

}

int main() {

lambda\_reference\_capture();

return 0;

}

OUTPUT:

stored\_value = 100

**CAPTURED BY BOTH:**

#include <iostream>

using namespace std;

int main() {

int m = 0;

int n = 0;

// Lambda captures m by reference and n by value

[&, n] (int a) mutable {

m = ++n + a;

}(4);

cout << m << endl; // Outputs the value of m

cout << n << endl; // Outputs the value of n

return 0;

}

OUTPUT:

5

0

**USE CASE:**

// lambda.cpp

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

// Function to assign a value to each element of a vector

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);

cout << endl;

return 0;

}

OUTPUT:

0 0 0 0 0 0 0 0 0 0

1 2 3 4 5 6 7 8 9 10

**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.**

[3:31 pm, 5/7/2024] Manu Wipro: Haa

[3:40 pm, 5/7/2024] Manu Wipro: #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:

The object with the highest value is: Object4 with value 40

**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>

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:

Even numbers: 2 4 6 8 10

Odd numbers: 1 3 5 7 9