**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).**

#ifndef FILEPROCESSING\_H

#define FILEPROCESSING\_H

#include <iostream>

#include <fstream>

#include <exception>

using namespace std;

namespace FileProcessing {

class File {

public:

virtual void readData() = 0;

virtual ~File() {} //

};

class TextFile : public File {

public:

void readData() override;

};

class ImageFile : public File {

public:

void readData() override;

};

void processFile(File\* file);

}

#endif // FILEPROCESSING\_H

namespace FileProcessing {

void TextFile::readData() {

ifstream file("example.txt");

if (!file) {

cerr << "Error opening text file" << endl;

return;

}

string line;

while (getline(file, line)) {

cout << "TextFile: " << line << endl;

}

file.close();

}

void ImageFile::readData() {

ifstream file("example.img", ios::binary);

if (!file) {

cerr << "Error opening image file" << endl;

return;

}

char buffer[256];

while (file.read(buffer, sizeof(buffer))) {

cout << "ImageFile: Read " << file.gcount() << " bytes" << endl;

}

file.close();

}

void processFile(File\* file) {

file->readData();

}

}

using namespace FileProcessing;

int main() {

TextFile txtFile;

ImageFile imgFile;

cout << "Processing TextFile:" << endl;

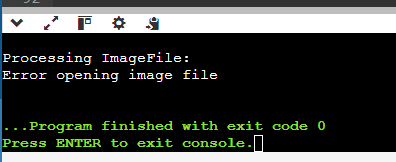
processFile(&txtFile);

cout << "\nProcessing ImageFile:" << endl;

processFile(&imgFile);

return 0;

}



Or

#include <iostream>

#include <string>

using namespace std;

class File { // Base class File

public:

virtual void readData() = 0; // Virtual function for reading data

};

class TextFile : public File { // Derived class TextFile

public:

void readData() override { // Override readData() for text files

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

cout << "Text data read successfully." << endl; // Simulate reading text file data

}

};

class ImageFile : public File { // Derived class ImageFile

public:

void readData() override { // Override readData() for image files

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

cout << "Image data read successfully." << endl; // Simulate reading image file data

}

};

void processFile(File\* file) { // Function to process file based on its type

try {

file->readData(); // Attempt to read data using polymorphism

}

catch (const exception& e) {

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

}

catch (...) {

cerr << "Unknown error occurred while reading file." << endl;

}

}

int main() {

TextFile txtFile; // Example usage:

ImageFile imgFile;

cout << "Processing Text File:" << endl; // Process a text file

processFile(&txtFile);

cout << endl;

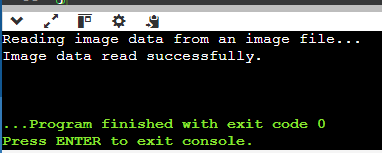
cout << "Processing Image File:" << endl; // Process an image file

processFile(&imgFile);

cout << endl;

return 0;

}



**2. 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>

// Abstract product: Chair

class Chair {

public:

virtual ~Chair() = default;

virtual void sitOn() const = 0;

};

// Concrete product: ModernChair

class ModernChair : public Chair {

public:

void sitOn() const override {

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

}

};

// Concrete product: ClassicChair

class ClassicChair : public Chair {

public:

void sitOn() const override {

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

}

};

// Abstract factory: FurnitureFactory

class FurnitureFactory {

public:

virtual ~FurnitureFactory() = default;

virtual std::unique\_ptr<Chair> createChair() const = 0;

};

// Concrete factory: ModernFurnitureFactory

class ModernFurnitureFactory : public FurnitureFactory {

public:

std::unique\_ptr<Chair> createChair() const override {

return std::make\_unique<ModernChair>();

}

};

// Concrete factory: ClassicFurnitureFactory

class ClassicFurnitureFactory : public FurnitureFactory {

public:

std::unique\_ptr<Chair> createChair() const override {

return std::make\_unique<ClassicChair>();

}

};

// Client code

void clientCode(const FurnitureFactory& factory) {

auto chair = factory.createChair();

chair->sitOn();

}

int main() {

ModernFurnitureFactory modernFactory;

ClassicFurnitureFactory classicFactory;

std::cout << "Modern Furniture Factory:" << std::endl;

clientCode(modernFactory);

std::cout << "Classic Furniture Factory:" << std::endl;

clientCode(classicFactory);

return 0;

}

**DATA STRUCTURES:**

**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;

struct Flight {

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime;

string arrivalDateTime;

int availableSeats;

float pricePerSeat;

};

struct Passenger {

string name;

string passportInfo;

};

void displayFlightInfo(const Flight& flight) {

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

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

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

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

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

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

cout << "Price per Seat: $" << flight.pricePerSeat << endl;

}

int main() {

Flight flight1 = {

"AA123",

"JFK",

"LAX",

"2024-07-05 10:00",

"2024-07-05 13:00",

150,

299.99

};

Passenger passenger1 = {

"John Doe",

"P12345678"

};

displayFlightInfo(flight1);

cout << "\nPassenger Name: " << passenger1.name << endl;

cout << "Passport Info: " << passenger1.passportInfo << endl;

return 0;

}

**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).**

**REQUIREMENTS OF THE QUESTION:**

#ifndef FLIGHTMANAGEMENT\_H

#define FLIGHTMANAGEMENT\_H

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

namespace FlightManagement {

struct Flight {

string flightNumber;

string departureAirport;

string arrivalAirport;

string departureDateTime;

string arrivalDateTime;

int availableSeats;

float pricePerSeat;

};

struct Passenger {

string name;

string passportInfo;

vector<string> bookedFlights; // List of flight numbers the passenger has booked

};

void displayFlightInfo(const Flight& flight);

void displayAvailableFlights(const vector<Flight>& flights, const string& origin, const string& destination, const string& startDate, const string& endDate);

bool bookSeats(vector<Flight>& flights, const string& flightNumber, int seats, Passenger& passenger);

bool cancelBooking(vector<Flight>& flights, const string& flightNumber, int seats, Passenger& passenger);

void displayBookedFlights(const Passenger& passenger);

void addFlight(vector<Flight>& flights, const Flight& newFlight);

} // namespace FlightManagement

#endif // FLIGHTMANAGEMENT\_H

IMPLEMENTATION:

#include "FlightManagement.h"

namespace FlightManagement {

void displayFlightInfo(const Flight& flight) {

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

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

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

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

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

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

cout << "Price per Seat: $" << flight.pricePerSeat << endl;

}

void displayAvailableFlights(const vector<Flight>& flights, const string& origin, const string& destination, const string& startDate, const string& endDate) {

for (const auto& flight : flights) {

if (flight.departureAirport == origin && flight.arrivalAirport == destination && flight.departureDateTime >= startDate && flight.departureDateTime <= endDate) {

displayFlightInfo(flight);

cout << "--------------------------" << endl;

}

}

}

bool bookSeats(vector<Flight>& flights, const string& flightNumber, int seats, Passenger& passenger) {

if (seats <= 0) {

cout << "Error: Number of seats to book must be positive." << endl;

return false;

}

for (auto& flight : flights) {

if (flight.flightNumber == flightNumber) {

if (flight.availableSeats >= seats) {

flight.availableSeats -= seats;

passenger.bookedFlights.push\_back(flightNumber);

cout << "Successfully booked " << seats << " seats on flight " << flightNumber << "." << endl;

return true;

} else {

cout << "Error: Not enough seats available." << endl;

return false;

}

}

}

cout << "Error: Flight not found." << endl;

return false;

}

bool cancelBooking(vector<Flight>& flights, const string& flightNumber, int seats, Passenger& passenger) {

if (seats <= 0) {

cout << "Error: Number of seats to cancel must be positive." << endl;

return false;

}

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 += seats;

passenger.bookedFlights.erase(it);

cout << "Successfully cancelled " << seats << " seats on flight " << flightNumber << "." << endl;

return true;

}

}

} else {

cout << "Error: Booking not found for the passenger." << endl;

}

return false;

}

void displayBookedFlights(const Passenger& passenger) {

cout << "Booked Flights for " << passenger.name << ":" << endl;

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

cout << "- " << flightNumber << endl;

}

}

void addFlight(vector<Flight>& flights, const Flight& newFlight) {

flights.push\_back(newFlight);

cout << "Flight " << newFlight.flightNumber << " added successfully." << endl;

}

} // namespace FlightManagement

MAIN:

#include "FlightManagement.h"

using namespace FlightManagement;

int main() {

vector<Flight> flights;

Passenger passenger1 = {"John Doe", "P12345678", {}};

// Adding flights

addFlight(flights, {"AA123", "JFK", "LAX", "2024-07-05 10:00", "2024-07-05 13:00", 150, 299.99});

addFlight(flights, {"BB456", "SFO", "ORD", "2024-07-06 14:00", "2024-07-06 18:00", 200, 199.99});

// Display available flights

cout << "Available flights from JFK to LAX between 2024-07-04 and 2024-07-06:" << endl;

displayAvailableFlights(flights, "JFK", "LAX", "2024-07-04", "2024-07-06");

// Book seats

bookSeats(flights, "AA123", 2, passenger1);

// Display booked flights for the passenger

displayBookedFlights(passenger1);

// Cancel booking

cancelBooking(flights, "AA123", 1, passenger1);

// Display booked flights for the passenger again

displayBookedFlights(passenger1);

return 0;

}

LAMBDA EXPRESSIONS:

#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

CAPTURE BYVALUE:

#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;

}

Int main(){

Lambda\_capture\_value();

Return 0;

}

REFERENCE CAPTURE:

#include <iostream>

using namespace std;

void lambda\_reference\_capture(){

int value = 1;

auto copy\_value = [&value]{

return value;

};

value=100;

auto stored\_value = copy\_value();

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

}

int main(){

lambda\_reference\_capture();

return 0;

}

CAPTURE BY BOTH:

#include <iostream>

using namespace std;

int main(){

int m=0;

int n=0;

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

cout<<m<<endl<<n<<endl;

}

USECASE:

#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;

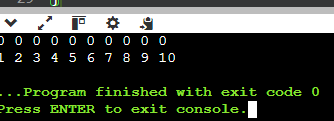
for\_each(vec.begin(),vec.end(),assign);

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

return 0;

}

OUTPUT:



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> // for std::copy\_if

#include <iterator> // for std::back\_inserter

// Function to filter even or odd numbers based on the is\_even flag

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

std::vector<int> filtered\_numbers; // Vector to store filtered numbers

// Lambda expression to check even or odd condition

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

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

};

// Use std::copy\_if to filter numbers based on the condition

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

return filtered\_numbers;

}

int main() {

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

// Filter even numbers

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

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

for (int num : even\_numbers) {

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

}

std::cout << std::endl;

// Filter odd numbers

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

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

for (int num : odd\_numbers) {

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

}

std::cout << std::endl;

return 0;

}

**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 <numeric>

// Template function to find the maximum value based on a comparison function

template <typename T>

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

// Use std::accumulate to find the maximum element

return std::accumulate(objects.begin(), objects.end(), objects[0],

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

return compare(max\_so\_far, current) ? current : max\_so\_far;

}

);

}

int main() {

// Example usage with integers

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

// Define comparison logic for integers

auto compare\_ints = [](const int& a, const int& b) {

return a < b; // Find the maximum value (b is greater)

};

// Find the maximum integer

int max\_number = find\_max(numbers, compare\_ints);

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

// Example usage with a custom struct

struct Person {

std::string name;

int age;

};

std::vector<Person> people = {

{"Alice", 30},

{"Bob", 25},

{"Charlie", 35},

{"Diana", 28}

};

// Define comparison logic for finding the oldest person

auto compare\_people = [](const Person& a, const Person& b) {

return a.age < b.age; // Find the person with the maximum age (b is older)

};

// Find the oldest person

Person oldest\_person = find\_max(people, compare\_people);

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

return 0;

}