

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 {                                // 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);

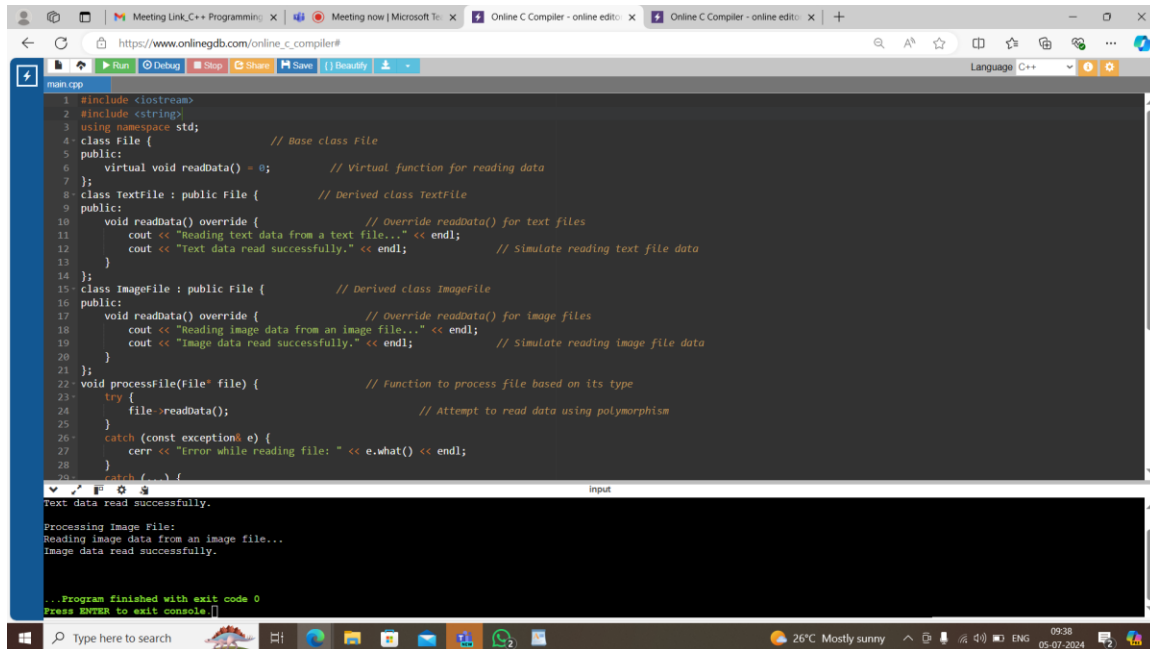
    cout << endl;

    return 0;

}

```

OUTPUT :-



```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class File { // Base class File
5 public:
6     virtual void readData() = 0; // Virtual function for reading data
7 };
8 class TextFile : public File { // Derived class TextFile
9 public:
10     void readData() override { // Override readData() for text files
11         cout << "Reading text data from a text file..." << endl;
12         cout << "Text data read successfully." << endl; // Simulate reading text file data
13     }
14 };
15 class ImageFile : public File { // Derived class ImageFile
16 public:
17     void readData() override { // Override readData() for image files
18         cout << "Reading image data from an image file..." << endl;
19         cout << "Image data read successfully." << endl; // Simulate reading image file data
20     }
21 };
22 void processFile(File* file) { // Function to process file based on its type
23     try {
24         file->readData(); // Attempt to read data using polymorphism
25     }
26     catch (const exception& e) {
27         cerr << "Error while reading file: " << e.what() << endl;
28     }
29     catch (...) {
30     }
31 }
32 int main() {
33     TextFile* t = new TextFile();
34     ImageFile* i = new ImageFile();
35     processFile(t);
36     processFile(i);
37     return 0;
38 }
```

Text data read successfully.

Processing Image File:

Reading image data from an image file...

Image data read successfully.

... Program finished with exit code 0

Press ENTER to exit console.

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>
```

```
class Chair {
```

```
public:
```

```
    virtual void sitOn() const = 0;
```

```
    virtual ~Chair() {} // virtual destructor for proper cleanup
```

```
};
```

```
class ModernChair : public Chair { // Concrete Products
```

```
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 {                                // Abstract Factory
public:
    virtual Chair* createChair() = 0;
    virtual ~FurnitureFactory() {}                    // virtual destructor for proper cleanup
};

class ModernFurnitureFactory : public FurnitureFactory { // Concrete Factories
public:
    Chair* createChair() override {
        return new ModernChair();
    }
};

class ClassicFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ClassicChair();
    }
}

```

```

};

void clientCode(FurnitureFactory* factory) {                                // Client code using
the factories

    Chair* chair = factory->createChair();

    chair->sitOn();

    delete chair;                                                         // Clean up allocated chair
}

int main() {

    FurnitureFactory* modernFactory = new ModernFurnitureFactory();

    clientCode(modernFactory);

    FurnitureFactory* classicFactory = new ClassicFurnitureFactory();

    clientCode(classicFactory);

    return 0;

}

```

OUTPUT :-

```

1 #include <iostream>
2 class Chair {
3 public:
4     virtual void sitOn() const = 0;
5     virtual ~Chair() {} // virtual destructor for proper cleanup
6 };
7 class ModernChair : public Chair { // concrete Products
8 public:
9     void sitOn() const override {
10         std::cout << "Sitting on a modern chair." << std::endl;
11     }
12 };
13 class ClassicChair : public Chair {
14 public:
15     void sitOn() const override {
16         std::cout << "Sitting on a classic chair." << std::endl;
17     }
18 };
19 class FurnitureFactory { // Abstract Factory
20 public:
21     virtual Chair* createChair() = 0;
22     virtual ~FurnitureFactory() {} // virtual destructor for proper cleanup
23 };
24 class ModernFurnitureFactory : public FurnitureFactory { // concrete Factories
25 public:
26     Chair* createChair() override {
27         return new ModernChair();
28     }
29 };

```

Sitting on a modern chair.
Sitting on a classic chair.
... Program finished with exit code 0
Press ENTER to exit console.

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:

```
#include <iostream>
```

```
#include <string>
```

```
struct Flight {                                // Define Flight structure
```

```
    std::string flightNumber;
```

```
    std::string departureAirport;
```

```
    std::string arrivalAirport;
```

```
    std::string departureDateTime;
```

```
    std::string arrivalDateTime;
```

```
    int availableSeats;
```

```
    float pricePerSeat;
```

```
};
```

```
int main() {
```

```
    Flight flight1;
```

```
    flight1.flightNumber = "ABC123";           // Assigning values to the  
members of the Flight structure
```

```

flight1.departureAirport = "JFK";

flight1.arrivalAirport = "LAX";

flight1.departureDateTime = "2024-07-05 10:00";

flight1.arrivalDateTime = "2024-07-05 13:00";

flight1.availableSeats = 150;

flight1.pricePerSeat = 250.0;

std::cout << "Flight Number: " << flight1.flightNumber << std::endl;           //
Displaying flight information

std::cout << "Departure Airport: " << flight1.departureAirport << std::endl;

std::cout << "Arrival Airport: " << flight1.arrivalAirport << std::endl;

std::cout << "Departure Date/Time: " << flight1.departureDateTime << std::endl;

std::cout << "Arrival Date/Time: " << flight1.arrivalDateTime << std::endl;

std::cout << "Available Seats: " << flight1.availableSeats << std::endl;

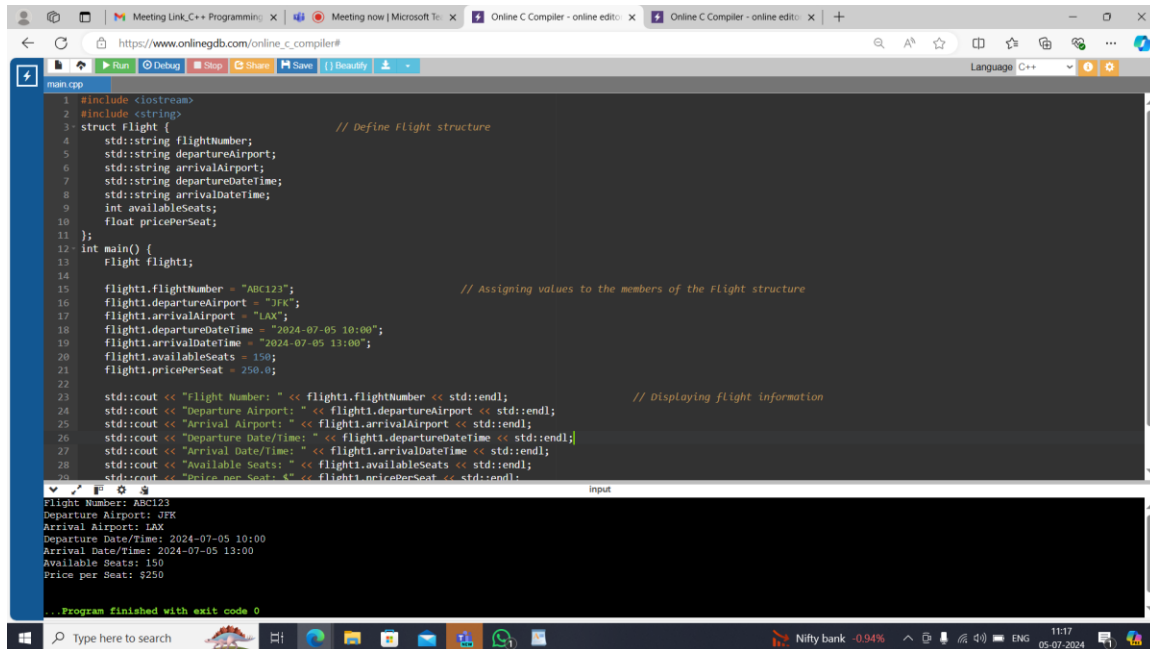
std::cout << "Price per Seat: $" << flight1.pricePerSeat << std::endl;

return 0;

}

```

OUTPUT :-



```
1 #include <iostream>
2 #include <string>
3 struct Flight {
4     std::string flightNumber;
5     std::string departureAirport;
6     std::string arrivalAirport;
7     std::string departureDateTime;
8     std::string arrivalDateTime;
9     int availableSeats;
10    float pricePerSeat;
11 };
12 int main() {
13     Flight flight1;
14
15     flight1.flightNumber = "ABC123";
16     flight1.departureAirport = "JFK";
17     flight1.arrivalAirport = "LAX";
18     flight1.departureDateTime = "2024-07-05 10:00";
19     flight1.arrivalDateTime = "2024-07-05 13:00";
20     flight1.availableSeats = 150;
21     flight1.pricePerSeat = 250.0;
22
23     std::cout << "Flight Number: " << flight1.flightNumber << std::endl;
24     std::cout << "Departure Airport: " << flight1.departureAirport << std::endl;
25     std::cout << "Arrival Airport: " << flight1.arrivalAirport << std::endl;
26     std::cout << "Departure Date/Time: " << flight1.departureDateTime << std::endl;
27     std::cout << "Arrival Date/Time: " << flight1.arrivalDateTime << std::endl;
28     std::cout << "Available Seats: " << flight1.availableSeats << std::endl;
29     std::cout << "Price per Seat: $" << flight1.pricePerSeat << std::endl;
30 }
31
32 ...Program finished with exit code 0
```

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 <string>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
struct Flight {                                // Define Flight structure
```

```
    std::string flightNumber;
```



```

    std::string departureAirport;

    std::string arrivalAirport;

    std::string departureDateTime;

    std::string arrivalDateTime;

    int availableSeats;

    float pricePerSeat;

};

struct Passenger {                                // Define Passenger structure

    std::string name;

    std::string passportInfo;

};

std::vector<Flight> flights;                        // Database to store flights

std::vector<std::pair<Flight*, int>> bookings;      // Vector of pairs to store flight and booked seats

void displayAvailableFlights(const std::string& origin, const std::string& destination, const std::string&
fromDate, const std::string& toDate) {

    std::cout << "Available flights from " << origin << " to " << destination << ":\n";

    for (const auto& flight : flights) {

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

            if (flight.departureDateTime >= fromDate && flight.departureDateTime <= toDate) {

                std::cout << "Flight Number: " << flight.flightNumber << ", Departure Date/Time: "
<< flight.departureDateTime << std::endl;

            }

        }

    }

}

void bookSeats(const std::string& flightNumber, int numSeats, const Passenger& passenger) {

```

```

Flight* chosenFlight = nullptr;

for (auto& flight : flights) {

    if (flight.flightNumber == flightNumber) {

        chosenFlight = &flight;

        break;

    }

}

if (chosenFlight) {

    if (numSeats > 0 && numSeats <= chosenFlight->availableSeats) {

        chosenFlight->availableSeats -= numSeats;

        bookings.push_back(std::make_pair(chosenFlight, numSeats));

        std::cout << "Booking successful for " << numSeats << " seats on flight " << flightNumber
<< " for passenger " << passenger.name << std::endl;

    } else {

        std::cout << "Error: Not enough seats available on flight " << flightNumber << std::endl;

    }

} else {

    std::cout << "Error: Flight " << flightNumber << " not found\n";

}

}

void cancelBooking(const std::string& flightNumber, int numSeats) {

    for (auto it = bookings.begin(); it != bookings.end(); ++it) {

        if (it->first->flightNumber == flightNumber && it->second == numSeats) {

            it->first->availableSeats += numSeats;

            bookings.erase(it);

            std::cout << "Booking canceled successfully for " << numSeats << " seats on flight " <<

```

```

flightNumber << std::endl;

        return;
    }
}

std::cout << "Error: Booking not found for flight " << flightNumber << " with " << numSeats << "
seats\n";
}

void displayBookedFlights(const std::string& passengerName) {

    std::cout << "Booked flights for passenger " << passengerName << ":\n";

    bool foundBookings = false;

    for (const auto& booking : bookings) {

        if (booking.first && booking.first->availableSeats < booking.second) {

            continue;

        }

        std::cout << "Flight Number: " << booking.first->flightNumber << ", Booked Seats: " <<
booking.second << std::endl;

        foundBookings = true;

    }

    if (!foundBookings) {

        std::cout << "No bookings found for passenger " << passengerName << std::endl;

    }

}

void addFlight(const Flight& newFlight) {                                     // Function to add new flights to
the system

    flights.push_back(newFlight);

    std::cout << "Flight " << newFlight.flightNumber << " added successfully\n";

}

```

```

int main() {

    Flight flight1 = {"ABC123", "JFK", "LAX", "2024-07-05 10:00", "2024-07-05 13:00", 150, 250.0};

    Flight flight2 = {"DEF456", "LAX", "JFK", "2024-07-06 12:00", "2024-07-06 15:00", 200, 300.0};

    flights.push_back(flight1);

    flights.push_back(flight2);

    Passenger passenger1 = {"John Doe", "AB123456"};

    Passenger passenger2 = {"Jane Smith", "CD987654"};

    displayAvailableFlights("JFK", "LAX", "2024-07-05", "2024-07-06");           // Example
function calls

    bookSeats("ABC123", 2, passenger1);

    bookSeats("DEF456", 3, passenger2);

    displayBookedFlights("John Doe");

    cancelBooking("ABC123", 2);

    displayBookedFlights("John Doe");

    Flight newFlight = {"GHI789", "LAX", "SFO", "2024-07-08 14:00", "2024-07-08 16:00", 180, 280.0};

    addFlight(newFlight);

    displayAvailableFlights("LAX", "SFO", "2024-07-08", "2024-07-09");

    return 0;

}

```

OUTPUT :-

```
Meeting Link_C++ Programmin... Meeting now | Microsoft 3... Online C Compiler - online edito... Online C Compiler - online edito...
https://www.onlinegdb.com/online_c_compiler#
main.cpp
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <algorithm>
5 struct Flight {
6     std::string flightNumber;
7     std::string departureAirport;
8     std::string arrivalAirport;
9     std::string departureDateTime;
10    std::string arrivalDateTime;
11    int availableSeats;
12    float pricePerSeat;
13};
14 struct Passenger {
15     std::string name;
16     std::string passportInfo;
17};
18 std::vector<Flight> flights; // Database to store flights
19 std::vector<std::pair<Flight*, int>> bookings; // Vector of pairs to store flight and booked seats
20 void displayAvailableFlights(const std::string& origin, const std::string& destination, const std::string& fromDate, const std::string& toDate) {
21     std::cout << "Available flights from " << origin << " to " << destination << ":\n";
22     for (const auto& flight : flights) {
23         if (flight.departureAirport == origin && flight.arrivalAirport == destination) {
24             if (flight.departureDateTime >= fromDate && flight.departureDateTime <= toDate) {
25
input
Available flights from JFK to LAX:
Flight Number: ABC123, Departure Date/Time: 2024-07-05 10:00
Booking successful for 2 seats on flight ABC123 for passenger John Doe
Booked flights for passenger John Doe:
Flight Number: ABC123, Booked Seats: 2
Flight Number: DEF456, Booked Seats: 3
Booking canceled successfully for 2 seats on flight ABC123
Booked flights for passenger John Doe:
Flight Number: DEF456, Booked Seats: 3
Flight GHI789 added successfully
Available flights from LAX to SFO:
Flight Number: GHI789, Departure Date/Time: 2024-07-08 14:00
```