SUNNY KUMAR
5 JULY TASK

```
/*****************************************************************************
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>

using namespace std;

class Chair {
public:
   virtual void sitOn() const = 0;
   virtual ~Chair() {}
};

class ModernChair : public Chair {
public:
   void sitOn() const override {
      cout<<"Sitting on a modern chair."<<endl;
   }
};

class ClassicChair : public Chair {
public:
   void sitOn() const override {
      cout<<"Sitting on a classic chair."<<endl;
   }
};

class FurnitureFactory {
public:
   virtual unique_ptr<Chair> createChair() const = 0;
```

```
    virtual ~FurnitureFactory() {}
};

class ModernFurnitureFactory : public FurnitureFactory {
public:
    unique_ptr<Chair> createChair() const override {
        return make_unique<ModernChair>();
    }
};

class ClassicFurnitureFactory : public FurnitureFactory {
public:
    unique_ptr<Chair> createChair() const override {
        return make_unique<ClassicChair>();
    }
};

void createAndUseChair(const FurnitureFactory& factory) {
    unique_ptr<Chair> chair = factory.createChair();
    chair->sitOn();
}

int main() {
    char choice;
    cout<<"Enter 'm' for Modern Furniture or 'c' for Classic Furniture: ";
    cin >> choice;

    unique_ptr<FurnitureFactory> factory;

    if (choice == 'm') {
        factory = make_unique<ModernFurnitureFactory>();
    } else if (choice == 'c') {
        factory = make_unique<ClassicFurnitureFactory>();
    } else {
        cout<<"Invalid choice."<<endl;
        return 1;
    }
    createAndUseChair(*factory);
    return 0;
}
```

Q.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 <memory>
#include <string>

using namespace std;

// Base class File
class File {
public:
    virtual void readData() const = 0; // Pure virtual function
    virtual ~File() {}
};

// Derived class TextFile
class TextFile : public File {
private:
    string fileName;

public:
    TextFile(const string& name) : fileName(name) {}

    void readData() const override {
        cout << "Reading text data from " << fileName << endl;
        // Specific code to read text file
        // Simulating file read with a simple message
        cout << "File content: Hello, this is a text file!" << endl;
    }
};

// Derived class ImageFile
class ImageFile : public File {
private:
    string fileName;

public:
    ImageFile(const string& name) : fileName(name) {}

    void readData() const override {
        cout << "Reading image data from " << fileName << endl;
        // Specific code to read image file
        // Simulating file read with a simple message
        cout << "File content: [Image data with width=1920 and height=1080]" << endl;
    }
};

// Function to process the file
void processFile(const unique_ptr<File>& file) {
    try {
        file->readData();
    } catch (const exception& e) {
        cout << "Error reading file: " << e.what() << endl;
    }
}

int main() {
    unique_ptr<File> textFile = make_unique<TextFile>("example.txt");
    unique_ptr<File> imageFile = make_unique<ImageFile>("example.jpg");

    cout << "Processing text file:" << endl;
    processFile(textFile);

    cout << "\nProcessing image file:" << endl;
    processFile(imageFile);

    return 0;
}
```

/**

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>
using namespace std;
struct Flight {            // Structure to represent a Flight
   string Flightno;
   string departureAirport;
   string arrivalAirport;
   string departureTime;
   string arrivalTime;
   int availableseat;
   float seatprice;
};

void displayAvailableFlights(const vector<Flight>& flights, const string& origin, const string&
destination) {
   bool found = false;
   for (const auto& flight : flights) {
      if (flight.departureAirport == origin && flight.arrivalAirport == destination) {
         cout<<"Flight Number is: "<<flight.Flightno<<", Available Seats are:
"<<flight.availableseat
            <<", SeatPrice is: "<<flight.seatprice<<endl;
         found = true;
      }
   }
   if (!found) {
      cout<<"No available flights from "<<origin<<" to "<<destination<<endl;
   }
}

void bookSeats(vector<Flight>& flights, const string& Flightno, int seats) {
   for (auto& flight : flights) {
        if (flight.Flightno == Flightno) {

            if (flight.availableseat >= seats) {
                flight.availableseat -= seats;
                cout<<"Booked "<<seats<<" seats on flight "<<Flightno<<endl;
            } else {
                cout<<"Error: Not enough available seats on flight "<<Flightno<<endl;
            }
            return;
        }
    }
    cout<<"Flight "<<Flightno<<" not found"<<endl;
}

void cancelBooking(vector<Flight>& flights, const string& Flightno, int seats) {
    for (auto& flight : flights) {
        if (flight.Flightno == Flightno) {

            flight.availableseat += seats;
            cout<<"Cancelled "<<seats<<" seats on flight "<<Flightno<<endl;
            return;
        }
    }
    cout<<"Flight "<<Flightno<<" not found"<<endl;
}

void addFlight(vector<Flight>& flights, const Flight& newFlight) {
    flights.push_back(newFlight);
    cout<<"Added flight "<<newFlight.Flightno<<endl;
}

int main() {
    vector<Flight> flights;

    flights.push_back({"FL001", "delhi", "mumbai", "2024-07-05 10:00", "2024-07-05 13:00", 150,
2990});
    flights.push_back({"FL002", "mumbai", "delhi", "2024-07-06 14:00", "2024-07-06 17:00", 100,
3499});
    flights.push_back({"FL003", "delhi", "odisha", "2024-07-07 15:00", "2024-07-09 19:00", 100,
3909});

    cout<<"Available flights from delhi to mumbai:"<<endl;
    displayAvailableFlights(flights, "delhi", "mumbai");
    bookSeats(flights, "FL001", 2);

    cancelBooking(flights, "FL001", 1);

    Flight newFlight = {"FL003", "patna", "goa", "2024-07-07 08:00", "2024-07-07 11:00", 200,
199.99};
    addFlight(flights, newFlight);

    cout<<"Available flights from delhi to mumbai after booking and cancellation:"<<endl;
    displayAvailableFlights(flights, "delhi", "mumbai");

    return 0;
}
```

```
/*****************************************************************************
lamda function in cpp

*****************************************************************************/
#include <iostream>

int multiply(int a, int b); // declaration

int main() {
    // normal function
    std::cout << multiply(4, 5) << std::endl; // 20

    // lambda expression (1)
    std::cout << [](int a, int b) { return a * b; }(4, 5) << std::endl; // 20

    // lambda expression (2)
    auto f = [](int a, int b) { return a * b; };
    std::cout << f(4, 5) << std::endl; // 20
}

int multiply(int a, int b) { // definition
    return a * b;
}
```

```
/*****************************************************************************
lambda function capture by value
*****************************************************************************/
#include <iostream>
using namespace std;
void lambda_value_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_value_capture();
   return 0;
}


/*****************************************************************************
lambda fn in cpp pass by reference

*****************************************************************************/

#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;
   // At this moment, stored_value == 100, value == 100.
   // Because copy_value stores reference
}

int main() {
   lambda_reference_capture();
   return 0;
}



/******************************************************************************
lambda fn in cpp both rfrence and value

*******************************************************************************/
#include <iostream>

using namespace std;


int main() {
    int m=0,n=0;
    [&,n](int a) mutable{m= ++n +a;}(4);
    cout<<m<<endl<<n<<endl;

    return 0;
}
```

/* 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 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>
using namespace std;
vector<int> filter_even_odds(const vector<int>& numbers, bool is_even) {
    vector<int> filtered_numbers;

    // Lambda function to filter numbers
    for_each(numbers.begin(), numbers.end(), [&](int num) {
        if ((num % 2 == 0) == is_even) {
            filtered_numbers.push_back(num);
        }
    });

    return filtered_numbers;
}

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

    // For even numbers
    vector<int> evens = filter_even_odds(numbers, true);
    cout << "Even numbers: ";
    for (int num : evens) {
        cout << num << " ";
    }
    cout << endl;

    // For odd numbers
    vector<int> odds = filter_even_odds(numbers, false);
    cout << "Odd numbers: ";
    for (int num : odds) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}

/*****************************************************************************
```

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 vector<T>& objects: The vector containing the objects (can be any type T).
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 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>
using namespace std;
template <typename T>
T find_max(const vector<T>& objects, function<bool(const T& a, const T& b)> compare) {
   return accumulate(objects.begin(), objects.end(), objects[0], [&](const T& max, const T& obj) {
      return compare(max, obj) ? obj : max;
   });
}
int main() {
   vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   auto compare_ints = [](const int& a, const int& b) {
      return a < b;
   };
   int max_int = find_max(numbers, compare_ints);
   cout << "Maximum integer: " << max_int << endl;
   struct Person {
      string name;
      int age;
   };
   vector<Person> people = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 35}};
   auto compare_people = [](const Person& a, const Person& b) {
      return a.age < b.age;
   };
   Person oldest_person = find_max(people, compare_people);
   cout << "Oldest person: " << oldest_person.name << " (" << oldest_person.age << ")" <<
endl;
   return 0;
}
```