SUNNY KUMAR
15 JULY TASK

Q. Create a class hierarchy (e.g., animals with different sounds) and manage object lifetimes and relationships using smart pointers. Include error handling to gracefully handle situations where resources might not be available.

STEPS:-

1. Define the Base Class
2. Define Derived Classes
3. Create a function to manage animals
4. Use Smart Pointers
5. Iterate over the collection and call makeSound
6. Main Function

Here is the code

```
/******************************************************************************
Create a class hierarchy (e.g., animals with different sounds) and manage object lifetimes and
relationships using smart pointers. Include error handling to gracefully handle situations where
resources might not be available.

*******************************************************************************/
#include <bits/stdc++.h>
using namespace std;

class Animal {
public:
   virtual ~Animal() = default;
   virtual void makeSound() const = 0;
};

class Dog : public Animal {
public:
   void makeSound() const override {
      cout << "Woof!" << endl;
   }
};

class Cat : public Animal {
```

```
public:
    void makeSound() const override {
        cout << "Meow!" << endl;
    }
};

class Bird : public Animal {
public:
    void makeSound() const override {
        cout << "Tweet!" << endl;
    }
};

void manageAnimals() {
    vector<shared_ptr<Animal>> animals;

    try {
        animals.push_back(make_shared<Dog>());
        animals.push_back(make_shared<Cat>());
        animals.push_back(make_shared<Bird>());

        for (const auto& animal : animals) {
            animal->makeSound();
        }
    } catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
    }
}

int main() {
    manageAnimals();
    return 0;
}
```

Q. Simulate rolling dice, flipping coins, or generating random temperatures within a range. Users can choose the type of distribution and potentially customize parameters.

STEPS:

1. DEFINE a random class that includes methods for rolling dice, flipping coins, and generating random temperatures.

2. Allow users to select the type of distribution and customize parameters
3. Implement Main Function provides a menu for users to choose the type of simulation and input parameters.

Here is the code

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

Simulate rolling dice, flipping coins, or generating random temperatures within a range. Users
can choose the type of distribution and potentially customize parameters.

*******************************************************************************/
#include <iostream>
#include <random>
#include <string>

using namespace std;

class RandomGenerator {
public:
    RandomGenerator() : gen(rd()) {}

    int rollDice(int sides) {
        uniform_int_distribution<> dis(1, sides);
        return dis(gen);
    }

    string flipCoin() {
        uniform_int_distribution<> dis(0, 1);
        return dis(gen) == 0 ? "Heads" : "Tails";
    }

    double generateTemperature(double minTemp, double maxTemp) {
        uniform_real_distribution<> dis(minTemp, maxTemp);
        return dis(gen);
    }

private:
    random_device rd;
    mt19937 gen;
};
```

```
void simulateDiceRoll(RandomGenerator& rng) {
    int sides;
    cout << "Enter the number of sides on the dice: ";
    cin >> sides;
    cout << "Rolling a " << sides << "-sided dice: " << rng.rollDice(sides) << endl;
}

void simulateCoinFlip(RandomGenerator& rng) {
    cout << "Flipping a coin: " << rng.flipCoin() << endl;
}

void simulateTemperatureGeneration(RandomGenerator& rng) {
    double minTemp, maxTemp;
    cout << "Enter the minimum temperature: ";
    cin >> minTemp;
    cout << "Enter the maximum temperature: ";
    cin >> maxTemp;
    cout << "Generated temperature: " << rng.generateTemperature(minTemp, maxTemp) << "
degrees" << endl;
}

int main() {
    RandomGenerator rng;
    int choice;

    do {
        cout << "Choose a simulation:" << endl;
        cout << "1. Roll Dice" << endl;
        cout << "2. Flip Coin" << endl;
        cout << "3. Generate Random Temperature" << endl;
        cout << "0. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                simulateDiceRoll(rng);
                break;
            case 2:
                simulateCoinFlip(rng);
                break;
            case 3:
                simulateTemperatureGeneration(rng);
            break;
        case 0:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
        }

        cout << endl;
    } while (choice != 0);

    return 0;
}
```

Q. Project 4: File I/O with Regular Expressions (Enhanced with Error Handling and Performance)

Concept: Employ C++11 file I/O streams (ifstream, ofstream) to read from and write to files.
Enhancements:
Error Handling: Implement robust error handling to gracefully deal with file opening failures, I/O errors, or invalid data formats. Consider using exceptions or custom error codes for better diagnostics.
Regular Expressions: Utilize the <regex> library to search for patterns within text files, allowing for more complex data extraction or manipulation.
Example: Create a program that reads a log file, searches for specific error messages using regular expressions, and writes the matching lines to a new file, providing informative error messages if issues arise during file access or processing.

Steps:

1. Define Custom Exception Classes
2. Implement Regular Expression Matching
3. File I/O Operations
4. Error Handling
5. Main Function

Here is the code

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

Project 4: File I/O with Regular Expressions (Enhanced with Error Handling and
Performance)
```

```
Concept: Employ C++11 file I/O streams (ifstream, ofstream) to read from and write to
files.
Enhancements:
Error Handling: Implement robust error handling to gracefully deal with file opening
failures, I/O errors, or invalid data formats. Consider using exceptions or custom error
codes for better diagnostics.
Regular Expressions: Utilize the <regex> library to search for patterns within text files,
allowing for more complex data extraction or manipulation.
Example: Create a program that reads a log file, searches for specific error messages
using regular expressions, and writes the matching lines to a new file, providing
informative error messages if issues arise during file access or processing.
****************************************************************************/
#include <iostream>
#include <fstream>
#include <regex>
#include <string>
#include <exception>
#include <vector>

using namespace std;
class FileOpenError : public exception {
public:
    FileOpenError(const string& filename)
        : message_("Failed to open file: " + filename) {}

    virtual const char* what() const noexcept override {
        return message_.c_str();
    }

private:
    string message_;
};

class InvalidDataFormatException : public exception {
public:
    InvalidDataFormatException(const string& line)
        : message_("Invalid data format in line: " + line) {}

    virtual const char* what() const noexcept override {
        return message_.c_str();
    }

private:
    string message_;
};

void searchLogFile(const string& inputFilename, const string& outputFilename, const
regex& pattern) {
    ifstream inputFile(inputFilename);
    if (!inputFile.is_open()) {
        throw FileOpenError(inputFilename);
    }

    ofstream outputFile(outputFilename);
    if (!outputFile.is_open()) {
        throw FileOpenError(outputFilename);
    }

    string line;
    while (getline(inputFile, line)) {
        if (regex_search(line, pattern)) {
            outputFile << line << endl;
        }
    }

    inputFile.close();
    outputFile.close();
}

int main() {
    string inputFilename = "logfile.txt";
    string outputFilename = "errorlog.txt";
    regex errorPattern("ERROR:.*");

    try {
        searchLogFile(inputFilename, outputFilename, errorPattern);
        cout << "Matching lines have been written to " << outputFilename << endl;
    } catch (const FileOpenError& e) {
        cerr << e.what() << endl;
    } catch (const InvalidDataFormatException& e) {
        cerr << e.what() << endl;
    } catch (const exception& e) {
        cerr << "Unexpected error: " << e.what() << endl;
    }

    return 0;
}
```

Q. Project 5: Modern C++ Design Patterns (Using Move Semantics and Lambdas)

Concept: Explore modern C++ design patterns like move semantics (rvalue references) and lambdas to write efficient and expressive code.
Enhancements:
Move Semantics: Optimize code by understanding how to efficiently move resources (like large objects) to avoid unnecessary copies.
Lambdas: Utilize lambda expressions to create concise and readable anonymous functions, particularly for short-lived logic or event handling.
Example: Create a container class that efficiently stores and moves large objects like images or scientific data. Implement custom iterators or member functions using lambdas to process elements in the container.
These enhanced projects will significantly improve your proficiency in C++11 by:

Steps:

1. Define a large class to represent large objects
2. Define a template class Container to store LargeObject instances.
3. Implement custom iterators for the container.
4. Use lambdas to process elements within the container
5. Demonstrate the use of the container class with large objects.
6. Show how move semantics optimize resource management.
7. Utilize lambdas for concise and readable processing of container elements.

Map in c++ stl

```
#include <iostream>
#include <map>
#include <iterator>
using namespace std;

int main()
{
   // empty map container
   map<int, int> gquiz1;
```

```
// insert elements in random order
gquiz1.insert(pair<int, int>(1, 40));
gquiz1.insert(pair<int, int>(2, 30));
gquiz1.insert(pair<int, int>(3, 60));
gquiz1.insert(pair<int, int>(4, 20));
gquiz1.insert(pair<int, int>(5, 50));
gquiz1.insert(pair<int, int>(6, 50));
gquiz1.insert(pair<int, int>(7, 10));

// printing map gquiz1
map<int, int>::iterator itr;
cout << "\nThe map gquiz1 is : \n";
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
   cout << '\t' << itr->first
        << '\t' << itr->second << '\n';
}
cout << endl;

// assigning the elements from gquiz1 to gquiz2
map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

// print all elements of the map gquiz2
cout << "\nThe map gquiz2 after"
     << " assign from gquiz1 is : \n";
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
   cout << '\t' << itr->first
        << '\t' << itr->second << '\n';
}
cout << endl;

// remove all elements up to
// element with key=3 in gquiz2
cout << "\ngquiz2 after removal of"
     << " elements less than key=3 : \n";
cout << "\tKEY\tELEMENT\n";
gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
   cout << '\t' << itr->first
        << '\t' << itr->second << '\n';
}

// remove all elements with key = 4
```

```
    int num;
    num = gquiz2.erase(4);
    cout << "\ngquiz2.erase(4) : ";
    cout << num << " removed \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
        cout << '\t' << itr->first
             << '\t' << itr->second << '\n';
    }

    cout << endl;

    // lower bound and upper bound for map gquiz1 key = 5
    cout << "gquiz1.lower_bound(5) : "
         << "\tKEY = ";
    cout << gquiz1.lower_bound(5)->first << '\t';
    cout << "\tELEMENT = "
         << gquiz1.lower_bound(5)->second << endl;
    cout << "gquiz1.upper_bound(5) : "
         << "\tKEY = ";
    cout << gquiz1.upper_bound(5)->first << '\t';
    cout << "\tELEMENT = "
         << gquiz1.upper_bound(5)->second << endl;

    return 0;
}
```

Q. Develop a C++ program that allows users to enter and store contact details (name, phone number, email) in a map. The program should provide options for adding new contacts, searching for existing contacts, and displaying all stored contacts.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

// Structure to hold contact details
struct Contact {
    string phoneNumber;
    string email;
};

// Function to add a new contact
void addContact(map<string, Contact>& contacts) {
    string name, phoneNumber, email;

    cout << "Enter name: ";
    getline(cin, name);

    cout << "Enter phone number: ";
    getline(cin, phoneNumber);

    cout << "Enter email address: ";
    getline(cin, email);

    // Create a Contact object
    Contact newContact;
    newContact.phoneNumber = phoneNumber;
    newContact.email = email;

    // Insert into the map
    contacts[name] = newContact;

    cout << "Contact added successfully!" << endl;
}

// Function to search for a contact
void searchContact(const map<string, Contact>& contacts) {
    string name;

    cout << "Enter name to search: ";
    getline(cin, name);

    // Search for the contact
    auto it = contacts.find(name);

    if (it != contacts.end()) {
        cout << "Contact found!" << endl;
        cout << "Name: " << it->first << endl;
        cout << "Phone number: " << it->second.phoneNumber << endl;
        cout << "Email: " << it->second.email << endl;
    } else {
        cout << "Contact not found." << endl;
    }
}

// Function to display all contacts
void displayContacts(const map<string, Contact>& contacts) {
    if (contacts.empty()) {
        cout << "No contacts to display." << endl;
        return;
    }

    cout << "Contacts:" << endl;
    for (const auto& pair : contacts) {
        cout << "Name: " << pair.first << endl;
        cout << "Phone number: " << pair.second.phoneNumber << endl;
        cout << "Email: " << pair.second.email << endl;
        cout << "------------------------" << endl;
    }
}

int main() {
    map<string, Contact> contacts;
    int choice;

    do {
        // Display menu
        cout << "\nContact Management System Menu:" << endl;
        cout << "1. Add a New Contact" << endl;
        cout << "2. Search for a Contact" << endl;
        cout << "3. Display All Contacts" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore();  // to consume the newline character

        switch (choice) {
            case 1:
                addContact(contacts);
                break;
            case 2:
                searchContact(contacts);
                break;
            case 3:
                displayContacts(contacts);
                break;
            case 4:
                cout << "Exiting program..." << endl;
                break;
            default:
                cout << "Invalid choice. Please try again." << endl;
        }
    } while (choice != 4);

    return 0;
}
```