1.) #include <iostream>

#include <memory>

#include <stdexcept>

// Base class Animal

class Animal {

public:

virtual ~Animal() = default;

virtual void makeSound() const = 0;

};

// Derived class Dog

class Dog : public Animal {

public:

void makeSound() const override {

std::cout << "Woof!" << std::endl;

}

};

// Derived class Cat

class Cat : public Animal {

public:

void makeSound() const override {

std::cout << "Meow!" << std::endl;

}

};

// Derived class Cow

class Cow : public Animal {

public:

void makeSound() const override {

std::cout << "Moo!" << std::endl;

}

};

// Function to create an animal based on type

std::shared\_ptr<Animal> createAnimal(const std::string& type) {

if (type == "Dog") {

return std::make\_shared<Dog>();

} else if (type == "Cat") {

return std::make\_shared<Cat>();

} else if (type == "Cow") {

return std::make\_shared<Cow>();

} else {

throw std::invalid\_argument("Unknown animal type");

}

}

// Main function demonstrating the usage

int main() {

try {

std::shared\_ptr<Animal> dog = createAnimal("Dog");

std::shared\_ptr<Animal> cat = createAnimal("Cat");

std::shared\_ptr<Animal> cow = createAnimal("Cow");

dog->makeSound();

cat->makeSound();

cow->makeSound();

} catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

// Using weak\_ptr to show relationships without creating circular dependencies

std::weak\_ptr<Animal> weakDog;

{

std::shared\_ptr<Animal> dog = createAnimal("Dog");

weakDog = dog;

}

if (auto sharedDog = weakDog.lock()) {

sharedDog->makeSound();

} else {

std::cout << "Dog has been destroyed" << std::endl;

}

return 0;

}

2.) #include <iostream>

#include <random>

#include <string>

#include <stdexcept>

// Function to roll a dice with a specified number of sides

void rollDice(int sides, int numRolls) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, sides);

std::cout << "Rolling a " << sides << "-sided dice " << numRolls << " times:\n";

for (int i = 0; i < numRolls; ++i) {

std::cout << dis(gen) << " ";

}

std::cout << std::endl;

}

// Function to flip a coin

void flipCoin(int numFlips) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, 1);

std::cout << "Flipping a coin " << numFlips << " times:\n";

for (int i = 0; i < numFlips; ++i) {

std::cout << (dis(gen) ? "Heads" : "Tails") << " ";

}

std::cout << std::endl;

}

// Function to generate random temperatures within a range

void generateTemperatures(int numTemps, double minTemp, double maxTemp, bool normalDist) {

std::random\_device rd;

std::mt19937 gen(rd());

if (normalDist) {

std::normal\_distribution<> dis((minTemp + maxTemp) / 2, (maxTemp - minTemp) / 6);

std::cout << "Generating " << numTemps << " temperatures (normal distribution) between "

<< minTemp << " and " << maxTemp << ":\n";

for (int i = 0; i < numTemps; ++i) {

double temp;

do {

temp = dis(gen);

} while (temp < minTemp || temp > maxTemp);

std::cout << temp << " ";

}

} else {

std::uniform\_real\_distribution<> dis(minTemp, maxTemp);

std::cout << "Generating " << numTemps << " temperatures (uniform distribution) between "

<< minTemp << " and " << maxTemp << ":\n";

for (int i = 0; i < numTemps; ++i) {

std::cout << dis(gen) << " ";

}

}

std::cout << std::endl;

}

// Function to display the menu and handle user input

void displayMenu() {

std::cout << "Random Simulation Menu:\n";

std::cout << "1. Roll Dice\n";

std::cout << "2. Flip Coin\n";

std::cout << "3. Generate Random Temperatures\n";

std::cout << "4. Exit\n";

std::cout << "Enter your choice: ";

}

int main() {

int choice;

while (true) {

displayMenu();

std::cin >> choice;

try {

switch (choice) {

case 1: {

int sides, numRolls;

std::cout << "Enter the number of sides on the dice: ";

std::cin >> sides;

std::cout << "Enter the number of rolls: ";

std::cin >> numRolls;

rollDice(sides, numRolls);

break;

}

case 2: {

int numFlips;

std::cout << "Enter the number of coin flips: ";

std::cin >> numFlips;

flipCoin(numFlips);

break;

}

case 3: {

int numTemps;

double minTemp, maxTemp;

bool normalDist;

std::cout << "Enter the number of temperatures to generate: ";

std::cin >> numTemps;

std::cout << "Enter the minimum temperature: ";

std::cin >> minTemp;

std::cout << "Enter the maximum temperature: ";

std::cin >> maxTemp;

std::cout << "Use normal distribution? (1 for yes, 0 for no): ";

std::cin >> normalDist;

generateTemperatures(numTemps, minTemp, maxTemp, normalDist);

break;

}

case 4:

std::cout << "Exiting...\n";

return 0;

default:

std::cerr << "Invalid choice. Please try again.\n";

}

} catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

}

return 0;

}

3.) #include <iostream>

#include <fstream>

#include <regex>

#include <string>

#include <stdexcept>

// Custom exception for file I/O errors

class FileIOException : public std::runtime\_error {

public:

explicit FileIOException(const std::string& message)

: std::runtime\_error(message) {}

};

// Function to read from a file, search for error messages, and write to another file

void processLogFile(const std::string& inputFilePath, const std::string& outputFilePath, const std::string& pattern) {

std::ifstream inputFile(inputFilePath);

if (!inputFile) {

throw FileIOException("Error opening input file: " + inputFilePath);

}

std::ofstream outputFile(outputFilePath);

if (!outputFile) {

throw FileIOException("Error opening output file: " + outputFilePath);

}

std::regex errorPattern(pattern);

std::string line;

while (std::getline(inputFile, line)) {

if (std::regex\_search(line, errorPattern)) {

outputFile << line << std::endl;

}

}

if (inputFile.bad()) {

throw FileIOException("Error reading input file: " + inputFilePath);

}

if (outputFile.bad()) {

throw FileIOException("Error writing to output file: " + outputFilePath);

}

}

// Main function to demonstrate the usage

int main() {

std::string inputFilePath;

std::string outputFilePath;

std::string pattern;

std::cout << "Enter the input log file path: ";

std::cin >> inputFilePath;

std::cout << "Enter the output file path: ";

std::cin >> outputFilePath;

std::cout << "Enter the regex pattern to search for: ";

std::cin.ignore(); // Clear the newline character from the input buffer

std::getline(std::cin, pattern);

try {

processLogFile(inputFilePath, outputFilePath, pattern);

std::cout << "Log file processed successfully. Matching lines written to " << outputFilePath << std::endl;

} catch (const FileIOException& e) {

std::cerr << "File I/O Error: " << e.what() << std::endl;

} catch (const std::regex\_error& e) {

std::cerr << "Regex Error: " << e.what() << std::endl;

} catch (const std::exception& e) {

std::cerr << "Unexpected Error: " << e.what() << std::endl;

}

return 0;

}

4.) #include <iostream>

#include <vector>

#include <memory>

#include <algorithm>

#include <stdexcept>

// Example large object class (Image)

class Image {

public:

Image(size\_t size) : size\_(size), data\_(new int[size]) {

std::cout << "Image of size " << size\_ << " created." << std::endl;

}

~Image() {

delete[] data\_;

std::cout << "Image of size " << size\_ << " destroyed." << std::endl;

}

// Move constructor

Image(Image&& other) noexcept : size\_(other.size\_), data\_(other.data\_) {

other.size\_ = 0;

other.data\_ = nullptr;

std::cout << "Image moved." << std::endl;

}

// Move assignment operator

Image& operator=(Image&& other) noexcept {

if (this != &other) {

delete[] data\_;

size\_ = other.size\_;

data\_ = other.data\_;

other.size\_ = 0;

other.data\_ = nullptr;

std::cout << "Image moved (assignment)." << std::endl;

}

return \*this;

}

// Deleted copy constructor and copy assignment operator

Image(const Image&) = delete;

Image& operator=(const Image&) = delete;

size\_t size() const { return size\_; }

private:

size\_t size\_;

int\* data\_;

};

// Container class for large objects

template<typename T>

class Container {

public:

void add(T&& item) {

items\_.emplace\_back(std::move(item));

}

template<typename Func>

void forEach(Func func) {

std::for\_each(items\_.begin(), items\_.end(), func);

}

private:

std::vector<T> items\_;

};

int main() {

try {

Container<Image> imageContainer;

imageContainer.add(Image(1000));

imageContainer.add(Image(2000));

imageContainer.add(Image(3000));

// Process images with a lambda function

imageContainer.forEach([](const Image& img) {

std::cout << "Processing image of size " << img.size() << std::endl;

});

} catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}

