15/7/24 Day16 Project

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. \*/

Creating a class hierarchy for animals with different sounds and managing object lifetimes using smart pointers involves designing classes that represent different animals, utilizing inheritance to capture common behaviors, and ensuring safe memory management with smart pointers like std::shared\_ptr and std::unique\_ptr

#include <iostream>

#include <memory> // for smart pointers

#include <vector>

// Base class Animal

class Animal {

public:

virtual ~Animal() = default; // Virtual destructor for polymorphic behavior

virtual void makeSound() const = 0; // Pure virtual function for making sound

};

// Derived classes for specific animals

class Dog : public Animal {

public:

void makeSound() const override {

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

}

};

class Cat : public Animal {

public:

void makeSound() const override {

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

}

};

class Cow : public Animal {

public:

void makeSound() const override {

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

}

};

// Function to create and manage animal objects using smart pointers

void createAndManageAnimals() {

// Using smart pointers for managing object lifetimes

std::shared\_ptr<Animal> dog = std::make\_shared<Dog>();

std::shared\_ptr<Animal> cat = std::make\_shared<Cat>();

std::shared\_ptr<Animal> cow = std::make\_shared<Cow>();

// Array of smart pointers

std::vector<std::shared\_ptr<Animal>> animals = {dog, cat, cow};

// Accessing and using objects

for (const auto& animal : animals) {

animal->makeSound();

}

}

int main() {

try {

createAndManageAnimals();

} catch (const std::bad\_alloc& e) {

std::cerr << "Memory allocation failed: " << e.what() << std::endl;

// Handle gracefully, perhaps retry or log the error

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

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

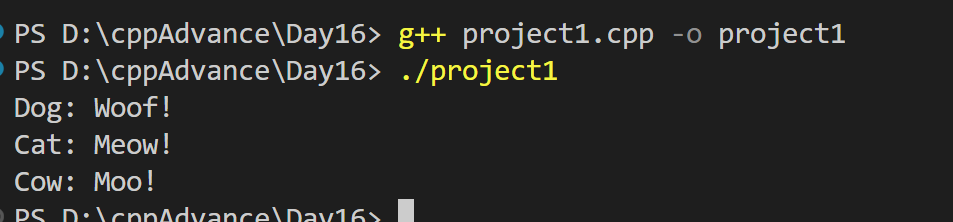
// Handle other exceptions

}

return 0;

}

Output



Ques2 Simulate rolling dice, flipping coins, or generating random temperatures within a range.Users can choose the type of distribution and potentially customize parameter

#include <iostream>

#include <random> // for random number generation

// Function to simulate rolling a dice with customizable sides

int rollDice(int sides) {

std::random\_device rd; // obtain a random number from hardware

std::mt19937 eng(rd()); // seed the generator

std::uniform\_int\_distribution<> distr(1, sides); // define the range

return distr(eng); // generate the random number

}

// Function to simulate flipping a coin (0 for tails, 1 for heads)

int flipCoin() {

std::random\_device rd; // obtain a random number from hardware

std::mt19937 eng(rd()); // seed the generator

std::uniform\_int\_distribution<> distr(0, 1); // define the range (0 or 1)

return distr(eng); // generate the random number

}

// Function to generate a random temperature within a specified range

double generateRandomTemperature(double minTemp, double maxTemp) {

std::random\_device rd; // obtain a random number from hardware

std::mt19937 eng(rd()); // seed the generator

std::uniform\_real\_distribution<> distr(minTemp, maxTemp); // define the range

return distr(eng); // generate the random number

}

int main() {

int choice;

std::cout << "Choose an option:" << std::endl;

std::cout << "1. Roll a dice" << std::endl;

std::cout << "2. Flip a coin" << std::endl;

std::cout << "3. Generate a random temperature" << std::endl;

std::cin >> choice;

switch (choice) {

case 1: {

int sides;

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

std::cin >> sides;

int result = rollDice(sides);

std::cout << "Result of rolling a " << sides << "-sided dice: " << result << std::endl;

break;

}

case 2: {

int result = flipCoin();

std::cout << "Result of flipping a coin: " << (result == 0 ? "Tails" : "Heads") << std::endl;

break;

}

case 3: {

double minTemp, maxTemp;

std::cout << "Enter the minimum and maximum temperatures: ";

std::cin >> minTemp >> maxTemp;

double temperature = generateRandomTemperature(minTemp, maxTemp);

std::cout << "Random temperature generated: " << temperature << " degrees Celsius" << std::endl;

break;

}

default:

std::cout << "Invalid choice. Please choose from 1 to 3." << std::endl;

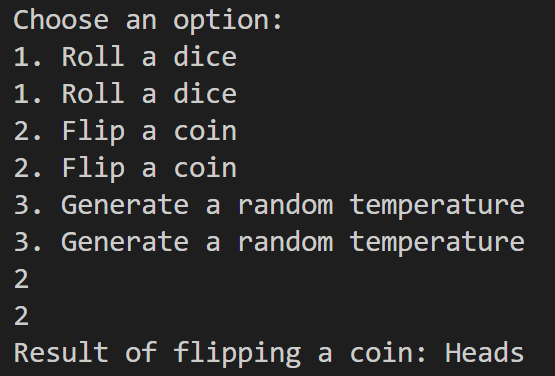
break;

}

return 0;

}

Output



Ques3 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: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.ample: 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 accessor processing.

#include <iostream>

#include <fstream>

#include <regex>

#include <stdexcept>

// Function to read log file, search for errors using regex, and write to output file

void processLogFile(const std::string& inputFile, const std::string& outputFile) {

std::ifstream inFile(inputFile);

if (!inFile) {

throw std::runtime\_error("Error opening input file: " + inputFile);

}

std::ofstream outFile(outputFile);

if (!outFile) {

throw std::runtime\_error("Error opening output file: " + outputFile);

}

// Regular expression to match error messages (example pattern)

std::regex errorRegex(R"(error|exception)", std::regex\_constants::icase); // Match "error" or "exception", case insensitive

std::string line;

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

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

outFile << line << std::endl; // Write matching line to output file

}

}

// Close files

inFile.close();

outFile.close();

}

int main() {

std::string inputFile = "input.log";

std::string outputFile = "errors.log";

try {

processLogFile(inputFile, outputFile);

std::cout << "Error messages extracted from " << inputFile << " and saved to " << outputFile << std::endl;

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

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

}

Ques4

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:

Emphasizing robust error handling for real-world application reliability.

Leveraging regular expressions for powerful text manipulation.

Optimizing code with move semantics and lambdas.

Applying modern design patterns for well-structured and maintainable code.

#include <iostream>

#include <vector>

#include <memory> // for std::unique\_ptr

#include <algorithm> // for std::for\_each

#include <functional> // for std::function

// Example class representing a large object (e.g., image)

class Image {

private:

std::string name;

// Simulating a large data buffer

std::unique\_ptr<char[]> data;

size\_t size;

public:

Image(const std::string& n, size\_t s) : name(n), size(s) {

data = std::make\_unique<char[]>(size);

std::cout << "Creating image " << name << " with size " << size << std::endl;

}

// Move constructor to efficiently transfer ownership of resources

Image(Image & other) noexcept : name(std::move(other.name)), data(std::move(other.data)), size(other.size) {

other.size = 0;

std::cout << "Moving image " << name << std::endl;

}

// Move assignment operator

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

if (this != &other) {

name = std::move(other.name);

data = std::move(other.data);

size = other.size;

other.size = 0;

std::cout << "Moving image " << name << std::endl;

}

return \*this;

}

~Image() {

if (data) {

std::cout << "Destroying image " << name << std::endl;

}

}

// Example member function for image processing using lambdas

void process(std::function<void(char\*, size\_t)> processor) {

if (data) {

processor(data.get(), size);

}

}

};

// Container class for images

class ImageContainer {

private:

std::vector<Image> images;

public:

void addImage(const std::string& name, size\_t size) {

images.emplace\_back(name, size);

}

// Example function using lambda to process all images in the container

void processAll(std::function<void(char\*, size\_t)> processor) {

std::for\_each(images.begin(), images.end(), [&](Image& img) {

img.process(processor);

});

}

};

int main() {

ImageContainer container;

// Adding images to the container

container.addImage("Image1", 1000);

container.addImage("Image2", 1500);

container.addImage("Image3", 800);

// Processing images using a lambda function

container.processAll([](char\* data, size\_t size) {

// Example processing: just print the first byte

if (size > 0) {

std::cout << "Processing image data. First byte: " << static\_cast<int>(data[0]) << std::endl;

}

});

return 0;

}

Output

