**1.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 <iostream>

#include <memory>

#include <string>

#include <stdexcept>

// Base class Animal

class Animal {

public:

virtual ~Animal() {

std::cout << "Animal destroyed" << std::endl;

}

virtual void makeSound() const = 0;

};

// Derived class Dog

class Dog : public Animal {

public:

void makeSound() const override {

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

}

~Dog() {

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

}

};

// Derived class Cat

class Cat : public Animal {

public:

void makeSound() const override {

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

}

~Cat() {

std::cout << "Cat destroyed" << std::endl;

}

};

// Derived class Cow

class Cow : public Animal {

public:

void makeSound() const override {

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

}

~Cow() {

std::cout << "Cow destroyed" << 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");

}

}

int main() {

try {

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

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

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

myDog->makeSound();

myCat->makeSound();

myCow->makeSound();

// Trying to create an unknown animal type to demonstrate error handling

std::shared\_ptr<Animal> unknownAnimal = createAnimal("Dinosaur");

} catch (const std::invalid\_argument& e) {

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

}

return 0;

}

**Explanation**

1. **Base Class Animal**: An abstract base class with a virtual destructor and a pure virtual function makeSound().
2. **Derived Classes Dog, Cat, Cow**: These classes override the makeSound() method to print their respective sounds. Each derived class also has a destructor to show when an object is destroyed.
3. **createAnimal Function**: This function takes a string representing the type of animal and returns a std::shared\_ptr to the corresponding animal object. It throws an std::invalid\_argument exception if the type is unknown.
4. **Main Function**:
   * Creates instances of Dog, Cat, and Cow using createAnimal.
   * Calls makeSound on each animal.
   * Demonstrates error handling by attempting to create an unknown animal type, which triggers an exception that is caught and handled gracefully.

This example shows how to manage object lifetimes with smart pointers and handle errors when creating objects based on a type string.

**2 . 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>

// Function to roll a dice

int rollDice(int sides) {

std::random\_device rd;

std::mt19937 gen(rd());

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

return dist(gen);

}

// Function to flip a coin

std::string flipCoin() {

std::random\_device rd;

std::mt19937 gen(rd());

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

return dist(gen) == 0 ? "Heads" : "Tails";

}

// Function to generate random temperature

double generateTemperature(double minTemp, double maxTemp) {

std::random\_device rd;

std::mt19937 gen(rd());

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

return dist(gen);

}

int main() {

int choice;

std::cout << "Choose an option:\n1. Roll Dice\n2. Flip Coin\n3. Generate Temperature\n";

std::cin >> choice;

if (choice == 1) {

int sides;

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

std::cin >> sides;

int result = rollDice(sides);

std::cout << "You rolled a " << result << std::endl;

} else if (choice == 2) {

std::string result = flipCoin();

std::cout << "The coin landed on " << result << std::endl;

} else if (choice == 3) {

double minTemp, maxTemp;

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

std::cin >> minTemp;

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

std::cin >> maxTemp;

double result = generateTemperature(minTemp, maxTemp);

std::cout << "The generated temperature is " << result << " degrees" << std::endl;

} else {

std::cerr << "Invalid choice" << std::endl;

}

return 0;

}

**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 <stdexcept>

// Custom exception for file errors

class FileError : public std::runtime\_error {

public:

FileError(const std::string& message) : std::runtime\_error(message) {}

};

// Function to search for error messages in a log file

void searchErrors(const std::string& inputFilePath, const std::string& outputFilePath, const std::regex& errorPattern) {

std::ifstream inputFile(inputFilePath);

if (!inputFile) {

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

}

std::ofstream outputFile(outputFilePath);

if (!outputFile) {

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

}

std::string line;

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

try {

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

outputFile << line << std::endl;

}

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

throw std::runtime\_error("Regex error: " + std::string(e.what()));

}

}

if (inputFile.bad()) {

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

}

if (outputFile.bad()) {

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

}

inputFile.close();

outputFile.close();

}

int main() {

try {

std::string inputFilePath = "log.txt";

std::string outputFilePath = "error\_log.txt";

std::regex errorPattern(R"(ERROR:.\*)");

searchErrors(inputFilePath, outputFilePath, errorPattern);

std::cout << "Error messages have been written to " << outputFilePath << std::endl;

} catch (const FileError& e) {

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

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

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

} catch (...) {

std::cerr << "An unexpected error occurred." << std::endl;

}

return 0;

}

**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:**

**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 <string>

#include <algorithm>

#include <memory>

#include <functional> // Include this header for std::function

// A simple class representing a large object, like an image or scientific data

class LargeObject {

public:

LargeObject(size\_t size, const std::string& name) : size\_(size), name\_(name) {

data\_ = new int[size\_];

std::cout << "Created LargeObject: " << name\_ << " of size " << size\_ << std::endl;

}

// Move constructor

LargeObject(LargeObject&& other) noexcept : size\_(other.size\_), name\_(std::move(other.name\_)), data\_(other.data\_) {

other.data\_ = nullptr;

other.size\_ = 0;

std::cout << "Moved LargeObject: " << name\_ << std::endl;

}

// Move assignment operator

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

if (this != &other) {

delete[] data\_;

size\_ = other.size\_;

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

data\_ = other.data\_;

other.data\_ = nullptr;

other.size\_ = 0;

std::cout << "Move-assigned LargeObject: " << name\_ << std::endl;

}

return \*this;

}

// Destructor

~LargeObject() {

delete[] data\_;

std::cout << "Destroyed LargeObject: " << name\_ << std::endl;

}

const std::string& getName() const { return name\_; }

private:

size\_t size\_;

std::string name\_;

int\* data\_;

// Disable copy constructor and copy assignment operator

LargeObject(const LargeObject&) = delete;

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

};

// Container class for managing LargeObject instances

class Container {

public:

// Add an object using move semantics

void addObject(LargeObject&& obj) {

objects\_.emplace\_back(std::move(obj));

}

// Process each object using a lambda function

void processObjects(const std::function<void(const LargeObject&)>& func) const {

for (const auto& obj : objects\_) {

func(obj);

}

}

// Custom iterator to iterate over LargeObject instances

std::vector<LargeObject>::const\_iterator begin() const { return objects\_.begin(); }

std::vector<LargeObject>::const\_iterator end() const { return objects\_.end(); }

private:

std::vector<LargeObject> objects\_;

};

int main() {

Container container;

// Adding objects to the container

container.addObject(LargeObject(1000, "Object1"));

container.addObject(LargeObject(2000, "Object2"));

container.addObject(LargeObject(3000, "Object3"));

// Process objects using a lambda function

container.processObjects([](const LargeObject& obj) {

std::cout << "Processing " << obj.getName() << std::endl;

});

// Using custom iterator with a lambda function to find an object

auto it = std::find\_if(container.begin(), container.end(), [](const LargeObject& obj) {

return obj.getName() == "Object2";

});

if (it != container.end()) {

std::cout << "Found " << it->getName() << std::endl;

} else {

std::cout << "Object not found" << std::endl;

}

return 0;

}