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 <vector>

#include <exception>

using namespace std;

class Animal {

public:

virtual void makeSound() const = 0;

virtual ~Animal() = default;

};

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 Cow : public Animal {

public:

void makeSound() const override {

cout << "Moo!" << endl;

}

};

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

if (type == "dog") {

return make\_shared<Dog>();

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

return make\_shared<Cat>();

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

return make\_shared<Cow>();

} else {

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

}

}

int main() {

vector<shared\_ptr<Animal>> zoo;

try {

zoo.push\_back(createAnimal("dog"));

zoo.push\_back(createAnimal("cat"));

zoo.push\_back(createAnimal("cow"));

zoo.push\_back(createAnimal("unknown"));

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

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

}

for (const auto& animal : zoo) {

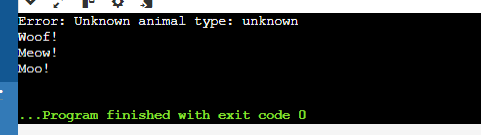
animal->makeSound();

}

return 0;

}

Output:



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 <cstdlib>

#include <ctime>

using namespace std;

int rollDice(int sides = 6) {

return (rand() % sides) + 1;

}

string flipCoin() {

return (rand() % 2) == 0 ? "Heads" : "Tails";

}

double generateTemperature(double minTemp = -30.0, double maxTemp = 50.0) {

double fraction = (double)rand() / RAND\_MAX;

return minTemp + fraction \* (maxTemp - minTemp);

}

int main() {

srand(time(0));

cout << "Choose the type of simulation:\n";

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

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

cout << "3. Generate Random Temperature\n";

cout << "Enter your choice (1-3): ";

int choice;

cin >> choice;

switch (choice) {

case 1: {

int sides;

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

cin >> sides;

cout << "You rolled a " << rollDice(sides) << "\n";

break;

}

case 2: {

cout << "You flipped a " << flipCoin() << "\n";

break;

}

case 3: {

double minTemp, maxTemp;

cout << "Enter the minimum temperature: ";

cin >> minTemp;

cout << "Enter the maximum temperature: ";

cin >> maxTemp;

cout << "Generated random temperature: " << generateTemperature(minTemp, maxTemp) << "°C\n";

break;

}

default:

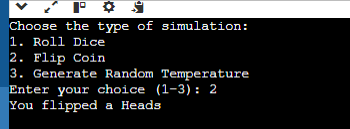
cout << "Invalid choice. Please enter a number between 1 and 3.\n";

}

return 0;

}

Output:



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>

using namespace std;

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

std::ifstream inputFile(inputFilePath);

if (!inputFile.is\_open()) {

cerr << "Error: Could not open input file: " << inputFilePath << endl;

return;

}

ofstream outputFile(outputFilePath);

if (!outputFile.is\_open()) {

cerr << "Error: Could not open output file: " << outputFilePath << endl;

inputFile.close();

return;

}

regex regexPattern(pattern);

string line;

try {

while (getline(inputFile, line)) {

if (regex\_search(line, regexPattern)) {

outputFile << line << endl;

}

}

} catch (const regex\_error& e) {

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

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

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

}

inputFile.close();

outputFile.close();

}

int main() {

string inputFilePath = "log.txt";

string outputFilePath = "errors.txt";

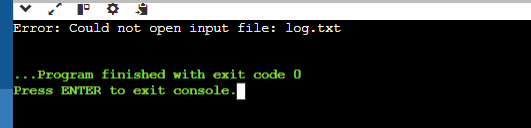
string errorPattern = R"(ERROR)";

processLogFile(inputFilePath, outputFilePath, errorPattern);

return 0;

}

Output:



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 <algorithm>

#include <functional>

using namespace std;

class LargeObject {

public:

LargeObject(int id) : id\_(id) {

cout << "Constructing LargeObject " << id\_ << endl;

}

LargeObject(LargeObject&& other) noexcept : id\_(other.id\_) {

cout << "Moving LargeObject " << id\_ << endl;

other.id\_ = -1;

}

~LargeObject() {

cout << "Destroying LargeObject " << id\_ << endl;

}

void process() {

cout << "Processing LargeObject " << id\_ << endl;

}

private:

int id\_;

};

class LargeObjectContainer {

public:

void addLargeObject(LargeObject&& obj) {

objects\_.push\_back(move(obj));

}

void forEach( function<void(LargeObject&)> func) {

for (auto& obj : objects\_) {

func(obj);

}

}

private:

vector<LargeObject> objects\_;

};

int main() {

LargeObjectContainer container;

container.addLargeObject(LargeObject(1));

container.addLargeObject(LargeObject(2));

container.addLargeObject(LargeObject(3));

container.forEach([](LargeObject& obj) {

obj.process();

});

return 0;

}

Output:

