**Date:15/07/2024**

**Day\_16\_Assignments**

**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 <vector>

#include <stdexcept>

vector is for storing a collection of animals.

stdexcept is for exception handling.

**Base Class Animal:**

class Animal {

public:

virtual void makeSound() const = 0;

virtual ~Animal() = default;

};

makeSound() is a pure virtual function, making Animal an abstract class.

The destructor is virtual to ensure proper cleanup of derived class objects.

**Derived Classes Dog and Cat:**

class Dog : public Animal {

public:

void makeSound() const override {

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

}

};

class Cat : public Animal {

public:

void makeSound() const override {

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

}

};

These classes override makeSound() to provide specific implementations.

**Function createAnimal:**

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 {

throw std::runtime\_error("Unknown animal type");

}

}

This function returns a smart pointer to a new Animal object based on the type.

Throws an exception if the type is unknown.

**Main Function:**

int main() {

try {

std::vector<std::shared\_ptr<Animal>> animals;

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

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

animals.push\_back(createAnimal("lion")); // This will throw an exception

for (const auto& animal : animals) {

animal->makeSound();

}

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

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

}

return 0;

}

Creates a vector of smart pointers to Animal objects.Adds animals to the vector.Attempts to add an unknown animal type (lion), which throws an exception.Catches and handles the exception, displaying an error message.

Output:



**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>

* random is for random number generation.
* string is for handling string input.

**Function rollDice:**

void rollDice(int numRolls) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 6);

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

std::cout << "Roll " << (i + 1) << ": " << dis(gen) << std::endl;

}

}

* Uses std::random\_device to seed the random number generator.
* Uses std::mt19937 as the random number generator.
* Uses std::uniform\_int\_distribution to generate numbers between 1 and 6.
* Rolls the dice numRolls times and prints the result.

**Function flipCoin:**

void flipCoin(int numFlips) {

std::random\_device rd;

std::mt19937 gen(rd());

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

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

std::cout << "Flip " << (i + 1) << ": " << (dis(gen) == 0 ? "Heads" : "Tails") << std::endl;

}

}

* Similar to rollDice, but generates 0 or 1 for heads or tails.

**Function generateTemperatures:**

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

std::random\_device rd;

std::mt19937 gen(rd());

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

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

std::cout << "Temperature " << (i + 1) << ": " << dis(gen) << "°C" << std::endl;

}

}

* Uses std::uniform\_real\_distribution to generate random temperatures between minTemp and maxTemp.

**Main Function:**

int main() {

std::string choice;

int num;

std::cout << "Choose simulation type (dice, coin, temperature): ";

std::cin >> choice;

if (choice == "dice") {

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

std::cin >> num;

rollDice(num);

} else if (choice == "coin") {

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

std::cin >> num;

flipCoin(num);

} else if (choice == "temperature") {

double minTemp, maxTemp;

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

std::cin >> num;

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

std::cin >> minTemp;

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

std::cin >> maxTemp;

generateTemperatures(num, minTemp, maxTemp);

} else {

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

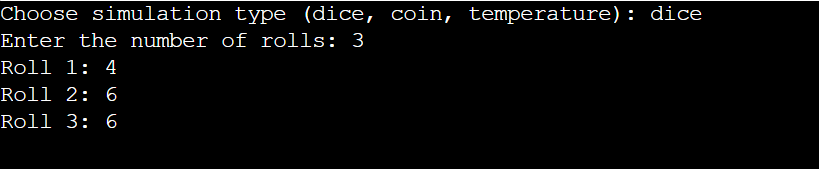
}

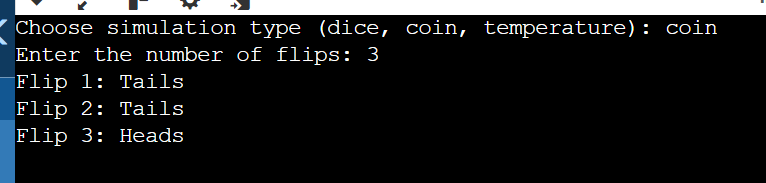
return 0;

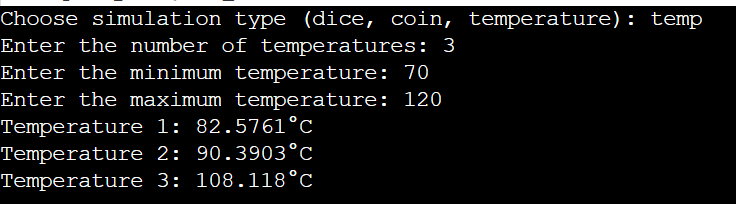
}

* Prompts the user to choose a simulation type.
* Based on the choice, prompts for additional parameters and calls the appropriate function.
* Handles invalid choices by displaying an error message.

Output:







**3. 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 <regex>: For regular expressions.
* #include <string>: For string handling.
* #include <stdexcept>: For exception handling.

**Function searchLogFile:**

* Opens the input file and checks if it was successfully opened. If not, throws an exception with an error message.
* Opens the output file and checks if it was successfully opened. If not, throws an exception with an error message.
* Uses std::regex to create a regular expression object from the user-provided pattern.
* Reads the input file line by line. For each line, it uses std::regex\_search to check if the line matches the pattern. If it does, the line is written to the output file.
* Closes both the input and output files.

**Main Function:**

* Prompts the user for the input file name, output file name, and the regular expression pattern.
* Calls searchLogFile inside a try-catch block to handle any exceptions that might occur.
* If an exception is thrown, it catches the exception and prints an error message. If no exceptions occur, it prints a success message.

#include <iostream>

#include <fstream>

#include <regex>

#include <string>

#include <stdexcept>

// Function to search log file for specific patterns and write matches to output file

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

// Open input file

std::ifstream inFile(inputFile);

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

throw std::runtime\_error("Failed to open input file: " + inputFile);

}

// Open output file

std::ofstream outFile(outputFile);

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

throw std::runtime\_error("Failed to open output file: " + outputFile);

}

// Create regular expression object from pattern

std::regex re(pattern);

std::string line;

// Read input file line by line

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

// Search for pattern in line

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

// Write matching lines to output file

outFile << line << std::endl;

}

}

// Close input and output files

inFile.close();

outFile.close();

}

int main() {

// Variables to store user input

std::string inputFile, outputFile, pattern;

// Prompt user for input file name

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

std::cin >> inputFile;

// Prompt user for output file name

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

std::cin >> outputFile;

// Prompt user for regular expression pattern to search for

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

std::cin >> pattern;

try {

// Call function to search log file

searchLogFile(inputFile, outputFile, pattern);

std::cout << "Processing completed successfully." << std::endl;

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

// Catch and display any exceptions

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

}

return 0;

}

4. 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>

* iostream for input-output operations.
* vector for dynamic array.
* algorithm for standard algorithms (not used here but useful for more advanced operations).
* functional for std::function.

**Container Class LargeObjectContainer:**

template<typename T>

class LargeObjectContainer {

public:

void add(T obj) {

objects.push\_back(std::move(obj));

}

void processElements(const std::function<void(T&)>& func) {

for (auto& obj : objects) {

func(obj);

}

}

private:

std::vector<T> objects;

};

* add method uses std::move to efficiently move objects into the container.
* processElements method takes a lambda function and applies it to each element in the container.

**Large Object Class LargeObject:**

class LargeObject {

public:

LargeObject(int size) : data(new int[size]), size(size) {

std::cout << "LargeObject created with size " << size << std::endl;

}

LargeObject(LargeObject&& other) noexcept

: data(other.data), size(other.size) {

other.data = nullptr;

other.size = 0;

std::cout << "LargeObject moved" << std::endl;

}

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

if (this != &other) {

delete[] data;

data = other.data;

size = other.size;

other.data = nullptr;

other.size = 0;

std::cout << "LargeObject move-assigned" << std::endl;

}

return \*this;

}

~LargeObject() {

delete[] data;

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

}

void process() {

std::cout << "Processing LargeObject of size " << size << std::endl;

}

private:

int\* data;

int size;

};

* Demonstrates the creation, moving, and destruction of large objects.
* Implements move constructor and move assignment operator to enable efficient resource management.

**Main Function:**

int main() {

LargeObjectContainer<LargeObject> container;

container.add(LargeObject(1000));

container.add(LargeObject(2000));

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

obj.process();

});

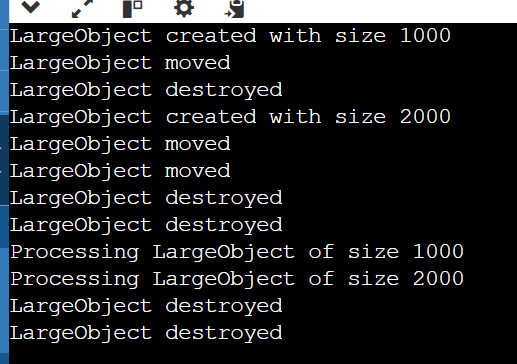
return 0;

}

Creates a LargeObjectContainer and adds large objects to it.

Uses a lambda function to process each large object in the container.

Output:



**5. 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;

// Struct to hold contact details

struct Contact {

std::string phoneNumber;

std::string email;

};

// Function to add a contact

void addContact(std::map<std::string, Contact>& contacts) {

std::string name, phoneNumber, email;

std::cout << "Enter name: ";

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

std::cout << "Enter phone number: ";

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

std::cout << "Enter email: ";

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

contacts[name] = {phoneNumber, email};

std::cout << "Contact added successfully." << std::endl;

}

// Function to search for a contact

void searchContact(const std::map<std::string, Contact>& contacts) {

std::string name;

std::cout << "Enter name to search: ";

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

auto it = contacts.find(name);

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

std::cout << "Name: " << it->first << std::endl;

std::cout << "Phone Number: " << it->second.phoneNumber << std::endl;

std::cout << "Email: " << it->second.email << std::endl;

} else {

std::cout << "Contact not found." << std::endl;

}

}

// Function to display all contacts

void displayContacts(const std::map<std::string, Contact>& contacts) {

if (contacts.empty()) {

std::cout << "No contacts available." << std::endl;

return;

}

for (const auto& [name, contact] : contacts) {

std::cout << "Name: " << name << std::endl;

cout << "Phone Number: " << contact.phoneNumber << std::endl;

std::cout << "Email: " << contact.email << std::endl;

std::cout << "-----------------------------" << std::endl;

}

}

// Main function

int main() {

std::map<std::string, Contact> contacts;

int choice;

while (true) {

cout << "Contact Management System" << std::endl;

std::cout << "1. Add New Contact" << std::endl;

std::cout << "2. Search Contact" << std::endl;

std::cout << "3. Display All Contacts" << std::endl;

std::cout << "4. Exit" << std::endl;

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

std::cin >> choice;

std::cin.ignore(); // Ignore newline character after choice input

switch (choice) {

case 1:

addContact(contacts);

break;

case 2:

searchContact(contacts);

break;

case 3:

displayContacts(contacts);

break;

case 4:

std::cout << "Exiting program." << std::endl;

return 0;

default:

std::cout << "Invalid choice. Please try again." << std::endl;

}

}

}

Output:

