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 <string>

using namespace std;

class Animal{ //base class

public:

virtual void makesound() const =0; //pure virtual function

virtual ~Animal()= default; //virtual destructor

};

class Dog: public Animal{ //derived class

public:

void makesound() const override {

cout<<"Woof"<<endl;

}

};

class Cat: public Animal{ //derived class

public:

void makesound() const override {

cout<<"Meow"<<endl;

}

};

class Bird: public Animal{ //derived class

public:

void makesound() const override {

cout<<"Chirp"<<endl;

}

};

shared\_ptr<Animal> createanimal(const string& type) //function to create animal based on type

{

if(type=="Dog"){

return make\_shared<Dog>();

}

else if(type=="Cat"){

return make\_shared<Cat>();

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

return make\_shared<Bird>();

}

else{

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

}

}

int main()

{

try{

vector<shared\_ptr<Animal>> animals; //creating vector using smart pointer

animals.push\_back(createanimal("Dog")); //adding animals to vector

animals.push\_back(createanimal("Cat"));

animals.push\_back(createanimal("Bird"));

//animals.push\_back(createanimal("Snake")); //error case

for(const auto& animal:animals){ //loop to call makesound for each animal

}

}catch(const exception& e){

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

}

return 0;

}

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>

using namespace std;

void rolldice(int sides,int rolls) //function to roll dice

{

random\_device rd;

mt19937 gen(rd());

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

cout<<"Rolling a "<<sides<<" sided dice "<<rolls<<" times \n";

for(int i=0;i<rolls;++i)

{

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

}

cout<<endl;

}

void flipcoin(int flips) //function to flip coin

{

random\_device rd;

mt19937 gen(rd());

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

cout<<"Flipping a coin "<<flips<<" times \n";

for(int i=0;i<flips;++i)

{

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

}

cout<<endl;

}

void generatetemp( int count,int mintemp,int maxtemp,const string& distribution) //function to generate random temperature

{

random\_device rd;

mt19937 gen(rd());

cout<<"Generating "<<count<<" random temp between "<<mintemp<<" and "<<maxtemp<<"\n";

if(distribution=="uniform") //for uniform

{

uniform\_real\_distribution<> dis(mintemp,maxtemp);

for(int i=0;i<count;++i)

{

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

}

}

else if(distribution=="normal") { //for normal

double mean=(mintemp+maxtemp)/2.0;

double stddev=(maxtemp-mintemp)/6.0;

normal\_distribution<> dis(mean,stddev);

for(int i=0;i<count;++i)

{

double temp;

do{

temp=dis(gen);

}while(temp<mintemp || temp>maxtemp);

cout<<temp;

}

}

else {

cerr<<"Unknown distribution type : "<<distribution<<endl; //error for unknown type

}

cout<<endl;

}

int main()

{

int ch;

cout<<"Choose an option: \n";

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

cout<<"2. Flip a coin \n";

cout<<"3. Generate random temperature \n";

cin>>ch;

switch(ch)

{

case 1:

{

int sides,rolls;

cout<<"Enter the number of sides on the dice\n";

cin>>sides;

cout<<"Enter the number of rolls\n";

cin>>rolls;

rolldice(sides,rolls);

break;

}

case 2:

{

int flips;

cout<<"Enter the no. of coin flips\n";

cin>>flips;

flipcoin(flips);

break;

}

case 3:

{

int count;

double mintemp,maxtemp;

string distribution;

cout<<"Enter the no. of temp. to generate:\n";

cin>>count;

cout<<"Enter the minimum temp:\n";

cin>>mintemp;

cout<<"Enter the maximum temp:\n";

cin>>maxtemp;

cout<<"Enter the distribution type (uniform/normal): ";

cin>>distribution;

generatetemp(count,mintemp,maxtemp,distribution);

break;

}

default:

cerr<<"Invalid Choice "<<endl; //error

break;

}

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>

using namespace std;

class FileIOException:public runtime\_error { // Custom exception for file I/O errors

public:

explicit FileIOException(const string& message): runtime\_error(message) {}

};

void extractErrors(const string& inputFile,const string& outputFile,const regex& pattern) // Function to read a log file and extract lines matching the regex pattern

{

ifstream inFile(inputFile);

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

throw FileIOException("Error opening input file "+inputFile);

}

ofstream outFile(outputFile);

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

throw FileIOException("Error opening output file "+outputFile);

}

string line;

while(getline(inFile,line))

{

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

outFile<<line<<endl;

}

}

if(inFile.bad()){

throw FileIOException("Error reading from input file "+inputFile);

}

if(outFile.bad()){

throw FileIOException("Error reading from output file "+outputFile);

}

inFile.close();

outFile.close();

}

int main()

{

string inputFile;

string outputFile;

string errorPattern;

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

cin>>inputFile;

cout<<"\nEnter the output file: ";

cin>>outputFile;

cout<<"\nEnter the regex pattern to search: ";

cin.ignore();

getline(cin,errorPattern);

try{

regex pattern(errorPattern);

extractErrors(inputFile,outputFile,pattern);

cout<<"Error extraction complete. "<<outputFile<<endl;

}catch(const regex\_error& e)

{

cerr<<"Invalid regex pattern: "<<e.what()<<endl;

}catch(const FileIOException& e)

{

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

}catch(const exception& e)

{

cerr<<"An unexpected error occured"<<e.what()<<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<algorithm>

#include<functional>

using namespace std;

class LargeObject{

private:

size\_t size;

int\* data;

public:

LargeObject(size\_t size): size(size),data(new int[size])

{

cout<<"Constructing large object of size "<<size<<endl;

}

LargeObject(const LargeObject& other): //copy constructor

size(other.size), data(new int[other.size]){

copy(other.data,other.data+other.size,data);

cout<<"Copy Constructing LargeObject of size "<<size<<endl;

}

LargeObject(LargeObject&& other) noexcept: //move constructor

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

other.size=0;

other.data=nullptr;

cout<<"Moving constructing LargeObject of size "<<size<<endl;

}

LargeObject& operator=(const LargeObject& other) //copy assignment operator

{

if(this== &other){

return \*this;

}

delete[] data;

size=other.size;

data=new int[size];

copy(other.data,other.data+size,data);

cout<<"Copy assigning LargeObject of size "<<size<<endl;

return \*this;

}

LargeObject& operator=(LargeObject&& other) noexcept //move assignment operator

{

if(this== &other){

return \*this;

}

delete[] data;

size=other.size;

data=other.data;

other.size = 0;

other.data = nullptr;

cout<<"Move assigning LargeObject of size "<<size<<endl;

return \*this;

}

~LargeObject(){ //destructor

delete[] data;

cout<<"Destructing LargeObject of size "<<size<<endl;

}

size\_t getSize() const {

return size;

}

};

class Container{

private:

vector<LargeObject> objects;

public:

void add(LargeObject&& obj){ //add a LargeObject to container

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

}

void processObjects(const function<void(LargeObject&)>& func) // process all objects in the container using a lambda

{

for(auto& obj: objects){

func(obj);

}

}

void iterate(const function<void(const LargeObject&)>& func) const{ //custom iterator using lamda

for\_each(objects.begin(),objects.end(),func);

}

};

int main()

{

Container c;

c.add(LargeObject(1000000)); //add a LargeObject to container

c.add(LargeObject(2000000));

c.processObjects([](LargeObject& obj) // process objects using a lambda

{

cout<<"Processing Object of size "<<obj.getSize()<<endl;

});

c.iterate([](const LargeObject& obj){ //custom iterator using lamda

cout<<"iterating over object of size "<<obj.getSize()<<endl;

});

return 0;

}