// Online C++ compiler to run C++ program online

#include<conio.h>

#include<fstream.h>

class student{

public:

struct stu

{

char name[20];

int roll;

}s;

void put\_data();

void get\_data();

};

void student::put\_data(){

cout<<"enter name";

cin>>s.name;

cout<<"enter roll";

cin>>s.roll;

file.open("hit.txt",ios::out|ios::app);

file.write((char\*)this,sizeof(student));

file.close();

//getch();

get\_data();

}

void main()

{

clrscr();

student st;

st.put\_data();

}

2.)

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

// Function prototypes

void addStudent();

void displayStudents();

void searchStudent();

void addContact();

void searchContact();

void encryptFile(const string &fileName, int key);

void decryptFile(const string &fileName, int key);

int main() {

int choice;

while (true) {

cout << "Menu:\n";

cout << "1. Add Student Record\n";

cout << "2. Display All Student Records\n";

cout << "3. Search Student by ID\n";

cout << "4. Add Contact\n";

cout << "5. Search Contact by Name\n";

cout << "6. Encrypt File\n";

cout << "7. Decrypt File\n";

cout << "8. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

addStudent();

break;

case 2:

displayStudents();

break;

case 3:

searchStudent();

break;

case 4:

addContact();

break;

case 5:

searchContact();

break;

case 6: {

string fileName;

int key;

cout << "Enter file name to encrypt: ";

cin >> fileName;

cout << "Enter encryption key: ";

cin >> key;

encryptFile(fileName, key);

break;

}

case 7: {

string fileName;

int key;

cout << "Enter file name to decrypt: ";

cin >> fileName;

cout << "Enter decryption key: ";

cin >> key;

decryptFile(fileName, key);

break;

}

case 8:

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

return 0;

}

// Function to add a new student record

void addStudent() {

ofstream outFile("students.txt", ios::app);

string name, id;

int marks;

cout << "Enter student name: ";

cin >> name;

cout << "Enter student ID: ";

cin >> id;

cout << "Enter student marks: ";

cin >> marks;

outFile << name << " " << id << " " << marks << endl;

outFile.close();

cout << "Student record added successfully.\n";

}

// Function to display all student records

void displayStudents() {

ifstream inFile("students.txt");

string name, id;

int marks;

while (inFile >> name >> id >> marks) {

cout << "Name: " << name << ", ID: " << id << ", Marks: " << marks << endl;

}

inFile.close();

}

// Function to search for a student by ID

void searchStudent() {

ifstream inFile("students.txt");

string name, id, searchId;

int marks;

bool found = false;

cout << "Enter student ID to search: ";

cin >> searchId;

while (inFile >> name >> id >> marks) {

if (id == searchId) {

cout << "Name: " << name << ", ID: " << id << ", Marks: " << marks << endl;

found = true;

break;

}

}

inFile.close();

if (!found) {

cout << "Student with ID " << searchId << " not found.\n";

}

}

// Function to add a new contact

void addContact() {

ofstream outFile("contacts.txt", ios::app);

string name, phoneNumber;

cout << "Enter contact name: ";

cin >> name;

cout << "Enter contact phone number: ";

cin >> phoneNumber;

outFile << name << " " << phoneNumber << endl;

outFile.close();

cout << "Contact added successfully.\n";

}

// Function to search for a contact by name

void searchContact() {

ifstream inFile("contacts.txt");

string name, phoneNumber, searchName;

bool found = false;

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

cin >> searchName;

while (inFile >> name >> phoneNumber) {

if (name == searchName) {

cout << "Name: " << name << ", Phone Number: " << phoneNumber << endl;

found = true;

break;

}

}

inFile.close();

if (!found) {

cout << "Contact with name " << searchName << " not found.\n";

}

}

// Function to encrypt a file using Caesar cipher

void encryptFile(const string &fileName, int key) {

ifstream inFile(fileName);

ofstream outFile(fileName + ".enc");

char ch;

while (inFile.get(ch)) {

outFile.put(ch + key);

}

inFile.close();

outFile.close();

cout << "File encrypted successfully.\n";

}

// Function to decrypt a file using Caesar cipher

void decryptFile(const string &fileName, int key) {

ifstream inFile(fileName);

ofstream outFile(fileName + ".dec");

char ch;

while (inFile.get(ch)) {

outFile.put(ch - key);

}

inFile.close();

outFile.close();

cout << "File decrypted successfully.\n";

}

3.) #include <iostream>

#include <fstream>

#include <string>

using namespace std;

// Function prototypes

void copyImage(const string &sourceFile, const string &destFile);

void addItem();

void displayItems();

void updateQuantity();

void saveHighScore();

void displayHighScore();

struct Item {

char name[50];

double price;

int quantity;

};

struct HighScore {

char playerName[50];

int score;

};

int main() {

int choice;

while (true) {

cout << "Menu:\n";

cout << "1. Copy Image\n";

cout << "2. Add New Inventory Item\n";

cout << "3. Display All Inventory Items\n";

cout << "4. Update Item Quantity\n";

cout << "5. Save High Score\n";

cout << "6. Display High Score\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

string sourceFile, destFile;

cout << "Enter source image file name: ";

cin >> sourceFile;

cout << "Enter destination image file name: ";

cin >> destFile;

copyImage(sourceFile, destFile);

break;

}

case 2:

addItem();

break;

case 3:

displayItems();

break;

case 4:

updateQuantity();

break;

case 5:

saveHighScore();

break;

case 6:

displayHighScore();

break;

case 7:

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

return 0;

}

// Function to copy an image file

void copyImage(const string &sourceFile, const string &destFile) {

ifstream inFile(sourceFile, ios::binary);

ofstream outFile(destFile, ios::binary);

if (!inFile) {

cout << "Error opening source file.\n";

return;

}

if (!outFile) {

cout << "Error creating destination file.\n";

return;

}

outFile << inFile.rdbuf();

cout << "Image copied successfully.\n";

}

// Function to add a new inventory item

void addItem() {

ofstream outFile("inventory.dat", ios::binary | ios::app);

Item item;

cout << "Enter item name: ";

cin.ignore();

cin.getline(item.name, 50);

cout << "Enter item price: ";

cin >> item.price;

cout << "Enter item quantity: ";

cin >> item.quantity;

outFile.write(reinterpret\_cast<char\*>(&item), sizeof(item));

outFile.close();

cout << "Item added successfully.\n";

}

// Function to display all inventory items

void displayItems() {

ifstream inFile("inventory.dat", ios::binary);

Item item;

while (inFile.read(reinterpret\_cast<char\*>(&item), sizeof(item))) {

cout << "Name: " << item.name << ", Price: " << item.price << ", Quantity: " << item.quantity << endl;

}

inFile.close();

}

// Function to update the quantity of an existing item

void updateQuantity() {

fstream file("inventory.dat", ios::binary | ios::in | ios::out);

Item item;

char searchName[50];

cout << "Enter the name of the item to update: ";

cin.ignore();

cin.getline(searchName, 50);

bool found = false;

while (file.read(reinterpret\_cast<char\*>(&item), sizeof(item))) {

if ((item.name, searchName) == 0) {

cout << "Enter new quantity: ";

cin >> item.quantity;

file.seekp(-sizeof(item), ios::cur);

file.write(reinterpret\_cast<char\*>(&item), sizeof(item));

found = true;

cout << "Item quantity updated successfully.\n";

break;

}

}

if (!found) {

cout << "Item not found.\n";

}

file.close();

}

// Function to save a new high score

void saveHighScore() {

ofstream outFile("highscores.dat", ios::binary | ios::app);

HighScore score;

cout << "Enter player name: ";

cin.ignore();

cin.getline(score.playerName, 50);

cout << "Enter score: ";

cin >> score.score;

outFile.write(reinterpret\_cast<char\*>(&score), sizeof(score));

outFile.close();

cout << "High score saved successfully.\n";

}

// Function to display the current high score

void displayHighScore() {

ifstream inFile("highscores.dat", ios::binary);

HighScore score, highScore;

bool first = true;

while (inFile.read(reinterpret\_cast<char\*>(&score), sizeof(score))) {

if (first || score.score > highScore.score) {

highScore = score;

first = false;

}

}

if (!first) {

cout << "Player: " << highScore.playerName << ", Score: " << highScore.score << endl;

} else {

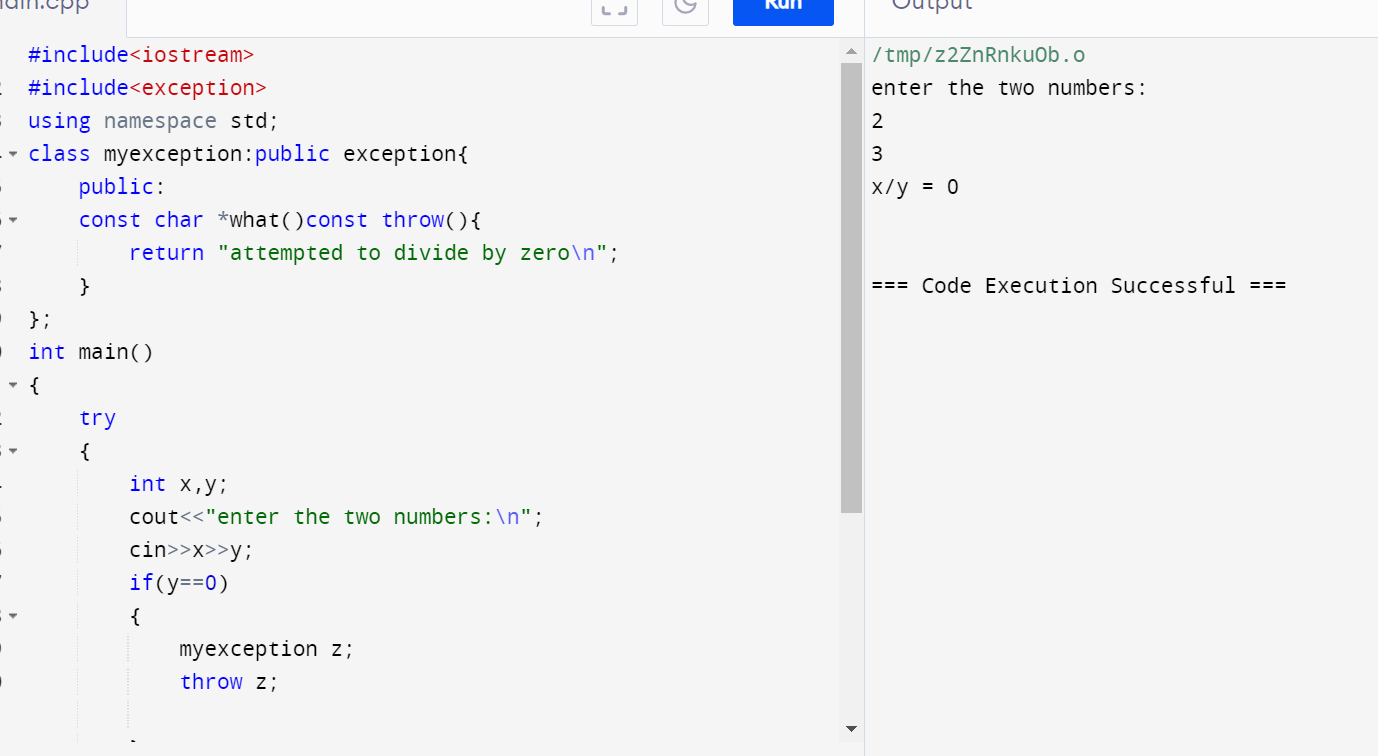
cout << "No high scores recorded.\n";

}

inFile.close();

}





1.) What are the advantages and disadvantages of using exceptions in C++ compared to traditional error codes?

Using exceptions in C++ offers several advantages and disadvantages compared to traditional error codes:

**Advantages of Exceptions**

1. **Separation of Error Handling and Logic**:
   * **Clarity**: Exceptions allow you to separate error handling from the main logic, making the code easier to read and maintain.
   * **Centralized Handling**: Exceptions can be caught and handled in a centralized place, avoiding scattered error handling code.
2. **Automatic Propagation**:
   * **Propagating Errors**: Exceptions automatically propagate up the call stack until they are caught, which simplifies error propagation compared to manually checking and returning error codes.
3. **Enhanced Readability and Maintainability**:
   * **Cleaner Code**: Code with exceptions can be cleaner and less cluttered since it doesn’t require constant checking of return values.
   * **Focus on Logic**: Programmers can focus on the main logic, handling errors separately.
4. **Rich Error Information**:
   * **Detailed Errors**: Exceptions can carry rich information, including error messages, error codes, and context-specific data.

**Disadvantages of Exceptions**

1. **Performance Overhead**:
   * **Runtime Cost**: Exceptions can introduce runtime overhead due to the need to maintain additional information for stack unwinding.
   * **Predictability**: The performance impact can be unpredictable and may vary between different compilers and environments.
2. **Complexity**:
   * **Learning Curve**: Understanding and using exceptions properly can be more complex for beginners.
   * **Misuse Risks**: Incorrect use of exceptions (e.g., using exceptions for regular control flow) can lead to inefficient and hard-to-maintain code.
3. **Non-Standardized Error Handling**:
   * **Inconsistent Handling**: Mixing exceptions and error codes in the same codebase can lead to inconsistent error handling practices.
   * **Legacy Code**: Integrating exceptions into legacy systems that use error codes can be challenging.
4. **Exception Safety**:
   * **Strong Guarantees**: Ensuring exception safety (basic, strong, or no-throw guarantees) can complicate code design and implementation.
   * **Resource Management**: Properly managing resources (like memory and file handles) in the presence of exceptions requires careful use of RAII (Resource Acquisition Is Initialization) and smart pointers.

2.) How can you ensure that exception classes provide informative error messages for debugging?

To ensure that exception classes provide informative error messages for debugging, follow these guidelines:

### 1. Inherit from std::exception

Inheriting from std::exception allows you to override the what() method, which is used to retrieve a descriptive error message.

### 2. Use Descriptive Error Messages

Include detailed and specific information in your error messages. Provide context that can help identify the cause of the error.

### 3. Include Relevant Data

Include relevant data such as variable values, function names, and context-specific details in the exception message.

### 4. Implement Custom Exception Classes

Create custom exception classes tailored to your application's needs. This can help categorize different types of errors and provide more specific information.

### 5. Utilize Constructor Initialization

Use constructor initialization to set the error message when the exception object is created. This ensures the message is set at the time of the error.

3.) Discuss strategies for optimizing exception handling performance, especially in performance-critical applications.

Optimizing exception handling performance is crucial in performance-critical applications where even minor inefficiencies can lead to significant overhead. Here are some strategies to optimize exception handling performance in C++:

**1. Minimize Exception Usage**

* **Avoid Exceptions for Regular Control Flow**: Use exceptions only for exceptional cases, not for regular control flow. Reserve exceptions for truly unexpected events.
* **Use Error Codes for Expected Errors**: In scenarios where errors are expected and frequent, consider using error codes instead of exceptions.

**2. Optimize Exception Throwing and Catching**

* **Preallocate Exception Objects**: If you must throw exceptions frequently, consider preallocating exception objects to avoid the overhead of memory allocation during exception throwing.
* **Narrow Scope of Try-Catch Blocks**: Minimize the scope of try-catch blocks to reduce the overhead of setting up exception handling context.

**3. Use noexcept Keyword**

* **Mark Functions as noexcept**: Mark functions that are not expected to throw exceptions with the noexcept keyword. This allows the compiler to optimize better and can improve performance.

**4. Leverage Compiler Optimizations**

* **Enable Compiler Optimizations**: Use compiler optimization flags (e.g., -O2 or -O3 for GCC) to let the compiler optimize exception handling code.
* **Profile and Optimize**: Profile your code to identify hot spots related to exception handling and optimize those areas specifically.

**5. Use RAII for Resource Management**

* **Resource Acquisition Is Initialization (RAII)**: Use RAII to manage resources like memory, file handles, and locks. This ensures resources are properly released even if exceptions are thrown, reducing the need for explicit try-catch blocks for cleanup.

**6. Avoid Dynamic Allocation in Exceptions**

* **Static Strings and Data**: Use static strings and data within exception objects to avoid the overhead of dynamic memory allocation during exception handling.

**7. Minimize Exception Stack Unwinding**

* **Catch by Reference**: Always catch exceptions by reference to avoid unnecessary copying of exception objects.
* **Minimal Destructors**: Ensure destructors are lightweight to minimize the overhead during stack unwinding.

**8. Use Efficient Exception Hierarchies**

* **Flat Hierarchies**: Use flat exception hierarchies where possible. Deep inheritance hierarchies can introduce additional overhead.

4.) How can you design a hierarchy of exception classes for improved code maintainability and reusability?

Designing a hierarchy of exception classes can improve code maintainability and reusability by organizing exceptions in a way that makes it easy to handle related errors together and extend the hierarchy as needed. Here's a strategy for designing an effective hierarchy of exception classes:

### 1. ****Define a Base Exception Class****

Create a base exception class from which all other exception classes will inherit. This class should inherit from std::exception and provide a common interface for all exceptions.

### 2. ****Categorize Exceptions****

Group exceptions into categories based on the type of errors they represent. For example, you might have categories for I/O errors, logic errors, and application-specific errors.

### 3. ****Create Specific Exception Classes****

For each category, create specific exception classes that inherit from the base exception class. These classes can further be subclassed to represent more specific errors.

### 4. ****Provide Informative Messages****

Ensure that each exception class provides informative error messages and relevant data. Override the what() method to return these messages.

### 5. ****Use Standard Library Exceptions****

Where applicable, use or inherit from standard library exceptions (e.g., std::runtime\_error, std::logic\_error) to leverage existing functionality.

5.) When might it be appropriate to not use exceptions in C++ for error handling? Explain your reasoning.

While exceptions are a powerful mechanism for error handling in C++, there are scenarios where using exceptions might not be appropriate. Here are some situations where avoiding exceptions can be beneficial, along with the reasoning:

**1. Performance-Critical Code**

* **Reasoning**: Exceptions can introduce performance overhead due to the cost of stack unwinding and the need to maintain additional information for exception handling. In performance-critical code, where every microsecond counts (e.g., real-time systems, high-frequency trading), this overhead can be unacceptable.
* **Alternative**: Use error codes or status flags to indicate errors, allowing the code to run faster by avoiding the overhead of exceptions.

**2. Resource-Constrained Environments**

* **Reasoning**: In environments with limited resources, such as embedded systems, the additional memory and processing requirements for handling exceptions might be too costly.
* **Alternative**: Use simpler error handling mechanisms, such as return values, to minimize resource consumption.

**3. Code with High Frequency of Errors**

* **Reasoning**: If errors are expected to occur frequently, using exceptions can lead to a significant performance hit due to the cost of throwing and catching exceptions.
* **Alternative**: Use error codes or other non-exception-based mechanisms to handle frequent errors efficiently.

**4. Interfacing with C or Other Languages**

* **Reasoning**: When writing C++ code that needs to interface with C libraries or other languages that do not support exceptions, using exceptions can complicate the integration.
* **Alternative**: Use error codes or other error handling mechanisms compatible with the interfaced language.

**5. Simple Control Flow**

* **Reasoning**: In cases where the error handling is straightforward and can be easily managed with return values, using exceptions might be overkill and add unnecessary complexity.
* **Alternative**: Use return values or status flags for simple error handling scenarios.

**6. Low-Level System Code**

* **Reasoning**: Low-level system code, such as operating system kernels or hardware drivers, often has strict performance and reliability requirements. Exceptions can introduce unpredictable performance and complicate debugging.
* **Alternative**: Use traditional error handling methods, such as error codes, which provide more predictable performance and simpler debugging.

**7. Legacy Codebases**

* **Reasoning**: In legacy codebases that were originally written without exceptions, introducing exceptions can lead to inconsistency and complicate maintenance. Consistency in error handling methods is crucial for maintainability.
* **Alternative**: Continue using the existing error handling mechanisms, such as error codes, to maintain consistency.

**8. Code with Non-Exception-Safe Operations**

* **Reasoning**: If the codebase contains operations that are not exception-safe (i.e., they may leave the program in an inconsistent state if an exception is thrown), using exceptions can lead to bugs and undefined behavior.
* **Alternative**: Use error handling methods that do not rely on exceptions to ensure program stability and consistency.

2.) #include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <string>

#include <exception>

// Custom exception for file opening failure

class FileOpenError : public std::exception {

public:

const char\* what() const noexcept override {

return "Error: Unable to open file.";

}

};

// Custom exception for invalid data format

class InvalidDataFormatException : public std::exception {

private:

std::string message;

public:

explicit InvalidDataFormatException(const std::string& msg) : message(msg) {}

const char\* what() const noexcept override {

return message.c\_str();

}

};

// Custom exception for calculation errors

class CalculationError : public std::exception {

private:

std::string message;

public:

explicit CalculationError(const std::string& msg) : message(msg) {}

const char\* what() const noexcept override {

return message.c\_str();

}

};

// Function prototypes

void readFileAndProcess(const std::string& fileName);

void validateAndCalculate(const std::vector<std::string>& lines);

int main() {

std::string fileName = "data.txt";

try {

readFileAndProcess(fileName);

} catch (const FileOpenError& e) {

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

} catch (const InvalidDataFormatException& e) {

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

} catch (const CalculationError& e) {

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

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

std::cerr << "An unexpected error occurred: " << e.what() << std::endl;

}

return 0;

}

// Function to read file and process data

void readFileAndProcess(const std::string& fileName) {

std::ifstream file(fileName);

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

throw FileOpenError();

}

std::vector<std::string> lines;

std::string line;

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

lines.push\_back(line);

}

file.close();

validateAndCalculate(lines);

}

// Function to validate data and perform calculations

void validateAndCalculate(const std::vector<std::string>& lines) {

for (const auto& line : lines) {

std::istringstream stream(line);

double value1, value2;

// Expecting two double values per line

if (!(stream >> value1 >> value2)) {

throw InvalidDataFormatException("Error: Invalid data format in line: " + line);

}

// Perform calculation (e.g., division)

if (value2 == 0) {

throw CalculationError("Error: Division by zero in line: " + line);

}

double result = value1 / value2;

std::cout << "Result of division: " << result << std::endl;

}

}