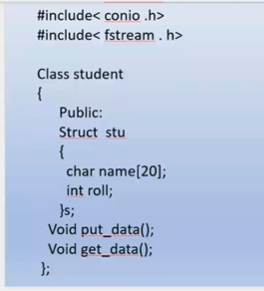
**Day 15 C++**

FILE-HANDLING:



#include <conio.h>

#include<fstream.h>

class student{

public:

Struct stu

{

char name[20];

int roll;

}s;

void put\_data();

void get\_data();

};

2. 

#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();

}

3.

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

// Function to create a text file

void createTextFile(const string& filename) {

ofstream outfile(filename);

if (outfile.is\_open()) {

outfile << "This is a sample text file.\n";

outfile << "You can add more content here.\n";

cout << "Text file " << filename << " created successfully!" << endl;

} else {

cerr << "Error creating file: " << filename << endl;

}

outfile.close(); // Close the file even on errors

}

// Function to read from a text file

void readTextFile(const string& filename) {

ifstream infile(filename);

if (infile.is\_open()) {

string line;

while (getline(infile, line)) {

cout << line << endl;

}

} else {

cerr << "Error opening file: " << filename << endl;

}

infile.close(); // Close the file even on errors

}

// Function to write to a binary file

void writeBinaryFile(const string& filename, const char\* data, int size) {

ofstream outfile(filename, ios::binary);

if (outfile.is\_open()) {

outfile.write(data, size);

cout << "Binary data written to file " << filename << endl;

} else {

cerr << "Error creating binary file: " << filename << endl;

}

outfile.close(); // Close the file even on errors

}

// Function to read from a binary file

void readBinaryFile(const string& filename, int size) {

char buffer[size];

ifstream infile(filename, ios::binary);

if (infile.is\_open()) {

infile.read(buffer, size);

cout << "Binary data from file " << filename << ":" << endl;

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

cout << hex << static\_cast<int>(buffer[i]) << " ";

}

cout << endl;

} else {

cerr << "Error opening binary file: " << filename << endl;

}

infile.close(); // Close the file even on errors

}

int main() {

string textFilename = "example.txt";

string binaryFilename = "data.bin";

// Create a text file

createTextFile(textFilename);

// Read from the text file

readTextFile(textFilename);

// Sample data for binary file

char binaryData[] = "This is binary data";

// Write to a binary file

writeBinaryFile(binaryFilename, binaryData, sizeof(binaryData));

// Read from the binary file (adjust size based on written data)

readBinaryFile(binaryFilename, sizeof(binaryData));

return 0;

}

Text Files:

Student Records: Create a program that allows users to enter student information (name, ID, marks) and store them in a text file. The program should allow users to:

Add new student records.

Display all student records from the file.

Search for a specific student by ID and display their details.

Phonebook: Develop a program that functions as a simple phonebook. Users can:

Add new contacts (name, phone number) to the file.

Search for a contact by name and display their phone number.

File Encryption/Decryption (Optional): Implement a program that encrypts/decrypts a text file using a simple Caesar cipher or another basic encryption method.

#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";

}

EXCEPTION HANDLING: TRY/CATCH/THROW:

DIVISION

#include<iostream>

using namespace std;

float division (int x,int y){

if(y == 0){

throw"attempted to divide by zero!";

}

return(x/y);

}

int main(){

int i =25;

int j =0;

float k =0;

try{

k = division(i,j);

cout<<k<<endl;

}

catch (const char\* e){

cerr<<e<<endl;

}

return 0;

}

ADDITION:

#include<iostream>

using namespace std;

float addition (int x,int y){

if(y == 0){

throw"attempted to addition by zero!";

}

return(x+y);

}

int main(){

int i =25;

int j =0;

float k =0;

try{

k = addition(i,j);

cout<<k<<endl;

}

catch (const char\* e){

cerr<<e<<endl;

}

return 0;

}

SUBSTRACTION :

#include<iostream>

using namespace std;

float substraction (int x,int y){

if(y == 0){

throw"attempted to substraction by zero!";

}

return(x-y);

}

int main(){

int i =25;

int j =0;

float k =0;

try{

k = substraction(i,j);

cout<<k<<endl;

}

catch (const char\* e){

cerr<<e<<endl;

}

return 0;

}

MULTIPLICATION:

#include<iostream>

using namespace std;

float multiplication (int x,int y){

if(y == 0){

throw"attempted to multiplication by zero!";

}

return(x\*y);

}

int main(){

int i =25;

int j =0;

float k =0;

try{

k = multiplication(i,j);

cout<<k<<endl;

}

catch (const char\* e){

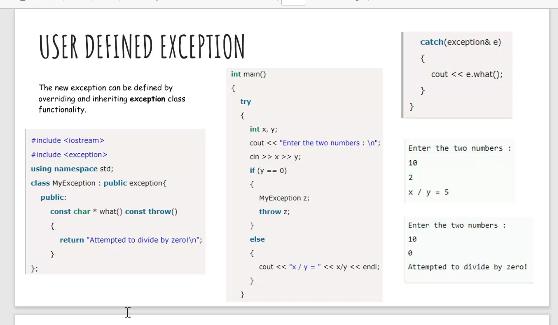
cerr<<e<<endl;

}

return 0;

}

USER DEFINED EXCEPTION :



TASKS:

ONE:

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

Using exceptions in C++ has several advantages and disadvantages compared to traditional error codes. Here are some of the key points:

**Advantages of Using Exceptions**

1. **Separation of Error Handling Code from Regular Code:**
   * Exceptions allow error handling code to be separated from the main logic of the program. This leads to cleaner, more readable code, as the main logic is not cluttered with error handling.
2. **Automatic Propagation:**
   * Exceptions propagate automatically up the call stack until they are caught by an appropriate handler. This can simplify error handling, as functions do not need to check and return error codes manually.
3. **Uniform Error Handling:**
   * With exceptions, different types of errors can be handled uniformly. A single try-catch block can handle multiple types of exceptions, making the code more flexible and easier to maintain.
4. **Resource Management:**
   * Exceptions work well with RAII (Resource Acquisition Is Initialization) and the concept of destructors. When an exception is thrown, destructors for all objects in scope are automatically called, which helps in managing resources like memory, file handles, and other resources properly.
5. **Encapsulation:**
   * Exceptions help encapsulate error handling within objects and functions, which aligns with the principles of object-oriented programming. This makes the code more modular and reusable.

**Disadvantages of Using Exceptions**

1. **Performance Overhead:**
   * Exception handling can introduce performance overhead due to the additional runtime checks and the complexity of the underlying mechanisms. This overhead might be significant in performance-critical applications.
2. **Complexity:**
   * Exceptions can add complexity to the code, especially if not used consistently and appropriately. Misuse of exceptions can lead to difficult-to-debug issues and can make the control flow harder to understand.
3. **Non-local Control Flow:**
   * Exceptions can result in non-local control flow, making it harder to track the program's execution path. This can complicate debugging and reasoning about the program, especially in large codebases.
4. **Not Always Suitable:**
   * Exceptions are not always the best choice for every error handling scenario. For example, simple validation checks and recoverable errors might be better handled with traditional error codes or other mechanisms.
5. **Compatibility:**
   * Some legacy codebases and APIs still use traditional error codes, and mixing both approaches can lead to inconsistencies and additional complexity in the code.

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

Ensuring that exception classes provide informative error messages for debugging is crucial for effective error handling and troubleshooting. Here are some best practices to achieve this:

**1. Use Descriptive Exception Class Names**

* Create custom exception classes that have descriptive names reflecting the nature of the error.
* Example: FileNotFoundException, InvalidUserInputException, NetworkTimeoutException.

**2. Inherit from Standard Exception Classes**

* Inherit from std::exception or other standard exception classes to maintain compatibility with standard exception handling mechanisms.
* Example:

class MyCustomException : public std::exception {

public:

MyCustomException(const std::string& message) : msg(message) {}

virtual const char\* what() const noexcept override {

return msg.c\_str();

}

private:

std::string msg;

};

**3. Provide Detailed Error Messages**

* Pass detailed error messages to the exception constructor. Include relevant information such as variable values, function names, or states.
* Example:

if (fileNotFound) {

throw FileNotFoundException("File not found: " + fileName);

}

**4. Use Constructor Initialization Lists**

* Use constructor initialization lists to initialize the error message in custom exception classes.
* Example:

class FileNotFoundException : public std::exception {

public:

FileNotFoundException(const std::string& file)

: message("File not found: " + file) {}

virtual const char\* what() const noexcept override {

return message.c\_str();

}

private:

std::string message;

};

**5. Include Contextual Information**

* Provide additional context in the error message, such as the function name, file name, line number, or specific conditions that led to the error.
* Example:

class DatabaseException : public std::exception {

public:

DatabaseException(const std::string& msg, const std::string& func, int line)

: message("Error in " + func + " at line " + std::to\_string(line) + ": " + msg) {}

virtual const char\* what() const noexcept override {

return message.c\_str();

}

private:

std::string message;

};

#define THROW\_DB\_EXCEPTION(msg) throw DatabaseException(msg, \_\_FUNCTION\_\_, \_\_LINE\_\_)

**6. Use Nested Exceptions for Better Context**

* Use nested exceptions (std::nested\_exception) to provide a chain of exceptions, giving more context on the error.
* Example:

try {

// Code that may throw

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

std::throw\_with\_nested(CustomException("Failed to process data"));

}

**7. Log Exception Details**

* Log detailed exception information including the message and stack trace (if possible) to help with post-mortem analysis.
* Example:

try {

// Code that may throw

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

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

// Additional logging code

}

**8. Provide Accessor Methods**

* Provide accessor methods in custom exception classes to retrieve additional error information.
* Example:

class ValidationException : public std::exception {

public:

ValidationException(const std::string& field, const std::string& issue)

: field(field), issue(issue), message("Validation error in " + field + ": " + issue) {}

virtual const char\* what() const noexcept override {

return message.c\_str();

}

std::string getField() const { return field; }

std::string getIssue() const { return issue; }

private:

std::string field;

std::string issue;

std::string message;

};

**9. Use Standard Library Facilities**

* Use facilities provided by the standard library, such as std::system\_error, to represent errors with additional context like error codes and messages.
* Example:

throw std::system\_error(std::make\_error\_code(std::errc::io\_error), "I/O operation failed");

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

### 1. Use Exceptions for Exceptional Situations Only

**Principle**: Exceptions should be reserved for truly exceptional situations, not for routine control flow or expected errors.

**Strategy**: Handle predictable errors (e.g., file not found, end of data) using return codes or status flags instead of exceptions.

// Instead of throwing an exception for a file not found

bool readFile(const std::string& fileName) {

if (!fileExists(fileName)) {

return false; // Use return code instead

}

// File reading logic

return true;

}

### 2. Avoid Exceptions in Performance-Critical Paths

**Principle**: Exceptions introduce overhead, so avoid them in the most performance-critical parts of your application.

**Strategy**: Design your code to prevent errors in critical paths by validating inputs and ensuring preconditions are met.

void process(int value) {

if (value <= 0) {

// Handle invalid value without exceptions

return;

}

// Critical processing logic

}

### 3. Use noexcept Keyword

**Principle**: The noexcept keyword informs the compiler that a function will not throw exceptions, enabling certain optimizations.

**Strategy**: Declare functions noexcept when they do not throw exceptions.

void process() noexcept {

// Function implementation that guarantees no exceptions

}

### 4. Minimize Stack Unwinding Overhead

**Principle**: Stack unwinding during exception handling can be costly.

**Strategy**: Keep functions shallow and avoid deep recursion to minimize the cost of unwinding the stack.

void shallowFunction() {

// Minimal stack depth to reduce unwinding cost

}

### 5. Use Efficient Exception Hierarchies

**Principle**: Catching exceptions efficiently can be optimized by designing a proper hierarchy.

**Strategy**: Organize exceptions in a hierarchy to catch specific exceptions without unnecessary checks.

class BaseException : public std::exception {};

class NetworkException : public BaseException {};

class TimeoutException : public NetworkException {};

### 6. Preallocate Resources

**Principle**: Allocating resources during exception handling can be costly.

**Strategy**: Preallocate resources such as memory or objects that might be needed in exception handling to avoid allocation overhead during the throw-catch process.

void preallocateResources() {

static char buffer[1024]; // Preallocated buffer

// Use buffer during exception handling

}

### 7. Minimize Dynamic Memory Usage

**Principle**: Dynamic memory allocation can slow down exception handling.

**Strategy**: Avoid using dynamic memory in exception objects. Instead, use stack allocation or static storage.

class MyException : public std::exception {

public:

MyException(const std::string& msg) : message(msg) {}

virtual const char\* what() const noexcept override {

return message.c\_str();

}

private:

std::string message; // Prefer fixed-size buffer for message

};

### 8. Optimize Exception Throwing and Catching

**Principle**: The process of throwing and catching exceptions can be optimized.

**Strategy**: Avoid creating complex exception objects and use efficient mechanisms for throwing and catching.

void foo() {

try {

// Code that might throw

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

// Handle exception efficiently

}

}

### 9. Use try-catch Sparingly

**Principle**: Frequent use of try-catch blocks can introduce performance overhead.

**Strategy**: Use try-catch blocks judiciously, only where necessary, and consolidate error handling in one place if possible.

void process() {

try {

// Code that might throw

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

// Handle all exceptions in one place

}

}

### 10. Profile and Measure

**Principle**: Actual performance impacts should be measured rather than assumed.

**Strategy**: Profile the application to identify bottlenecks related to exception handling and optimize based on real data. Tools like gprof, Valgrind, or built-in profiler tools in IDEs can help identify performance issues.

void profileApplication() {

// Use profiling tools to measure performance impact of exceptions

}

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

### . Base Exception Class

**Principle**: Create a base exception class that all other exceptions will inherit from. This allows for catching all application-specific exceptions with a single catch block if needed.

**Strategy**: Derive the base exception class from std::exception to leverage the standard exception handling mechanisms.

class MyBaseException : public std::exception {

public:

MyBaseException(const std::string& message) : msg(message) {}

virtual const char\* what() const noexcept override {

return msg.c\_str();

}

private:

std::string msg;

};

### 2. Domain-Specific Exception Classes

**Principle**: Create domain-specific exception classes that inherit from the base exception class. This helps in categorizing exceptions based on their functionality or module.

**Strategy**: Define exceptions for different domains or modules, like I/O operations, network operations, or user input.

class IOException : public MyBaseException {

public:

IOException(const std::string& message) : MyBaseException(message) {}

};

class NetworkException : public MyBaseException {

public:

NetworkException(const std::string& message) : MyBaseException(message) {}

};

class UserInputException : public MyBaseException {

public:

UserInputException(const std::string& message) : MyBaseException(message) {}

};

### 3. More Specific Exception Classes

**Principle**: Create more specific exception classes within each domain-specific exception class to handle detailed error conditions.

**Strategy**: Further derive exception classes for specific error conditions within a domain.

class FileNotFoundException : public IOException {

public:

FileNotFoundException(const std::string& fileName)

: IOException("File not found: " + fileName) {}

};

class NetworkTimeoutException : public NetworkException {

public:

NetworkTimeoutException(const std::string& host)

: NetworkException("Network timeout: " + host) {}

};

class InvalidUserInputException : public UserInputException {

public:

InvalidUserInputException(const std::string& input)

: UserInputException("Invalid user input: " + input) {}

};

### 4. Use Constructors for Detailed Messages

**Principle**: Provide constructors in your exception classes that accept parameters for detailed error messages.

**Strategy**: Pass relevant information like file names, host names, or invalid inputs to the exception constructors to create informative error messages.

class DatabaseException : public MyBaseException {

public:

DatabaseException(const std::string& message, const std::string& query)

: MyBaseException(message), query(query) {}

const std::string& getQuery() const { return query; }

private:

std::string query;

};

### 5. Add Custom Data and Methods

**Principle**: Enhance exception classes with custom data members and methods to provide more context and functionality.

**Strategy**: Add data members and accessor methods to your exception classes for additional information.

class ValidationException : public MyBaseException {

public:

ValidationException(const std::string& field, const std::string& issue)

: MyBaseException("Validation error in " + field + ": " + issue), field(field), issue(issue) {}

const std::string& getField() const { return field; }

const std::string& getIssue() const { return issue; }

private:

std::string field;

std::string issue;

};

### 6. Maintain a Consistent Naming Convention

**Principle**: Use a consistent naming convention for exception classes to improve readability and maintainability.

**Strategy**: Follow a naming pattern like <Domain><Specific>Error for exception classes.

class DatabaseConnectionError : public DatabaseException {

public:

DatabaseConnectionError(const std::string& dbName)

: DatabaseException("Failed to connect to database: " + dbName) {}

};

### 7. Document the Hierarchy

**Principle**: Document the exception hierarchy to provide clear guidance on when and how to use each exception class.

**Strategy**: Create documentation or comments explaining the purpose and usage of each exception class.

/\*\*

\* @brief Base class for all database-related exceptions.

\*/

class DatabaseException : public MyBaseException {

public:

DatabaseException(const std::string& message) : MyBaseException(message) {}

};

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

There are certain scenarios where it might be appropriate to not use exceptions in C++ for error handling. The decision to avoid exceptions often hinges on specific application requirements, performance considerations, or code maintainability. Here are some situations where not using exceptions might be justified, along with the reasoning:

**1. Performance-Critical Code**

**Reasoning**: Exceptions can introduce overhead due to stack unwinding and context-switching, which can be detrimental in performance-critical applications like real-time systems, embedded systems, or high-frequency trading platforms.

**Alternative**: Use return codes or error flags to handle errors more efficiently.

bool performCriticalTask() {

if (!preconditionMet()) {

return false; // Error handling with return code

}

// Critical task execution

return true;

}

**2. Simple or Small-Scale Applications**

**Reasoning**: For small-scale applications or simple scripts where the overhead of exception handling may not be justified, and the control flow is straightforward, using exceptions might add unnecessary complexity.

**Alternative**: Use simple error handling mechanisms like return codes.

int main() {

int result = simpleOperation();

if (result != 0) {

std::cerr << "Error occurred: " << result << std::endl;

return result;

}

// Continue with normal flow

}

**3. Legacy Codebases**

**Reasoning**: In large, mature codebases that predominantly use traditional error-handling methods, introducing exceptions might lead to inconsistency and integration challenges. It can also require significant refactoring.

**Alternative**: Stick to the existing error handling conventions used in the codebase.

int legacyFunction() {

// Legacy code with error codes

if (someErrorCondition) {

return -1; // Indicate error with return code

}

// Normal execution

return 0;

}

**4. Low-Level System Programming**

**Reasoning**: In low-level system programming, such as writing kernel code, device drivers, or certain parts of the standard library, the environment might not support exceptions, or the overhead of exceptions might be unacceptable.

**Alternative**: Use error codes and handle errors explicitly.

int readFromHardware() {

// Direct hardware access

if (hardwareError) {

return HARDWARE\_ERROR; // Return specific error code

}

return SUCCESS;

}

**5. Resource-Constrained Environments**

**Reasoning**: In environments with limited resources (e.g., embedded systems, microcontrollers), the additional memory and processing overhead associated with exceptions might not be feasible.

**Alternative**: Use minimalistic error handling methods.

bool readSensorData() {

if (sensorError) {

return false; // Return boolean to indicate error

}

// Process sensor data

return true;

}

**6. APIs and Libraries with C Compatibility**

**Reasoning**: When writing C++ libraries that need to be compatible with C code, using exceptions can be problematic because C does not support exceptions. This can make it difficult to handle errors consistently across the language boundary.

**Alternative**: Use error codes or status structures that are compatible with C.

extern "C" int libraryFunction() {

// C-compatible error handling

if (someErrorCondition) {

return ERROR\_CODE;

}

return SUCCESS;

}

**7. Predictable and Frequent Errors**

**Reasoning**: For predictable and frequently occurring errors, using exceptions might be overkill and can clutter the code with try-catch blocks, making it harder to read and maintain.

**Alternative**: Use return values or error flags to manage these errors.

bool parseInput(const std::string& input) {

if (input.empty()) {

return false; // Handle empty input without exceptions

}

// Parse input

return true;

}

**Summary**

* **Performance-Critical Code**: Avoid exceptions where overhead is unacceptable.
* **Simple or Small-Scale Applications**: Use simpler error handling methods for clarity.
* **Legacy Codebases**: Maintain consistency with existing error handling practices.
* **Low-Level System Programming**: Use error codes due to environment constraints.
* **Resource-Constrained Environments**: Avoid the overhead of exceptions.
* **APIs and Libraries with C Compatibility**: Use C-compatible error handling.
* **Predictable and Frequent Errors**: Use return codes or error flags to avoid clutter.

TASK2:

Develop a C++ program that demonstrates robust exception handling for file operations.

The program should:

Read data from a text file.

Validate the data format (e.g., expecting specific number of values per line).

Perform calculations based on the valid data.

Implement exception handling for the following error scenarios:

File opening failure: Throw a custom exception named FileOpenError if the file cannot be opened.

Invalid data format: Throw a custom exception named InvalidDataFormatException if a line in the file doesn't match the expected format.

Calculation errors: Throw a custom exception named CalculationError with a descriptive message if any calculation fails (e.g., division by zero).

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <stdexcept>

// Custom exceptions

class FileOpenError : public std::runtime\_error {

public:

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

};

class InvalidDataFormatException : public std::runtime\_error {

public:

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

};

class CalculationError : public std::runtime\_error {

public:

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

};

// Function to read data from file

std::vector<std::vector<double>> readData(const std::string& filename) {

std::ifstream file(filename);

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

throw FileOpenError("Failed to open the file: " + filename);

}

std::vector<std::vector<double>> data;

std::string line;

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

std::istringstream iss(line);

std::vector<double> values;

double value;

while (iss >> value) {

values.push\_back(value);

}

if (values.size() != 3) { // Expecting 3 values per line

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

}

data.push\_back(values);

}

return data;

}

// Function to perform calculations

void performCalculations(const std::vector<std::vector<double>>& data) {

for (const auto& values : data) {

double a = values[0];

double b = values[1];

double c = values[2];

if (b == 0) {

throw CalculationError("Division by zero error: b = 0");

}

double result = (a + c) / b;

std::cout << "Calculation result: " << result << std::endl;

}

}

int main() {

try {

std::vector<std::vector<double>> data = readData("data.txt");

performCalculations(data);

} catch (const FileOpenError& e) {

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

} catch (const InvalidDataFormatException& e) {

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

} catch (const CalculationError& e) {

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

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

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

}

return 0;

}