**Date:12/07/2024**

**Day\_15\_Assignments**

1.

#include<conio.h>

#include<fstream>

#include<iostream>

using namespace std;

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;

std::ofstream file;

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

file.write(reinterpret\_cast<char\*>(this), sizeof(student));

file.close();

//getch();

get\_data();

}

void student::get\_data()

{

int temp;

cout<<"enter roll no";

cin>>temp;

fstream file;

file.open("hit.txt",ios::in);

file.seekg(0,ios::beg);

while(file.read((char\*)this,sizeof(student)))

{

if(temp==s.roll)

{

cout<<"student name"<<s.name<<"\n";

cout<<"student roll"<<s.roll<<"\n";

}

}

}

int main()

{

//clrscr();

student st;

st.put\_data();

return 0;

}

2.

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

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";

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

readBinaryFile(binaryFilename, sizeof(binaryData));

return 0;

}

**3. 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.**

#include <iostream>

#include <fstream>

#include <string>

class Student {

public:

std::string name;

int id;

float marks;

void get\_data() {

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

std::cin >> name;

std::cout << "Enter student ID: ";

std::cin >> id;

std::cout << "Enter student marks: ";

std::cin >> marks;

}

void display\_data() const {

std::cout << "Name: " << name << "\nID: " << id << "\nMarks: " << marks << std::endl;

}

};

void add\_student() {

Student s;

s.get\_data();

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

if (file.is\_open()) {

file << s.name << " " << s.id << " " << s.marks << "\n";

file.close();

} else {

std::cerr << "Unable to open file" << std::endl;

}

}

void display\_students() {

std::ifstream file("students.txt");

if (file.is\_open()) {

Student s;

while (file >> s.name >> s.id >> s.marks) {

s.display\_data();

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

}

file.close();

} else {

std::cerr << "Unable to open file" << std::endl;

}

}

void search\_student() {

int search\_id;

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

std::cin >> search\_id;

std::ifstream file("students.txt");

if (file.is\_open()) {

Student s;

bool found = false;

while (file >> s.name >> s.id >> s.marks) {

if (s.id == search\_id) {

std::cout << "Student found:" << std::endl;

s.display\_data();

found = true;

break;

}

}

if (!found) {

std::cout << "Student with ID " << search\_id << " not found." << std::endl;

}

file.close();

} else {

std::cerr << "Unable to open file" << std::endl;

}

}

int main() {

int choice;

while (true) {

std::cout << "1. Add new student record\n2. Display all student records\n3. Search student by ID\n4. Exit\n";

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

std::cin >> choice;

switch (choice) {

case 1:

add\_student();

break;

case 2:

display\_students();

break;

case 3:

search\_student();

break;

case 4:

return 0;

default:

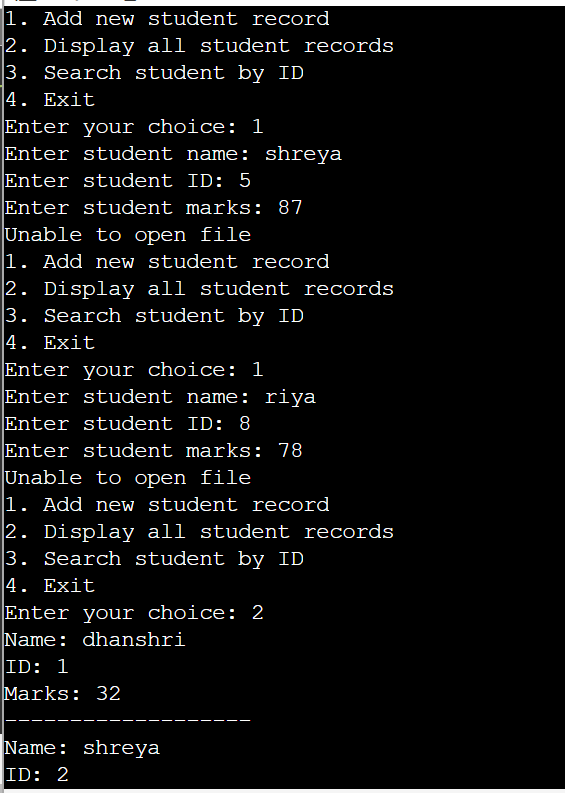
std::cout << "Invalid choice, please try again." << std::endl;

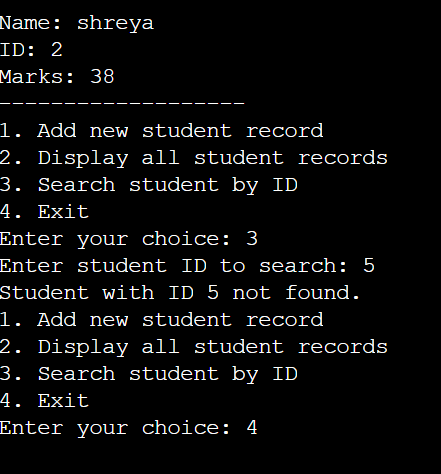
}

}

return 0;

}





**4.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.**

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

class Contact {

public:

string name;

string phone\_number;

void get\_data() {

cout << "Enter contact name: ";

cin >> name;

cout << "Enter phone number: ";

cin >> phone\_number;

}

void display\_data() const {

cout << "Name: " << name << "\nPhone Number: " << phone\_number << endl;

}

};

void add\_contact() {

Contact c;

c.get\_data();

ofstream file("phonebook.txt",ios::app);

if (file.is\_open()) {

file << c.name << " " << c.phone\_number << "\n";

file.close();

} else {

cerr << "Unable to open file" <<endl;

}

}

void search\_contact() {

string search\_name;

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

cin >> search\_name;

ifstream file("phonebook.txt");

if (file.is\_open()) {

Contact c;

bool found = false;

while (file >> c.name >> c.phone\_number) {

if (c.name == search\_name) {

cout << "Contact found:" <<endl;

c.display\_data();

found = true;

break;

}

}

if (!found) {

cout << "Contact with name " << search\_name << " not found." <<endl;

}

file.close();

} else {

cerr << "Unable to open file" <<endl;

}

}

int main() {

int choice;

while (true) {

cout << "1. Add new contact\n 2. Search contact by name\n 3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

add\_contact();

break;

case 2:

search\_contact();

break;

case 3:

return 0;

default:

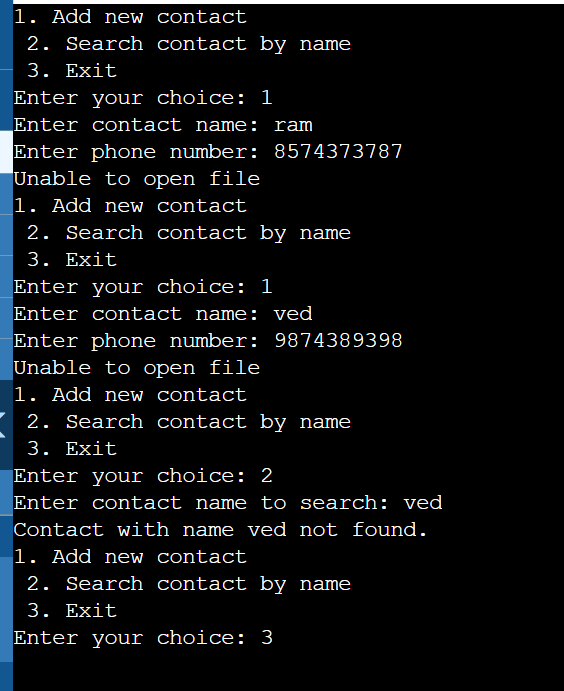
cout << "Invalid choice" << endl;

}

}

return 0;

}



**5. Add new items (name, price, quantity) to a binary file.Display all items from the inventory.Update the quantity of an existing item.**

#include <iostream>

#include <fstream>

#include <cstring>

using namespace std;

class Item {

public:

char name[50];

double price;

int quantity;

void get\_data() {

cout << "Enter item name: ";

cin >> name;

cout << "Enter item price: ";

cin >> price;

cout << "Enter item quantity: ";

cin >> quantity;

}

void display\_data() const {

std::cout << "Name: " << name << "\nPrice: " << price << "\nQuantity: " << quantity << std::endl;

}

};

void add\_item() {

Item item;

item.get\_data();

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

if (file.is\_open()) {

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

file.close();

} else {

cerr << "Unable to open file" << endl;

}

}

void display\_items() {

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

if (file.is\_open()) {

Item item;

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

item.display\_data();

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

}

file.close();

} else {

cerr << "Unable to open file" <<endl;

}

}

void update\_quantity() {

char search\_name[50];

int new\_quantity;

cout << "Enter item name to update quantity: ";

cin >> search\_name;

cout << "Enter new quantity: ";

cin >> new\_quantity;

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

if (file.is\_open()) {

Item item;

bool found = false;

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

if (strcmp(item.name, search\_name) == 0) {

item.quantity = new\_quantity;

int pos = static\_cast<int>(file.tellg()) - sizeof(Item);

file.seekp(pos);

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

found = true;

cout << "Quantity updated successfully." << endl;

break;

}

}

if (!found) {

cout << "Item with name " << search\_name << " not found." << endl;

}

file.close();

} else {

cerr << "Unable to open file" << endl;

}

}

int main() {

int choice;

while (true) {

cout << " 1. Add new item\n 2. Display all items\n 3. Update item quantity\n 4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

add\_item();

break;

case 2:

display\_items();

break;

case 3:

update\_quantity();

break;

case 4:

return 0;

default:

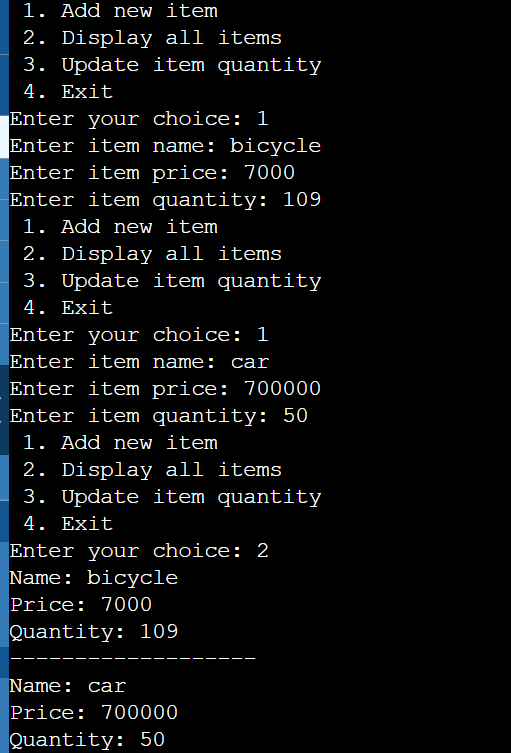
cout << "Invalid choice." <<endl;

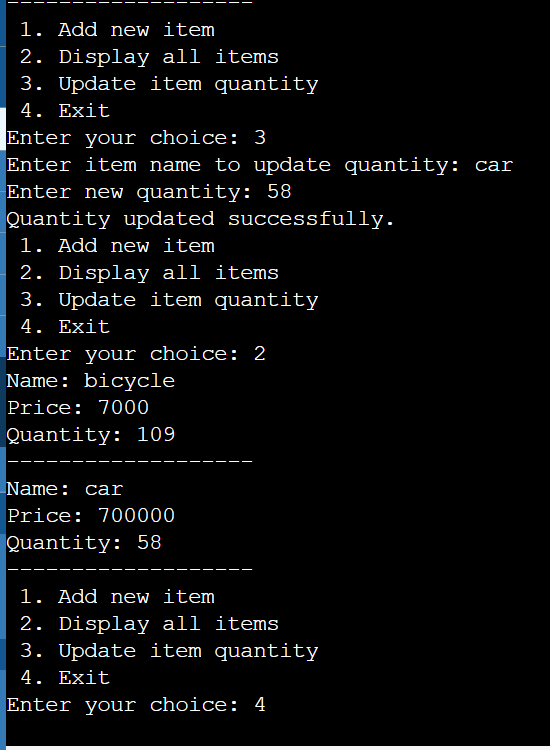
}

}

return 0;

}





**6. High Score Tracking (Optional): Create a program that keeps track of high scores for a game. Users can:Save a new high score to a binary file.Display the current high score.**

#include <iostream>

#include <fstream>

class HighScore {

public:

int score;

HighScore() : score(0) {}

void get\_data() {

std::cout << "Enter new high score: ";

std::cin >> score;

}

void display\_data() const {

std::cout << "Current high score: " << score << std::endl;

}

};

void save\_high\_score() {

HighScore hs;

hs.get\_data();

std::ofstream file("highscore.dat", std::ios::binary);

if (file.is\_open()) {

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

file.close();

std::cout << "High score saved successfully." << std::endl;

} else {

std::cerr << "Unable to open file" << std::endl;

}

}

void display\_high\_score() {

std::ifstream file("highscore.dat", std::ios::binary);

if (file.is\_open()) {

HighScore hs;

if (file.read(reinterpret\_cast<char\*>(&hs), sizeof(HighScore))) {

hs.display\_data();

} else {

std::cout << "No high score found." << std::endl;

}

file.close();

} else {

std::cerr << "Unable to open file" << std::endl;

}

}

int main() {

int choice;

while (true) {

std::cout << "1. Save new high score\n2. Display current high score\n3. Exit\n";

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

std::cin >> choice;

switch (choice) {

case 1:

save\_high\_score();

break;

case 2:

display\_high\_score();

break;

case 3:

return 0;

default:

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

}

}

return 0;

}

7.

#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;

}

8. #include <iostream>

using namespace std;

double add(double a, double b) {

return a + b;

}

double subtract(double a, double b) {

return a - b;

}

double multiply(double a, double b) {

return a \* b;

}

double divide(double a, double b) {

if (b == 0) {

throw runtime\_error("Error");

}

return a / b;

}

int main() {

double num1, num2;

char operation;

cout << "Enter first number: ";

cin >> num1;

cout << "Enter an operator ";

cin >> operation;

cout << "Enter second number: ";

cin >> num2;

try {

double result;

switch (operation) {

case '+':

result = add(num1, num2);

break;

case '-':

result = subtract(num1, num2);

break;

case '\*':

result = multiply(num1, num2);

break;

case '/':

result = divide(num1, num2);

break;

default:

std::cout << "Invalid operator." <<endl;

return 1;

}

cout << "Result="<< result <<endl;

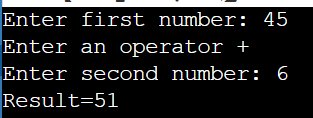
} catch (const runtime\_error &e) {

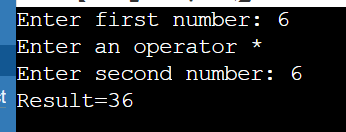
cout <<e.what()<< endl;

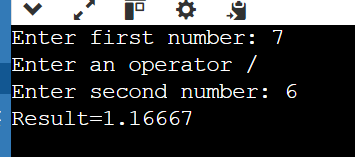
}

return 0;

}







9.

#include <iostream>

#include<exception>

using namespace std;

class Myexception:public exception

{

public:

const char\* what() const throw()

{

return "attempted to divide by zero";

}

};

int main()

{

try

{

int x,y;

cout<<"enter the two numbers:";

cin>>x>>y;

if(y==0)

{

Myexception z;

throw z;

}

else

{

cout<<"x/y="<<x/y<<endl;

}

}

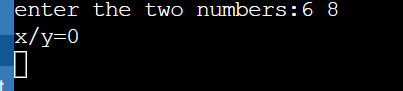
catch(exception& e)

{

cout<<e.what();

}

}



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

**Advantages of Using Exceptions**

1. **Separation of Error Handling and Business Logic**:
   * **Clarity**: Exceptions allow you to separate the error handling code from the main logic, making the code easier to read and maintain.
   * **Focus**: Developers can focus on the normal flow of the program without intertwining error handling logic, leading to cleaner and more understandable code.
2. **Automatic Propagation**:
   * **Simplified Error Propagation**: When an exception is thrown, it automatically propagates up the call stack until it is caught. This simplifies error handling, especially in deeply nested function calls.
   * **No Need for Explicit Return Values**: Functions don't need to return error codes, reducing the risk of forgetting to check for errors.
3. **Standardization**:
   * **Consistent Error Handling**: Using exceptions provides a consistent way to handle errors across the program.
   * **Rich Information**: Exceptions can carry detailed information about the error, which can be accessed through the exception object.
4. **Stack Unwinding**:
   * **Resource Management**: During stack unwinding, destructors for all objects created on the stack are called. This ensures proper cleanup of resources, reducing the chances of resource leaks.
5. **Error Handling Flexibility**:
   * **Granularity**: Different types of exceptions can be thrown and caught, allowing for more granular and specific error handling.
   * **Catching Multiple Exceptions**: You can catch multiple types of exceptions with different catch blocks, providing tailored responses to different error conditions.

**Disadvantages of Using Exceptions**

1. **Performance Overhead**:
   * **Runtime Cost**: Throwing and catching exceptions can be more computationally expensive compared to using traditional error codes.
   * **Code Size**: Exception handling can increase the size of the executable due to the additional code and data structures required.
2. **Complexity**:
   * **Learning Curve**: Exceptions introduce additional syntax and concepts that may be complex for new programmers to grasp.
   * **Misuse**: Incorrect use of exceptions (e.g., using them for control flow instead of error handling) can lead to convoluted and inefficient code.
3. **Non-Local Control Flow**:
   * **Debugging Difficulty**: Exceptions can make the control flow non-linear and harder to follow, complicating debugging and reasoning about the program.
   * **Hidden Errors**: If exceptions are not properly documented or understood, they can lead to unexpected behaviors.
4. **Compatibility Issues**:
   * **Interfacing with Other Languages**: Exceptions may pose challenges when interfacing with code written in other languages that do not support exceptions.
   * **Legacy Code**: Integrating exceptions into existing codebases that use traditional error codes can be problematic.
5. **Exception-Safe Code**:
   * **Additional Effort**: Writing exception-safe code requires careful design and often more boilerplate code (e.g., RAII for resource management).

**Comparison with Traditional Error Codes**

1. **Error Codes**:
   * **Advantages**:
     + **Predictability**: Control flow remains linear, making the program easier to follow.
     + **Performance**: Generally, error codes have less performance overhead.
     + **Simplicity**: Easier for beginners to understand and implement.
   * **Disadvantages**:
     + **Clutter**: Error handling code is mixed with business logic, making the codebase cluttered.
     + **Propagation**: Manually propagating error codes through multiple layers can be tedious and error-prone.
     + **Resource Leaks**: Without careful management, resource leaks are more likely as stack unwinding doesn’t automatically occur.

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

### 1. Use Descriptive Exception Messages

When you throw an exception, include a detailed message that describes the error. This message should provide context about what went wrong and, if possible, why it happened.

### 2. Create Custom Exception Classes

Define custom exception classes that inherit from standard exception classes (e.g., std::exception, std::runtime\_error). This allows you to add more specific information relevant to your application.

### 3. Include Contextual Information

When throwing exceptions, include contextual information such as function names, variable values, and the state of the application. This helps in understanding the exact conditions that led to the error.

### 4. Implement What() Method in Custom Exceptions

Override the what() method in your custom exception classes to return detailed error messages.

### 5. Log Exception Information

Log exception information to a file or a logging system. This ensures that detailed error information is available even after the application has terminated.

### 6. Use Exception Hierarchies

Define a hierarchy of exception classes to represent different types of errors. This helps in categorizing errors and providing specific information for each error type.

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

Optimizing exception handling performance is crucial in performance-critical applications where the overhead of exceptions can impact the overall efficiency. Here are several strategies to optimize exception handling performance:

**1. Avoid Exceptions for Control Flow**

**Strategy**: Use exceptions only for exceptional situations, not for regular control flow or expected conditions.

**Explanation**: Exceptions are designed to handle unexpected errors and not as a replacement for regular control flow mechanisms. Using exceptions for control flow can introduce unnecessary overhead.

**2. Minimize Exception Throwing**

**Strategy**: Design your code to avoid situations where exceptions are frequently thrown.

**Explanation**: Frequent throwing and catching of exceptions can degrade performance. Check conditions that might cause exceptions before performing operations that could trigger them.

**3. Use Lightweight Exception Objects**

**Strategy**: Use lightweight exception objects that have minimal impact on performance.

**Explanation**: Custom exceptions that inherit from std::exception should be as lightweight as possible. Avoid unnecessary data members or complex logic within the exception class.

**4. Leverage Static Exception Strings**

**Strategy**: Use static strings for exception messages to avoid the overhead of dynamic memory allocation.

**Explanation**: Creating and destroying dynamic strings can be expensive. Using static or constant strings reduces the overhead.

**5. Compile with Optimization Flags**

**Strategy**: Compile your code with optimization flags that optimize exception handling.

**Explanation**: Modern compilers provide optimization options that can improve the performance of exception handling.

**6. Use noexcept for Functions**

**Strategy**: Declare functions that do not throw exceptions as noexcept.

**Explanation**: The noexcept keyword informs the compiler that a function will not throw exceptions, allowing for potential optimizations and better code generation.

**7. Optimize Stack Unwinding**

**Strategy**: Minimize the cost of stack unwinding by ensuring that destructors and cleanup code are lightweight.

**Explanation**: During stack unwinding, destructors for all local objects are called. Ensure that these destructors are efficient and do not perform heavy operations.

**8. Profile and Benchmark**

**Strategy**: Profile and benchmark your application to identify hotspots and optimize accordingly.

**Explanation**: Use profiling tools to measure the impact of exception handling on performance. Optimize based on actual data rather than assumptions.

**Example Tools**:

* gprof
* Valgrind
* perf

**9. Avoid Throwing Exceptions in Destructors**

**Strategy**: Never throw exceptions from destructors.

**Explanation**: Throwing exceptions from destructors can lead to termination of the program if another exception is already active.

**10. Use std::optional and std::variant**

**Strategy**: Use std::optional or std::variant for functions that can return an error or a valid result.

**Explanation**: These types can represent the absence of a value or a valid result without using exceptions.

By implementing these strategies, you can optimize exception handling performance in your C++ applications, ensuring that exceptions are used effectively and efficiently without compromising the overall performance.

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

### 1. Define a Base Exception Class

Start with a base exception class that inherits from std::exception or one of its derived classes. This class will serve as the root of your exception hierarchy.

cpp

Copy code

#include <exception>

#include <string>

class BaseException : public std::exception {

protected:

std::string message;

public:

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

const char\* what() const noexcept override {

return message.c\_str();

}

};

### 2. Create Specific Derived Exception Classes

Create derived exception classes for different categories of errors. These classes should inherit from the base exception class.

* **IO Exceptions**: For input/output related errors.
* **Network Exceptions**: For network-related errors.
* **Database Exceptions**: For database-related errors.

### 3. Further Specialize Exceptions

Create more specialized exceptions for specific error conditions within each category. This allows for more granular error handling.

### 4. Use the Exception Hierarchy

Throw and catch exceptions based on their specific types. This allows you to handle errors at different levels of granularity.

### 5. Document Your Exceptions

Provide clear documentation for each exception class, explaining when it should be thrown and what information it contains. This helps maintainers understand the purpose and usage of each exception.

### 6. Consider Adding Error Codes

For more structured error handling, consider adding error codes to your exceptions. This allows for programmatic differentiation between error types without relying solely on exception types or messages.

### 7. Use noexcept Where Appropriate

Use the noexcept specifier for functions that are guaranteed not to throw exceptions. This can help with performance and clarity.

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

While exceptions provide a powerful mechanism for error handling in C++, there are scenarios where using exceptions may not be appropriate or beneficial. Here are some situations where you might choose not to use exceptions:

### 1. Embedded Systems and Real-Time Applications

In embedded systems or real-time applications where deterministic behavior and strict timing constraints are critical, the overhead introduced by exceptions (both in terms of performance and code size) might be unacceptable. Exceptions involve stack unwinding and dynamic memory allocation, which can be unpredictable in these contexts.

### 2. Performance-Critical Applications

Applications that prioritize performance over ease of error handling might opt for error codes or alternative mechanisms. Exceptions can introduce overhead due to stack unwinding and the associated cleanup process, impacting performance-sensitive operations.

### 3. Existing Codebases Using Error Codes

If you are working with an existing codebase that extensively uses error codes for error handling, introducing exceptions can lead to inconsistency and complexity. It might be more practical to continue using error codes to maintain compatibility and avoid refactoring overhead.

### 4. Code Size Constraints

In environments where minimizing executable size is crucial (e.g., embedded systems, limited memory environments), exceptions can significantly increase code size due to the additional exception handling mechanisms required by the compiler.

### 5. Compatibility with External Libraries or APIs

Some libraries or APIs may not support or handle exceptions well. Introducing exceptions into such contexts can lead to interoperability issues or unexpected behaviors. In such cases, sticking to error codes or other agreed-upon error handling mechanisms may be more appropriate.

### 6. Team Familiarity and Skill Level

If the development team is not experienced with exceptions or lacks the necessary understanding to use them effectively, it might be prudent to stick with simpler error handling mechanisms like error codes or status flags. This reduces the risk of misuse or improper handling of exceptions.

### 7. Predictable Control Flow

In some cases, using exceptions can obscure the flow of control in the code, making it harder to follow and reason about. Error codes or explicit error-checking conditions can provide a more straightforward and predictable control flow, enhancing code readability and maintainability.

**15. 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).**

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <stdexcept>

void readDataFromFile(const std::string& filename) {

std::ifstream file(filename);

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

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

}

std::string line;

int lineCount = 0;

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

try {

std::istringstream iss(line);

std::string value1, value2;

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

throw std::invalid\_argument("Invalid format in line " + std::to\_string(lineCount + 1));

}

// Example validation: Check if values are integers (optional)

// int num1 = std::stoi(value1);

// int num2 = std::stoi(value2);

// Process valid data (e.g., store in a vector, print, etc.)

std::cout << "Read from file: " << value1 << " " << value2 << std::endl;

} catch (const std::invalid\_argument& e) {

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

// Handle format error (e.g., log, skip line, etc.)

}

lineCount++;

}

file.close();

}

int main() {

std::string filename = "data.txt"; // Replace with your file name

try {

readDataFromFile(filename);

} catch (const std::runtime\_error& e) {

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

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

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

}

return 0;

}