#include <iostream>

#include<conio.h>

#include<fstream>

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;

fstream file;

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

file.write((char\*)&s,sizeof(s));

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\*)&s,sizeof(s))) {

if(temp==s.roll) {

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

cout<<"student roll: "<<s.roll;

}

}

file.close();

}

int main() {

//clrscr();

student st;

st.put\_data();

}

OUTPUT:  
Enter name: mouni

Enter roll: 202

Enter roll no.: 202

Student name: mouni

Student roll: 202

**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"; }

}

void addContact() { // Function to add a new contact

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";

}

void searchContact() { // Function to search for a contact by name

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"; }

}

void encryptFile(const string &fileName, int key) { // Function to encrypt a file

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";

}

void decryptFile(const string &fileName, int key) { // Function to decrypt a file using Caesar cipher

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";

}

OUTPUT:

Menu:

1. Add Student Record

2. Display All Student Records

3. Search Student by ID

4. Add Contact

5. Search Contact by Name

6. Encrypt File

7. Decrypt File

8. Exit

Enter your choice: 1

Enter student name: mounika

Enter student ID: 202

Enter student marks: 79

Student record added successfully.

**Binary Files:**

**Image Copy: Write a program that copies the contents of an image file (e.g., JPG, PNG) to a new file. Ensure you handle binary data correctly.**

**Inventory Management: Develop a program that manages a store inventory. Users can:**

**Add new items (name, price, quantity) to a binary file.**

**Display all items from the inventory.**

**Update the quantity of an existing item.**

**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>

#include <cstring>

using namespace std;

// Function to copy an image file

void copyImage(const string& source, const string& destination) {

ifstream src(source, ios::binary);

ofstream dest(destination, ios::binary);

if (!src || !dest) {

cerr << "Error opening file!" << endl;

return;

}

dest << src.rdbuf();

cout << "Image copied successfully!" << endl;

}

// Inventory Management

struct Item {

char name[50];

double price;

int quantity;

};

void addItem() {

ofstream file("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;

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

cout << "Item added successfully!" << endl;

}

void displayItems() {

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

Item item;

cout << "Inventory Items:\n";

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

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

}

}

void updateItemQuantity() {

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

char name[50];

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

cin.ignore();

cin.getline(name, 50);

Item item;

bool found = false;

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

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

cout << "Enter new quantity: ";

cin >> item.quantity;

file.seekp(-static\_cast<int>(sizeof(Item)), ios::cur);

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

cout << "Item quantity updated successfully!" << endl;

found = true;

break;

}

}

if (!found) cout << "Item not found." << endl;

}

// High Score Tracking

struct HighScore {

char name[50];

int score;

};

void saveHighScore() {

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

HighScore highScore;

cout << "Enter player name: ";

cin.ignore();

cin.getline(highScore.name, 50);

cout << "Enter player score: ";

cin >> highScore.score;

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

cout << "High score saved successfully!" << endl;

}

void displayHighScore() {

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

HighScore highScore;

int highestScore = 0;

string highestScorer;

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

if (highScore.score > highestScore) {

highestScore = highScore.score;

highestScorer = highScore.name;

}

}

if (highestScore > 0) {

cout << "Highest Score: " << highestScore << " by " << highestScorer << endl;

} else {

cout << "No high scores found." << endl;

}

}

int main() {

while (true) {

cout << "Enter:\n1 to copy an image\n2 to add inventory item\n3 to display inventory\n4 to update item quantity\n5 to save high score\n6 to display high score\n0 to exit\nChoice: ";

int choice;

cin >> choice;

switch (choice) {

case 1: {

string src, dest;

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

cin >> src;

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

cin >> dest;

copyImage(src, dest);

break;

}

case 2: addItem(); break;

case 3: displayItems(); break;

case 4: updateItemQuantity(); break;

case 5: saveHighScore(); break;

case 6: displayHighScore(); break;

case 0: cout << "Exiting program." << endl; return 0;

default: cout << "Invalid choice. Try again." << endl;

}

}

}

**EXCEPTION HANDLING**

Exception Handling in C++ is a process to handle runtime errors. We perform

exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

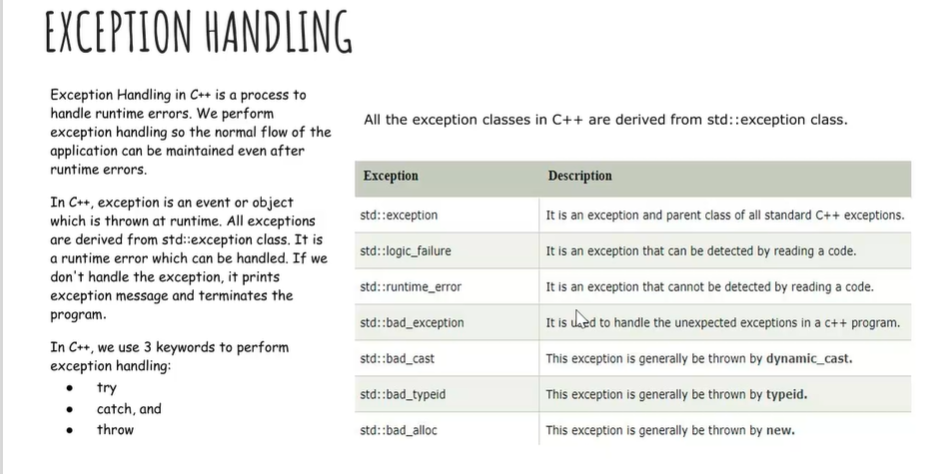
In C++, we use 3 keywords to perform exception handling:

⚫ try

catch, and

throw

All the exception classes in C++ are derived from std::exception class.



TRY/CATCH

#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;

}

OUTPUT:

Attempted to divide by zero!

**code for every calculation like addition subtraction multiplication and division using try and catch exception**

#include <iostream>

using namespace std;

float addition(float x, float y){

return x + y;

}

float subtraction(float x, float y){

return x - y;

}

float multiplication(float x, float y){

return x \* y;

}

float division(float x, float y){

if(y == 0){

throw "Attempted to divide by zero!";

}

return x / y;

}

int main(){

float num1, num2;

string operation;

cout << "Enter a number: ";

cin >> num1;

cout << "Enter an operation (+, -, \*, /): ";

cin >> operation;

cout << "Enter another number: ";

cin >> num2;

try{

if(operation == "+"){

cout << "Result: " << addition(num1, num2) << endl;

} else if(operation == "-"){

cout << "Result: " << subtraction(num1, num2) << endl;

} else if(operation == "\*"){

cout << "Result: " << multiplication(num1, num2) << endl;

} else if(operation == "/"){

cout << "Result: " << division(num1, num2) << endl;

} else {

cout << "Invalid operation!" << endl;

}

} catch(const char\* e){

cerr << e << endl;

}

return 0;

}

OUTPUT:  
Enter a number: 2

Enter an operation (+, -, \*, /): +

Enter another number: 3

Result: 5

**Advantages and Disadvantages of Using Exceptions in C++ Compared to Traditional Error Codes**

Advantages:

Separation of Error-Handling Code: Exceptions allow the separation of error-handling code from the main logic, making the code cleaner and more readable.

Automatic Cleanup: When an exception is thrown, destructors for objects created with automatic storage duration are called automatically, helping to prevent resource leaks.

Stack Unwinding: The call stack is unwound automatically when an exception is thrown, ensuring that destructors are called and resources are released.

Uniform Error Handling: Exceptions provide a consistent way to handle errors, whereas error codes can be inconsistent and easily missed.

Error Propagation: Exceptions propagate errors up the call stack, allowing higher-level functions to handle errors without having to check error codes after each function call.

Disadvantages:

Performance Overhead: Exception handling can introduce performance overhead, especially in terms of stack unwinding and catching exceptions.

Complexity: Exceptions can make the control flow of a program more complex, particularly if not used carefully.

Non-Local Control Flow: Exceptions introduce non-local control flow, which can make debugging and reasoning about the program more difficult.

Not Always Intuitive: Developers accustomed to traditional error codes might find exceptions less intuitive and harder to adopt.

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

To ensure that exception classes provide informative error messages:

Custom Exception Classes: Derive custom exception classes from std::exception and override the what() method to return detailed error messages.

Error Codes and Descriptions: Include error codes and descriptive messages as members of your exception classes.

Context Information: Provide context information such as the function name, file name, and line number where the exception was thrown.

Stack Trace: In some environments, capturing and displaying a stack trace can provide valuable debugging information.

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

Optimizing Exception Handling Performance,Strategies for Optimizing Performance:

Use Sparingly: Use exceptions for exceptional conditions, not for regular control flow.

Narrow Try Blocks: Minimize the code within try blocks to reduce the performance impact of potential exceptions.

Precondition Checks: Validate preconditions and use assertions to catch errors early, before they trigger exceptions.

Avoid Exceptions in Performance-Critical Paths: In performance-critical sections, prefer error codes or other mechanisms if exceptions introduce significant overhead.

Compile-Time Optimizations: Modern compilers optimize exception handling, but profile your application to ensure that exception handling does not introduce unacceptable overhead.

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

Base Exception Class: Create a base exception class from which all other exceptions will inherit.

Categorize Exceptions: Group exceptions into logical categories (e.g., I/O errors, network errors) and create derived classes for each category.

Granularity: Provide fine-grained exceptions for specific errors, but avoid an overly complex hierarchy.

Common Interface: Ensure all exceptions provide a consistent interface for accessing error information.

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

Appropriate Scenarios:

Performance-Critical Code: In high-performance or real-time systems where the overhead of exception handling is prohibitive.

Low-Level Libraries: In low-level libraries where exceptions could interfere with other error-handling mechanisms or where the client code prefers error codes.

Embedded Systems: In embedded systems with limited resources, where the cost of exception handling might be too high.

Simple Functions: For simple functions where error codes are sufficient and easier to manage.

Consistent Error Handling: When integrating with legacy code or libraries that use error codes, maintaining consistency might be preferable.

Reasoning:

Exceptions can add complexity and overhead, which might not be acceptable in all contexts.

Error codes can be more predictable and easier to control in certain scenarios.

In some environments, the cost of exceptions in terms of performance and memory usage can be prohibitive.

**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 <sstream>

#include <vector>

#include <string>

#include <iterator>

#include <exception>

using namespace std;

class FileOpenError : public exception {

public:

const char\* what() const noexcept override {

return "Error: Unable to open file.";

}

};

class InvalidDataFormatException : public exception {

string message;

public:

InvalidDataFormatException(const string& line) {

message = "Error: Invalid data format in line: " + line;

}

const char\* what() const noexcept override {

return message.c\_str();

}

};

class CalculationError : public exception {

string message;

public:

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

const char\* what() const noexcept override {

return message.c\_str();

}

};

vector<vector<int>> readData(istream& file) {

vector<vector<int>> data;

string line;

while (getline(file, line)) {

istringstream iss(line);

vector<int> values((istream\_iterator<int>(iss)), istream\_iterator<int>());

if (values.size() != 3) throw InvalidDataFormatException(line);

data.push\_back(values);

}

return data;

}

double performCalculations(const vector<vector<int>>& data) {

double result = 0.0;

for (const auto& values : data) {

if (values[2] == 0) throw CalculationError("Error: Division by zero.");

result += static\_cast<double>(values[0] + values[1]) / values[2];

}

return result;

}

int main() {

try {

// Simulating file content using stringstream

string fileContent = "10 20 5\n15 30 3\n25 35 7\n";

istringstream file(fileContent);

vector<vector<int>> data = readData(file);

double result = performCalculations(data);

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

} catch (const exception& e) {

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

}

return 0;

}

OUTPUT:

Calculation result: 29.5714