Day-15:

#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() {

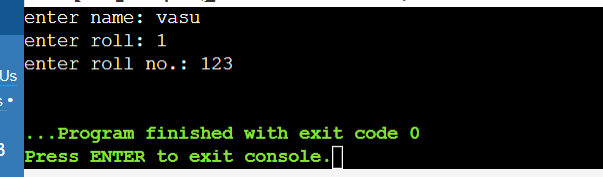
//clrscr();

student st;

st.put\_data();

}

Output:



#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();

}

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();

}

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();

}

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();

}

int main() {

string textFilename="example.text";

string binaryFilename="data.bin";

createTextFile(textFilename);

readTextFile(textFilename);

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

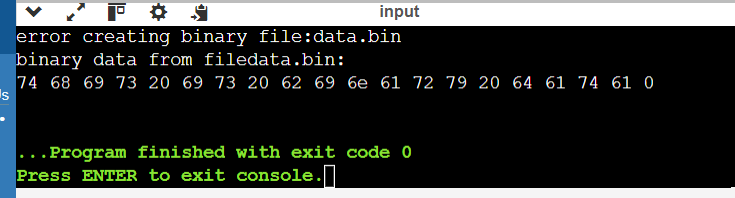
writeBinaryFile(binaryFilename,binaryData,sizeof(binaryData));

readBinaryFile(binaryFilename,sizeof(binaryData));

return 0;

}

Output:



3File Handling Practice Problems

This set of problems will help you practice the concepts of file handling in C++ covered in the provided code.

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>

using namespace std;

// Structure to hold student information

struct Student {

string name;

int id;

float marks;

};

// Function to add a new student record to the file

void addStudentRecord(const string& filename) {

ofstream outfile(filename, ios::app); // Append mode to add new records

if (!outfile) {

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

return;

}

Student student;

// Input student details

cout << "Enter student name: ";

getline(cin >> ws, student.name); // Using getline to handle spaces in names

cout << "Enter student ID: ";

cin >> student.id;

cout << "Enter student marks: ";

cin >> student.marks;

// Write student details to the file

outfile << student.name << " " << student.id << " " << student.marks << endl;

cout << "Student record added successfully!" << endl;

outfile.close();

}

// Function to display all student records from the file

void displayAllStudentRecords(const string& filename) {

ifstream infile(filename);

if (!infile) {

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

return;

}

Student student;

cout << "Student records:" << endl;

// Read and display each student record

while (infile >> student.name >> student.id >> student.marks) {

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

}

infile.close();

}

// Function to search for a specific student by ID and display details

void searchStudentByID(const string& filename, int searchID) {

ifstream infile(filename);

if (!infile) {

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

return;

}

Student student;

bool found = false;

// Search for the student by ID

while (infile >> student.name >> student.id >> student.marks) {

if (student.id == searchID) {

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

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

found = true;

break;

}

}

if (!found) {

cout << "Student with ID " << searchID << " not found." << endl;

}

infile.close();

}

int main() {

string filename = "student\_records.txt";

int choice;

int searchID;

do {

cout << "\n----- Menu -----" << endl;

cout << "1. Add a new student record" << endl;

cout << "2. Display all student records" << endl;

cout << "3. Search for a student by ID" << endl;

cout << "4. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cin.ignore(); // Clear input buffer

addStudentRecord(filename);

break;

case 2:

displayAllStudentRecords(filename);

break;

case 3:

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

cin >> searchID;

searchStudentByID(filename, searchID);

break;

case 4:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice. Please enter a number from 1 to 4." << endl;

break;

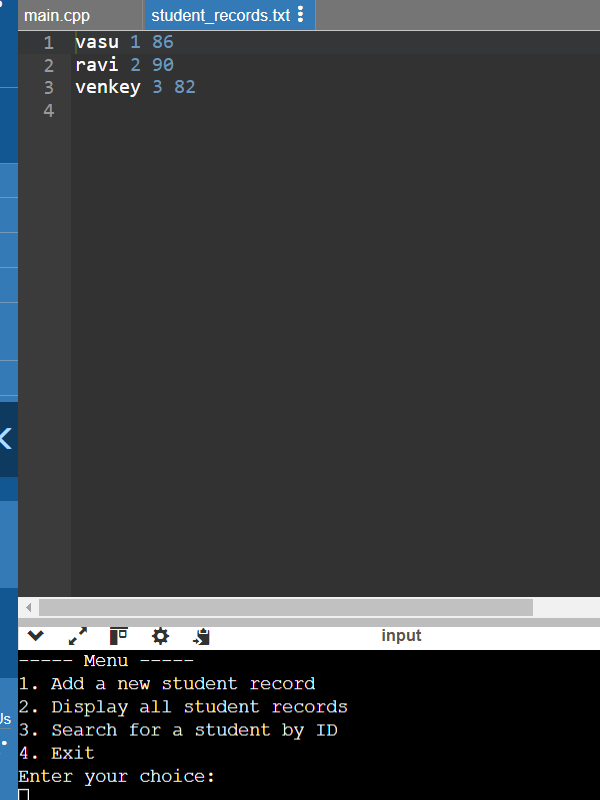
}

} while (choice != 4);

return 0;

}

Output:



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 to add a new contact to the phonebook file

void addContact(const string& filename) {

ofstream outfile(filename, ios::app); // Append mode to add new contacts

if (!outfile) {

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

return;

}

string name, phoneNumber;

// Input contact details

cout << "Enter contact name: ";

getline(cin >> ws, name); // Using getline to handle spaces in names

cout << "Enter contact phone number: ";

getline(cin, phoneNumber);

// Write contact details to the file

outfile << name << " " << phoneNumber << endl;

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

outfile.close();

}

// Function to search for a contact by name and display phone number

void searchContactByName(const string& filename, const string& searchName) {

ifstream infile(filename);

if (!infile) {

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

return;

}

string name, phoneNumber;

bool found = false;

// Search for the contact by name

while (infile >> name >> phoneNumber) {

if (name == searchName) {

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

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

found = true;

break;

}

}

if (!found) {

cout << "Contact with name '" << searchName << "' not found." << endl;

}

infile.close();

}

int main() {

string filename = "phonebook.txt";

int choice;

string searchName;

do {

cout << "\n----- Menu -----" << endl;

cout << "1. Add a new contact" << endl;

cout << "2. Search for a contact by name" << endl;

cout << "3. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

cin.ignore(); // Clear input buffer

switch (choice) {

case 1:

addContact(filename);

break;

case 2:

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

getline(cin >> ws, searchName

searchContactByName(filename, searchName);

break;

case 3:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice. Please enter 1, 2, or 3." << endl;

break;

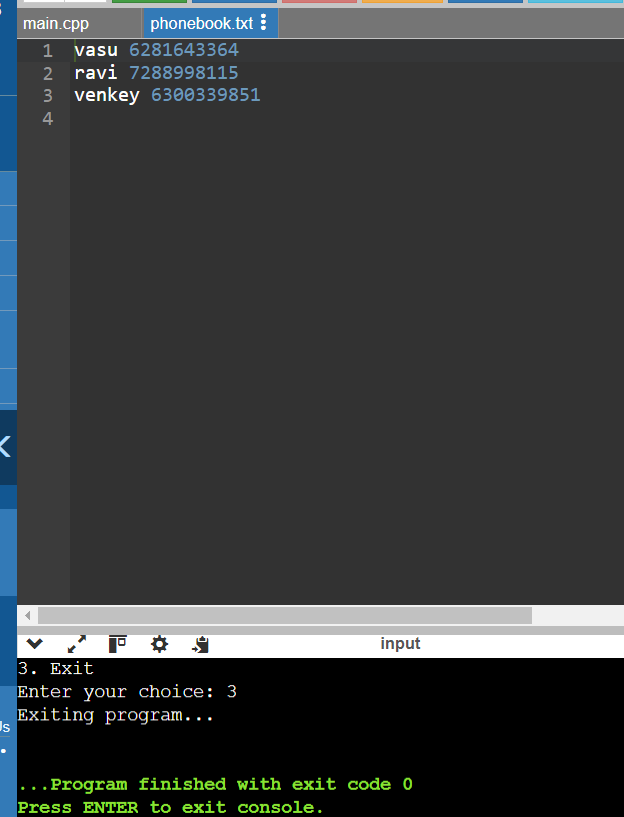
}

} while (choice != 3);

return 0;

}

Output:



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 <string>

#include <vector>

using namespace std;

// Structure for item in inventory

struct Item {

string name;

double price;

int quantity;

};

// Structure for high score record

struct HighScore {

string playerName;

int score;

};

// Function to copy contents of an image file to another file

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

ifstream infile(sourceFile, ios::binary);

ofstream outfile(destFile, ios::binary);

if (!infile) {

cerr << "Error opening source file: " << sourceFile << endl;

return;

}

if (!outfile) {

cerr << "Error opening destination file: " << destFile << endl;

infile.close();

return;

}

char buffer[1024];

while (infile.read(buffer, sizeof(buffer))) {

outfile.write(buffer, sizeof(buffer));

}

outfile.write(buffer, infile.gcount());

cout << "Image copied successfully from " << sourceFile << " to " << destFile << endl;

infile.close();

outfile.close();

}

// Function to add a new item to the inventory file

void addItem(const string& filename) {

ofstream outfile(filename, ios::binary | ios::app);

if (!outfile) {

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

return;

}

Item item;

cout << "Enter item name: ";

getline(cin >> ws, item.name);

cout << "Enter item price: ";

cin >> item.price;

cout << "Enter item quantity: ";

cin >> item.quantity;

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

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

outfile.close();

}

// Function to display all items from the inventory file

void displayInventory(const string& filename) {

ifstream infile(filename, ios::binary);

if (!infile) {

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

return;

}

Item item;

cout << "Inventory items:" << endl;

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 in the inventory file

void updateItemQuantity(const string& filename, const string& itemName, int newQuantity) {

fstream file(filename, ios::binary | ios::in | ios::out);

if (!file) {

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

return;

}

Item item;

bool found = false;

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

if (item.name == itemName) {

item.quantity = newQuantity;

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

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

found = true;

}

}

if (found) {

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

} else {

cout << "Item not found in inventory." << endl;

}

file.close();

}

// Function to save a new high score to the file

void saveHighScore(const string& filename, const string& playerName, int score) {

ofstream outfile(filename, ios::binary);

if (!outfile) {

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

return;

}

HighScore newScore = { playerName, score };

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

cout << "New high score saved successfully!" << endl;

outfile.close();

}

// Function to display the current high score from the file

void displayHighScore(const string& filename) {

ifstream infile(filename, ios::binary);

if (!infile) {

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

return;

}

HighScore currentScore;

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

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

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

} else {

cout << "No high score recorded yet." << endl;

}

infile.close();

}

int main() {

string inventoryFilename = "inventory.dat";

string highScoresFilename = "high\_scores.dat";

string sourceImage = "source\_image.jpg";

string copiedImage = "copied\_image.jpg";

int choice;

string itemName;

int newQuantity;

string playerName;

int score;

do {

cout << "\n----- Menu -----" << endl;

cout << "1. Copy Image" << endl;

cout << "2. Add a new item to Inventory" << endl;

cout << "3. Display all Inventory items" << endl;

cout << "4. Update item quantity" << endl;

cout << "5. Save new high score" << endl;

cout << "6. Display current high score" << endl;

cout << "7. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

cin.ignore();

switch (choice) {

case 1:

copyImage(sourceImage, copiedImage);

break;

case 2:

addItem(inventoryFilename);

break;

case 3:

displayInventory(inventoryFilename);

break;

case 4:

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

getline(cin >> ws, itemName);

cout << "Enter new quantity: ";

cin >> newQuantity;

updateItemQuantity(inventoryFilename, itemName, newQuantity);

break;

case 5:

cout << "Enter player name: ";

getline(cin >> ws, playerName);

cout << "Enter score: ";

cin >> score;

saveHighScore(highScoresFilename, playerName, score);

break;

case 6:

displayHighScore(highScoresFilename);

break;

case 7:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice. Please enter a number between 1 and 7." << endl;

break;

}

} while (choice != 7);

return 0;

}

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){

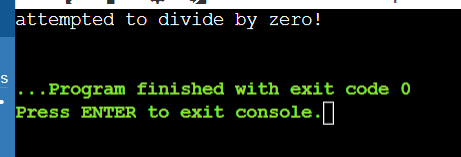
cerr<<e<<endl;

}

return 0;

}

Output:



#include<iostream>

#include<exception>

using namespace std;

class MyException:public exception{

public:

const char\*what()const throw()

{

return"Attempted to divide by zero!\n";

}

};

int main(){

try{

int x,y;

cout<<"enter the two number:\n";

cin>>x>>y;

if(y==0)

{

MyException z;

throw z;

}

else

{

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

}

}

catch(exception&e)

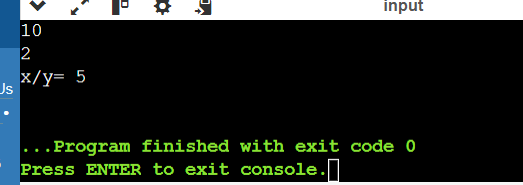
{

cout<<e.what();

}

}

Output:



Questions:

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

Advantages of Exceptions in C++:

* **Simplicity**: Exceptions simplify error handling by separating normal code flow from error-handling logic, making code cleaner and easier to read.
* **Hierarchical Handling**: Exceptions support hierarchical error handling through try, catch, and throw, allowing different types of errors to be handled differently.
* **Focus on Normal Flow**: Code can focus on its primary task without being cluttered by error-checking code at every step.

Disadvantages of Exceptions in C++:

* **Performance Overhead**: Exception handling may introduce performance overhead, especially when exceptions are thrown frequently, due to stack unwinding and catch block searching.
* **Complex Resource Management**: Managing resources (like memory or file handles) can be more complex with exceptions, requiring careful use of RAII (Resource Acquisition Is Initialization) to ensure proper cleanup.
* **Compatibility Issues**: Exceptions may not be universally supported across all platforms or libraries, limiting their usage in certain environments.

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 Exceptions Sparingly**: Reserve exceptions for truly exceptional errors, not for regular flow control.

 **Catch Specific Exceptions**: Handle specific exception types to avoid unnecessary checks and improve efficiency.

 **Minimize Stack Unwinding**: Keep exception throwing within shallow call stacks to reduce overhead during stack unwinding.

 **Avoid Complex Resource Management**: Limit resource management tasks within exception handling to avoid performance bottlenecks.

 **Profile and Benchmark**: Measure the impact of exceptions using profiling tools to identify and optimize performance hotspots.

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:

