PROGRAMS

DATE:19/03/24

1. Define a base class **Shape** with a virtual function **draw()**. Derive classes **Circle** and **Rectangle** from **Shape**. Implement the **draw()** function in each derived class to draw the respective shape using cout statements.

PROGRAM:

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() {

cout << "Drawing a shape..." << endl;

}

};

class Square : public Shape {

public:

void draw() override {

cout << "Drawing a square..." << endl;

}

};

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing a circle..." << endl;

}

};

class Triangle : public Shape {

public:

void draw() override {

cout << "Drawing a triangle..." << endl;

}

};

int main() {

Square;

Circle;

Triangle;

square.draw();

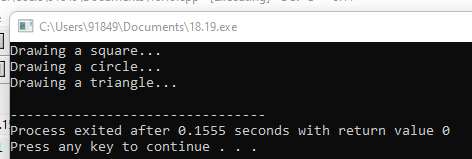
circle.draw();

triangle.draw();

return 0;

}

OUTPUT:



2. Create an abstract base class **Animal** with a pure virtual function **makeSound()**. Derive classes **Dog** and **Cat** from **Animal**. Implement the **makeSound()** function in each derived class to output appropriate sounds for a dog and a cat.

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() const {

cout << "Some generic sound from an animal" <<endl;

}

};

class Dog : public Animal {

public:

void makeSound() const override {

cout << "Woof! Woof!" <<endl;

}

};

class Cat : public Animal {

public:

void makeSound() const override {

cout << "Meow! Meow!" <<endl;

}

};

int main() {

Animal \*animal1 = new Dog();

Animal \*animal2 = new Cat();

animal1->makeSound();

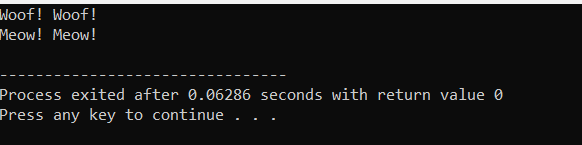
animal2->makeSound();

delete animal1;

delete animal2;

return 0;

}



3. Define a base class **Employee** with a virtual function **displayInfo()**. Derive classes **Manager** and **Clerk** from **Employee**. Implement the **displayInfo()** function in each derived class to display the details of the employee.

#include <iostream>

#include <string>

using namespace std;

class Employee {

protected:

string name;

int employeeID;

public:

Employee(string name, int employeeID) : name(name), employeeID(employeeID) {}

virtual void displayInfo() {

cout << "Name: " << name << endl;

cout << "Employee ID: " << employeeID << endl;

}

};

class Manager : public Employee {

string department;

public:

Manager(string name, int employeeID, string department) : Employee(name, employeeID), department(department) {}

void displayInfo() override {

cout << "Name: " << name << endl;

cout << "Employee ID: " << employeeID << endl;

cout << "Department: " << department << endl;

}

};

class Clerk : public Employee {

int experience;

public:

Clerk(string name, int employeeID, int experience) : Employee(name, employeeID), experience(experience) {}

void displayInfo() override {

cout << "Name: " << name << endl;

cout << "Employee ID: " << employeeID << endl;

cout << "Experience: " << experience << " years" << endl;

}

};

int main() {

Manager manager("John Doe", 1001, "Sales");

Clerk clerk("Alice Smith", 2001, 3);

cout << "Manager's Info:" << endl;

manager.displayInfo();

cout << endl;

cout << "Clerk's Info:" << endl;

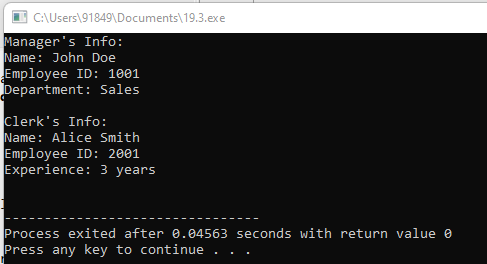
clerk.displayInfo();

cout << endl;

return 0;

}

OUTPUT:



4. Define an abstract base class **BankAccount** with a pure virtual function **withdraw()**. Derive classes **SavingsAccount** and **CurrentAccount** from **BankAccount**. Implement the **withdraw()** function in each derived class to perform withdrawal operations.

#include <iostream>

using namespace std;

class BankAccount {

protected:

double balance;

public:

BankAccount(double initialBalance) : balance(initialBalance) {}

virtual void withdraw(double amount) = 0;

};

class SavingsAccount : public BankAccount {

double interestRate;

public:

SavingsAccount(double initialBalance, double rate) : BankAccount(initialBalance), interestRate(rate) {}

void withdraw(double amount) override {

if (amount > balance) {

cout << "Insufficient funds!" << endl;

} else {

balance -= amount;

cout << "Withdrawal of $" << amount << " from Savings Account. Current balance: $" << balance << endl;

}

}

};

class CurrentAccount : public BankAccount {

double overdraftLimit;

public:

CurrentAccount(double initialBalance, double limit) : BankAccount(initialBalance), overdraftLimit(limit) {}

void withdraw(double amount) override {

if (amount > balance + overdraftLimit) {

cout << "Withdrawal exceeds overdraft limit!" << endl;

} else {

balance -= amount;

cout << "Withdrawal of $" << amount << " from Current Account. Current balance: $" << balance << endl;

}

}

};

int main() {

SavingsAccount savings(1000, 0.02);

CurrentAccount current(500, 200);

savings.withdraw(200);

savings.withdraw(1500);

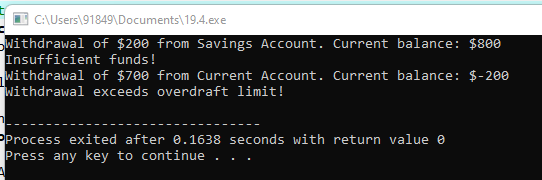
current.withdraw(700);

current.withdraw(800);

return 0;

}

OUTPUT:



5. Write a C++ program to demonstrate the usage of try-catch blocks for handling exceptions when dividing two numbers. Handle the division by zero exception.

#include <iostream>

#include <stdexcept>

using namespace std;

float Division(float num, float den)

{

if (den == 0) {

throw runtime\_error("Math error: Attempted to divide by Zero\n");

}

return (num / den);

}

int main()

{

float numerator, denominator, result;

numerator = 12.5;

denominator = 0;

try {

result = Division(numerator, denominator);

cout << "The quotient is "

<< result << endl;

}

catch (runtime\_error& e) {

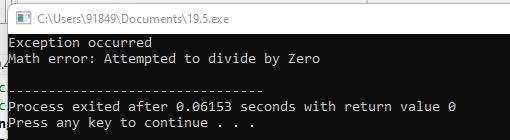
cout << "Exception occurred" << endl

<< e.what();

}

}

OUTPUT:



6. Create a function int calculateAverage(int array[], int size) that calculates the average of elements in an integer array. Handle the case where the array size is zero by throwing an appropriate exception.

#include <iostream>

#include <stdexcept>

using namespace std;

double calculateAverage(int array[], int size) {

if (size == 0) {

throw invalid\_argument("Array size is zero.");

}

double sum = 0;

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

sum += array[i];

}

return sum / size;

}

int main() {

int array1[] = {1, 2, 3, 4, 5};

int size1 = sizeof(array1) / sizeof(array1[0]);

try {

double avg1 = calculateAverage(array1, size1);

cout << "Average of array1: " << avg1 << endl;

} catch (const invalid\_argument& e) {

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

}

int array2[] = {};

int size2 = sizeof(array2) / sizeof(array2[0]);

try {

double avg2 = calculateAverage(array2, size2);

cout << "Average of array2: " << avg2 << endl;

} catch (const invalid\_argument& e) {

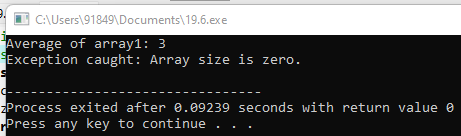
cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

OUTPUT:



7. Write a C++ program that asks the user to input their age. If the age entered is less than 0 or greater than 150, throw an exception. Otherwise, print the age.

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

int age;

cout << "Enter your age: ";

cin >> age;

if (age < 0 || age > 150) {

throw out\_of\_range("Age must be between 0 and 150.");

}

cout << "Your age is: " << age << endl;

} catch (const out\_of\_range& e) {

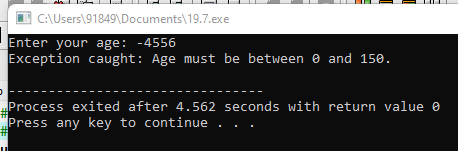
cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

OUTPUT:



8. Write a C++ program that reads an integer from the user. If the integer is not in the range [1, 100], throw an exception. Otherwise, print the integer.

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

int num;

cout << "Enter an integer between 1 and 100: ";

cin >> num;

if (num < 1 || num > 100) {

throw out\_of\_range("Integer must be between 1 and 100.");

}

cout << "The entered integer is: " << num << endl;

} catch (const out\_of\_range& e) {

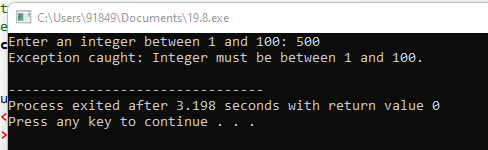
cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

OUTPUT:



9. Create a C++ program that reads an integer from the user and calculates its square root. Handle the exception if the user enters a negative number and display an appropriate error message.

#include <iostream>

#include <cmath>

#include <stdexcept>

using namespace std;

int main() {

try {

int num;

cout << "Enter an integer: ";

cin >> num;

if (num < 0) {

throw invalid\_argument("Negative numbers do not have real square roots.");

}

double squareRoot = sqrt(num);

cout << "Square root of " << num << " is: " << squareRoot << endl;

} catch (const invalid\_argument& e) {

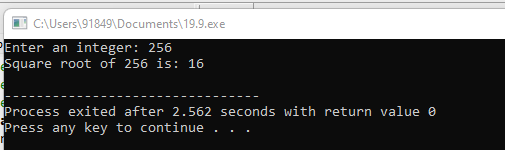
cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

OUTPUT:



10. Write a function int findElement(const std::vector<int>& vec, int target) that searches for a target element in a vector. Handle the exception if the vector is empty and return a default value in such cases.

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

int findElement(const vector<int>& vec, int target) {

if (vec.empty()) {

throw runtime\_error("Vector is empty.");

}

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

if (vec[i] == target) {

return i;

}

}

return -1;

}

int main() {

vector<int> numbers = {10, 20, 30, 40, 50};

try {

int index = findElement(numbers, 30);

if (index != -1) {

cout << "Element found at index: " << index << endl;

} else {

cout << "Element not found." << endl;

}

} catch (const runtime\_error& e) {

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

}

try {

int index = findElement({}, 30);

if (index != -1) {

cout << "Element found at index: " << index << endl;

} else {

cout << "Element not found." << endl;

}

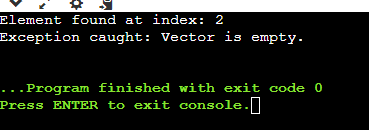
} catch (const runtime\_error& e) {

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

}

return 0;

}

OUTPUT:  


11. Write a C++ program to calculate the factorial of a number using recursion. Handle the exception if the user enters a negative number and throw an exception. Write a test case to calculate the factorial of a positive number and validate the result.

Test Cases:

Input: 5 (positive number)

Output: Factorial of 5 is 120

#include <iostream>

#include <stdexcept>

using namespace std;

int factorial(int n) {

if (n < 0) {

throw invalid\_argument("Factorial is not defined for negative numbers.");

}

if (n == 0 || n == 1) {

return 1;

}

return n \* factorial(n - 1);

}

int main() {

int number = 5;

try {

int result = factorial(number);

cout << "Factorial of " << number << " is " << result << endl;

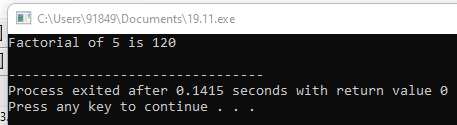
} catch (const invalid\_argument& e) {

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

}

return 0;

}

OUTPUT:  


12. Define a class TemperatureConverter with a member function convertToFahrenheit(double celsius). Handle the exception if the Celsius temperature provided is below absolute zero (-273.15°C) and throw an exception. Write a test case to convert a valid Celsius temperature to Fahrenheit.

Test Cases:

Input: 25 (valid Celsius temperature)

Output: Fahrenheit temperature: <result>

#include <iostream>

#include <stdexcept>

using namespace std;

class TemperatureConverter {

public:

double convertToFahrenheit(double celsius) {

if (celsius < -273.15) {

throw invalid\_argument("Celsius temperature cannot be below absolute zero.");

}

return (celsius \* 9 / 5) + 32;

}

};

int main() {

TemperatureConverter converter;

double celsius = 25;

try {

double fahrenheit = converter.convertToFahrenheit(celsius);

cout << "Fahrenheit temperature: " << fahrenheit << endl;

} catch (const invalid\_argument& e) {

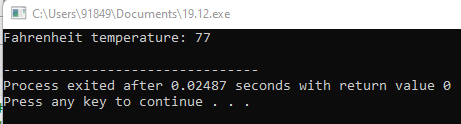
cout << "Exception caught: " << e.what() << endl;

}

return 0;

}

OUTPUT:



13. Create a class Array to represent a dynamic array. Implement member functions to insert elements into the array and retrieve elements by index. Handle the exception if an invalid index is provided for retrieval and throw an exception. Write a test case to insert elements and retrieve elements by index.

Test Cases:

Insert elements into the array and verify if they are inserted correctly.

Retrieve elements by index and validate the retrieved values.

#include <iostream>

#include <stdexcept>

using namespace std;

class Array {

private:

int\* arr;

int size;

public:

Array(int initialSize = 10) {

arr = new int[initialSize];

size = initialSize;

}

~Array() {

delete[] arr;

}

void insert(int index, int element) {

if (index < 0 || index >= size) {

throw out\_of\_range("Invalid index for insertion.");

}

arr[index] = element;

}

int retrieve(int index) {

if (index < 0 || index >= size) {

throw out\_of\_range("Invalid index for retrieval.");

}

return arr[index];

}

};

int main() {

Array arr;

for (int i = 0; i < 10; ++i) {

arr.insert(i, i \* 2);

}

cout << "Elements in the array:" << endl;

for (int i = 0; i < 10; ++i) {

try {

cout << "Index " << i << ": " << arr.retrieve(i) << endl;

} catch (const out\_of\_range& e) {

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

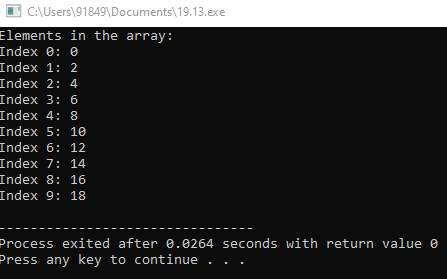
}

}

return 0;

}

OUTPUT:



14. Implement a class Student with attributes rollNumber and marks. Create a dynamic array of Student objects and store student details in the array. Handle the exception if memory allocation fails and throw an exception. Write a test case to create and populate the array with student details.

Test Cases:

Verify if the student details are stored correctly in the array.

#include <iostream>

#include <stdexcept>

using namespace std;

class Student {

public:

int rollNumber;

float marks;

Student(int roll = 0, float score = 0) : rollNumber(roll), marks(score) {}

};

class StudentArray {

private:

Student\* students;

int size;

public:

StudentArray(int initialSize) {

students = new Student[initialSize];

if (students == nullptr) {

throw bad\_alloc();

}

size = initialSize;

}

~StudentArray() {

delete[] students;

}

void setStudent(int index, int roll, float score) {

if (index < 0 || index >= size) {

throw out\_of\_range("Invalid index for setting student details.");

}

students[index].rollNumber = roll;

students[index].marks = score;

}

Student getStudent(int index) {

if (index < 0 || index >= size) {

throw out\_of\_range("Invalid index for retrieving student details.");

}

return students[index];

}

};

int main() {

const int numStudents = 3;

try {

StudentArray studentArray(numStudents);

studentArray.setStudent(0, 101, 85.5);

studentArray.setStudent(1, 102, 78.0);

studentArray.setStudent(2, 103, 92.3);

cout << "Student Details:" << endl;

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

Student student = studentArray.getStudent(i);

cout << "Roll Number: " << student.rollNumber << ", Marks: " << student.marks << endl;

}

} catch (const out\_of\_range& e) {

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

} catch (const bad\_alloc& e) {

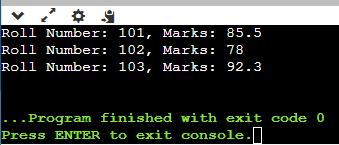
cout << "Memory allocation failed." << endl;

}

return 0;

}

OUTPUT:



15. Define a class Matrix to represent a 2-dimensional matrix. Implement member functions to perform matrix addition and multiplication. Handle the exception if matrices with incompatible dimensions are provided for addition or multiplication and throw an exception. Write a test case to perform matrix addition and multiplication.

Test Cases:

Perform matrix addition and verify the result.

Perform matrix multiplication and validate the result.

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

class Matrix {

private:

vector<vector<int>> data;

int rows;

int cols;

public:

Matrix(int rows, int cols) : rows(rows), cols(cols) {

data.resize(rows, vector<int>(cols, 0));

}

void set(int row, int col, int value) {

if (row >= rows || col >= cols || row < 0 || col < 0) {

throw out\_of\_range("Index out of range.");

}

data[row][col] = value;

}

Matrix add(const Matrix &other) const {

if (rows != other.rows || cols != other.cols) {

throw invalid\_argument("Matrix dimensions are not compatible for addition.");

}

Matrix result(rows, cols);

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

for (int j = 0; j < cols; ++j) {

result.data[i][j] = data[i][j] + other.data[i][j];

}

}

return result;

}

Matrix multiply(const Matrix &other) const {

if (cols != other.rows) {

throw invalid\_argument("Matrix dimensions are not compatible for multiplication.");

}

Matrix result(rows, other.cols);

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

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

result.data[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return result;

}

void display() const {

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

for (int j = 0; j < cols; ++j) {

cout << data[i][j] << " ";

}

cout << endl;

}

}

};

int main() {

Matrix A(2, 2);

A.set(0, 0, 1);

A.set(0, 1, 2);

A.set(1, 0, 3);

A.set(1, 1, 4);

Matrix B(2, 2);

B.set(0, 0, 5);

B.set(0, 1, 6);

B.set(1, 0, 7);

B.set(1, 1, 8);

try {

Matrix C = A.add(B);

cout << "Matrix Addition Result:" << endl;

C.display();

} catch (const exception &e) {

cout << "Error during addition: " << e.what() << endl;

}

Matrix D(2, 3);

D.set(0, 0, 1);

D.set(0, 1, 2);

D.set(0, 2, 3);

D.set(1, 0, 4);

D.set(1, 1, 5);

D.set(1, 2, 6);

Matrix E(3, 2);

E.set(0, 0, 7);

E.set(0, 1, 8);

E.set(1, 0, 9);

E.set(1, 1, 10);

E.set(2, 0, 11);

E.set(2, 1, 12);

try {

Matrix F = D.multiply(E);

cout << "Matrix Multiplication Result:" << endl;

F.display();

} catch (const exception &e) {

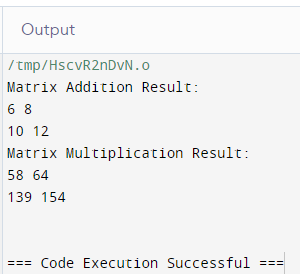
cout << "Error during multiplication: " << e.what() << endl;

}

return 0;

}

OUTPUT:



16. Write a C++ program to read a file and display its contents. Handle the exception if the file cannot be opened and throw an exception. Write a test case to read and display the contents of a valid file.

Test Cases:

Verify if the contents of the file are displayed correctly**.**

#include <iostream>

#include <fstream>

#include <stdexcept>

using namespace std;

void displayFileContents(const string& filename) {

ifstream file(filename);

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

throw runtime\_error("Unable to open file.");

}

string line;

while (getline(file, line)) {

cout << line << endl;

}

file.close();

}

int main() {

const string filename = "test.txt";

try {

cout << "Contents of the file '" << filename << "':" << endl;

displayFileContents(filename);

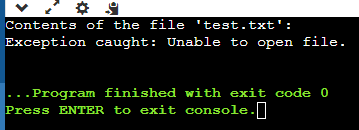
} catch (const exception& e) {

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

}

return 0;

}

OUTPUT:  


17. Create a base class Shape with a pure virtual function draw(). Derive classes Circle, Rectangle, and Triangle from Shape. Implement the draw() function in each derived class to draw the respective shape. Write a function to dynamically create an array of pointers to Shape objects, populate it with objects of different shapes, and call the draw() function for each object.

Test Cases:

Verify if the draw() function correctly displays the respective shape for each object in the array.

#include <iostream>

#include <vector>

using namespace std;

class Shape {

public:

virtual void draw() const = 0;

};

class Circle : public Shape {

public:

void draw() const override {

cout << "Drawing Circle" << endl;

}

};

class Rectangle : public Shape {

public:

void draw() const override {

cout << "Drawing Rectangle" << endl;

}

};

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing Triangle" << endl;

}

};

vector<Shape\*> createShapes() {

vector<Shape\*> shapes;

shapes.push\_back(new Circle());

shapes.push\_back(new Rectangle());

shapes.push\_back(new Triangle());

return shapes;

}

void drawShapes(const vector<Shape\*>& shapes) {

for (const auto& shape : shapes) {

shape->draw();

}

}

void deleteShapes(vector<Shape\*>& shapes) {

for (auto& shape : shapes) {

delete shape;

}

}

int main() {

vector<Shape\*> shapes = createShapes();

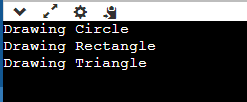
drawShapes(shapes);

deleteShapes(shapes);

return 0;

}

OUTPUT:



18. Write a program that demonstrates runtime polymorphism by creating a base class Shape with a virtual function displayArea(). Derive classes Rectangle and Circle from Shape and implement the displayArea() function to calculate and display the area of the shape. Create objects of both derived classes and call the displayArea() function using pointers to the base class.

Test Cases:

Verify if the displayArea() function correctly calculates and displays the area of both shapes.

#include <iostream>

using namespace std;

class Shape {

public:

virtual void displayArea() const = 0;

};

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

void displayArea() const override {

double area = length \* width;

cout << "Area of Rectangle: " << area << endl;

}

};

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

void displayArea() const override {

double area = 3.14159 \* radius \* radius;

cout << "Area of Circle: " << area << endl;

}

};

int main() {

Rectangle rectangle(5.0, 3.0);

Circle circle(2.5);

Shape\* shape1 = &rectangle;

Shape\* shape2 = &circle;

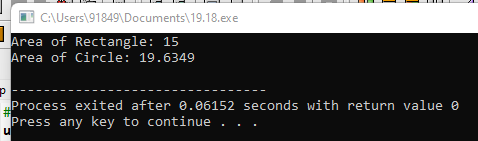
shape1->displayArea();

shape2->displayArea();

return 0;

}

OUTPUT:



19. Implement a function to read integers from a file and calculate their sum. Handle exceptions for file opening errors, invalid input format, and arithmetic overflow. Rethrow appropriate exceptions after catching them.

Test Cases:

Provide test files with valid and invalid input formats and verify if the function handles exceptions correctly.

#include <iostream>

#include <fstream>

#include <vector>

#include <stdexcept>

#include <limits>

using namespace std;

int sumIntegersFromFile(const string& filename) {

ifstream file(filename);

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

throw runtime\_error("Failed to open file.");

}

int sum = 0;

int num;

while (file >> num) {

if (sum > 0 && num > numeric\_limits<int>::max() - sum) {

file.close();

throw overflow\_error("Arithmetic overflow occurred while calculating the sum.");

} else if (sum < 0 && num < numeric\_limits<int>::min() - sum) {

file.close();

throw overflow\_error("Arithmetic overflow occurred while calculating the sum.");

}

sum += num;

}

if (file.bad()) {

file.close();

throw runtime\_error("Error reading file.");

}

file.close();

return sum;

}

int main() {

try {

cout << "Sum of integers from valid\_input.txt: " << sumIntegersFromFile("valid\_input.txt") << endl;

cout << "Sum of integers from invalid\_input.txt: " << sumIntegersFromFile("invalid\_input.txt") << endl;

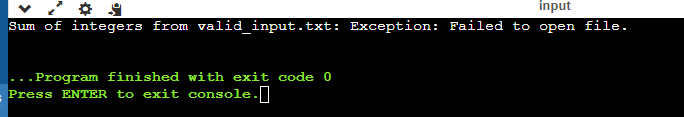
} catch (const exception& e) {

cout << "Exception: " << e.what() << endl;

}

return 0;

}

OUTPUT:  


20. Define a class BankAccount with attributes accountNumber and balance. Implement a member function withdraw() to perform a withdrawal operation. Throw exceptions for invalid withdrawal amounts, insufficient balance, and other applicable error scenarios.

Test Cases:

Test the withdraw() function with various withdrawal amounts and verify if the exceptions are thrown and handled appropriately.

#include <iostream>

#include <stdexcept>

using namespace std;

class BankAccount {

private:

int accountNumber;

double balance;

public:

BankAccount(int accountNum, double initialBalance) : accountNumber(accountNum), balance(initialBalance) {}

void withdraw(double amount) {

if (amount <= 0) {

throw invalid\_argument("Invalid withdrawal amount.");

}

if (amount > balance) {

throw runtime\_error("Insufficient balance.");

}

balance -= amount;

cout << "Withdrawal of $" << amount << " successful. New balance: $" << balance << endl;

}

};

int main() {

BankAccount account(12345, 1000.0);

try {

account.withdraw(500);

account.withdraw(2000);

account.withdraw(0);

} catch (const invalid\_argument& e) {

cout << "Invalid Argument Error: " << e.what() << endl;

} catch (const runtime\_error& e) {

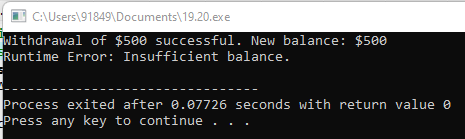
cout << "Runtime Error: " << e.what() << endl;

}

return 0;

}

OUTPUT:



21. . Create a program that reads a series of integers from the user until a sentinel value is entered. Use exception handling to handle invalid input types and out-of-range values. Rethrow exceptions to handle different error scenarios.

Test Cases:

Provide test cases with valid and invalid input values to verify if the program handles exceptions correctly

#include <iostream>

#include <stdexcept>

#include <limits>

using namespace std;

int main() {

const int SENTINEL\_VALUE = -1;

try {

while (true) {

cout << "Enter an integer (or -1 to quit): ";

int num;

cin >> num;

if (cin.fail()) {

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

throw runtime\_error("Invalid input type. Please enter an integer.");

}

if (num == SENTINEL\_VALUE) {

break;

}

if (num < 0 || num > 100) {

throw out\_of\_range("Input out of range. Please enter a value between 0 and 100.");

}

cout << "You entered: " << num << endl;

}

} catch (const runtime\_error& e) {

cout << "Runtime Error: " << e.what() << endl;

} catch (const out\_of\_range& e) {

cout << "Out of Range Error: " << e.what() << endl;

}

return 0;

}

OUTPUT:  
