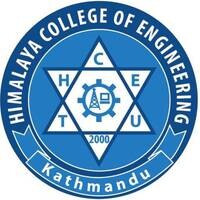


**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**



**HIMALAYA COLLEGE OF ENGINEERING**

**CHYASAL, LALITPUR**

**Lab Report No: - 7**

**Title: - Exception handling and Stream computation**

**Submitted by: - Submitted To: -**

**Name: - Atul Bhattarai Department Of Computer and Electronics**

**Roll NO: - HCOE 081 BEI 011 Checked by: -**

**Date of submission: -**

### **Objective**

* To understand and implement exception handling using try, catch, and throw in C++.
* To perform basic input/output stream manipulations and computations using file and console streams.

**Theory**

#### **Exception Handling in C++**

Exception handling is a mechanism in C++ that allows a program to respond to **runtime errors or unusual situations** gracefully, instead of terminating abruptly. It separates **normal code logic** from **error handling**, making the program more reliable and easier to maintain.

The basic components of exception handling in C++ are:

* **try block**: Contains the code that may potentially throw an exception. It is used to wrap operations that need monitoring for errors.
* **throw statement**: Used to signal the occurrence of an exception. It transfers control from the try block to the matching catch block.
* **catch block**: Handles the exception thrown. Multiple catch blocks can be used to handle different types of exceptions.

This mechanism supports **type-based handling**, meaning specific types of exceptions can be caught (e.g., int, float, string, or even user-defined classes).

**Benefits of Exception Handling:**

* Improves **program robustness**
* Helps to **separate error handling** from regular logic
* Supports **propagation of errors** across function calls
* Ensures proper **resource cleanup** when used with destructors

**Example Scenario:**

Dividing a number by zero is undefined and can crash a program. By using exception handling, we can throw an error message and terminate the operation safely without crashing the program.

**Stream Computation in C++**

Streams in C++ are objects that facilitate **input and output operations**. These are part of the standard C++ library and enable communication with:

* the **keyboard and screen** (console),
* **files** stored on disk, and
* **in-memory buffers** (e.g., stringstreams).

##### Types of Streams:

1. **Standard Input/Output Streams:**
   * cin: standard input stream (keyboard input)
   * cout: standard output stream (console output)
   * cerr: for error messages (unbuffered)
   * clog: for error messages (buffered)
2. **File Streams:**
   * ifstream: input file stream (read from file)
   * ofstream: output file stream (write to file)
   * fstream: combined input and output stream
3. **String Streams:**
   * stringstream: used for in-memory string manipulation (similar to file I/O)

##### Stream Computation:

Stream computation refers to using these stream objects to **perform operations on data** as it flows in or out of the program. Examples include:

* Reading numeric values from a file
* Writing formatted output to a file
* Calculating sums, averages, or parsing data during I/O
* Converting between strings and numbers using stringstreams

**Importance in Real World:**

* File streams are used in applications that handle **data storage, logs, and reports**.
* String streams are useful for **parsing user input, serialization, or text formatting.**
* Stream-based processing is efficient and scalable for **large data operations.**

By mastering both **exception handling** and **stream computation**, a C++ programmer can create robust applications that are not only safe from unexpected errors but also capable of reading and writing data efficiently for real-world tasks such as file processing, data logging, and user interaction.

**Lab Assignment**

**Q1. Write a program to show exception handling inside a class constructor and destructor. Handle constructor exceptions properly; avoid throwing from destructor.**

#include <iostream>

#include <stdexcept>

using namespace std;

class Sample {

public:

    Sample(int x) {

        cout << "Constructor called.\n";

        if (x < 0) {

            throw runtime\_error("Negative value not allowed in constructor.");

        }

    }

    ~Sample() {

        cout << "Destructor called.\n";

        // Destructors should not throw exceptions

    }

};

int main() {

    try {

        Sample obj(-1); // This will throw

    }

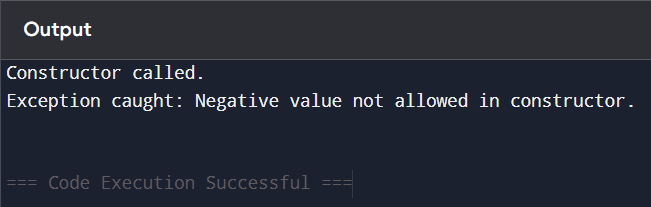
    catch (const exception &e) {

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

    }

    return 0;

}

**­­­**

**Q2. Write a C++ program to handle divide-by-zero exception using try-catch block. Input two numbers. If denominator is zero, throw and catch an exception.**

#include <iostream>

using namespace std;

int main() {

    int num, denom;

    cout << "Enter numerator: ";

    cin >> num;

    cout << "Enter denominator: ";

    cin >> denom;

    try {

        if (denom == 0)

            throw "Division by zero error!";

        cout << "Result: " << num / denom << endl;

    }

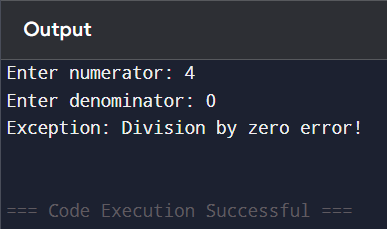
    catch (const char\* msg) {

        cout << "Exception: " << msg << endl;

    }

    return 0;

}

****

**Q3. Write a program to store and retrieve 'n' recordsof items (item\_ID, name, price, mfd\_date, company) in Inventory system.**

#include <iostream>

#include <fstream>

using namespace std;

struct Item {

    int item\_ID;

    char name[30];

    float price;

    char mfd\_date[20];

    char company[30];

};

int main() {

    int n;

    cout << "Enter number of items: ";

    cin >> n;

    Item item;

    ofstream out("inventory.dat", ios::binary);

    for (int i = 0; i < n; i++) {

        cout << "\nEnter details for item " << i + 1 << ":\n";

        cout << "ID: "; cin >> item.item\_ID;

        cout << "Name: "; cin.ignore(); cin.getline(item.name, 30);

        cout << "Price: "; cin >> item.price;

        cout << "Mfd Date: "; cin.ignore(); cin.getline(item.mfd\_date, 20);

        cout << "Company: "; cin.getline(item.company, 30);

        out.write((char\*)&item, sizeof(item));

    }

    out.close();

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

    cout << "\nStored Item Records:\n";

    while (in.read((char\*)&item, sizeof(item))) {

        cout << "\nID: " << item.item\_ID

             << "\nName: " << item.name

             << "\nPrice: " << item.price

             << "\nMfd Date: " << item.mfd\_date

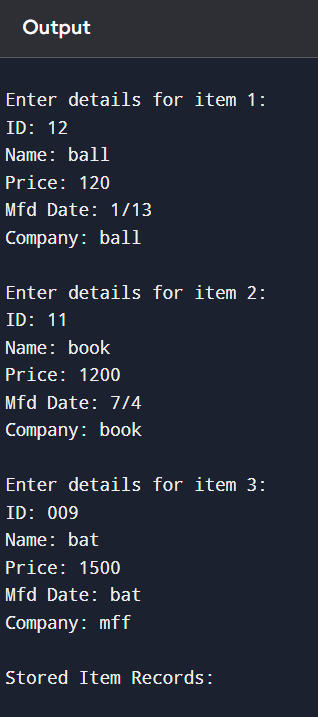
             << "\nCompany: " << item.company << endl;

    }

    in.close();

    return 0;

}

****

**Q4. Write a program to make simple library management system of a college. Your program should store and retrieve the information (Book Name, Book ID, Number of books and purchase date).**

#include <iostream>

#include <fstream>

using namespace std;

struct Book {

    int book\_ID;

    char book\_name[50];

    int quantity;

    char purchase\_date[20];

};

int main() {

    int n;

    cout << "Enter number of books: ";

    cin >> n;

    Book book;

    ofstream out("library.dat", ios::binary);

    for (int i = 0; i < n; i++) {

        cout << "\nEnter details for book " << i + 1 << ":\n";

        cout << "Book ID: "; cin >> book.book\_ID;

        cout << "Book Name: "; cin.ignore(); cin.getline(book.book\_name, 50);

        cout << "Quantity: "; cin >> book.quantity;

        cout << "Purchase Date: "; cin.ignore(); cin.getline(book.purchase\_date, 20);

        out.write((char\*)&book, sizeof(book));

    }

    out.close();

    ifstream in("library.dat", ios::binary);

    cout << "\nStored Library Records:\n";

    while (in.read((char\*)&book, sizeof(book))) {

        cout << "\nBook ID: " << book.book\_ID

             << "\nBook Name: " << book.book\_name

             << "\nQuantity: " << book.quantity

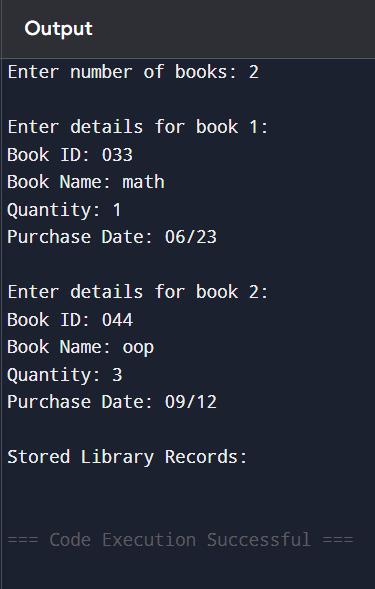
             << "\nPurchase Date: " << book.purchase\_date << endl;

    }

    in.close();

    return 0;

}

****

### **Discussion**

Exception handling allowed safer code execution, especially for unpredictable inputs like division. Stream I/O enabled data persistence and processing of external data, simulating real-world applications. This exercise demonstrated how modern C++ supports both robustness and scalability in program design.

**Conclusion**

The lab successfully illustrated exception handling and stream computation in C++. Handling runtime errors with exceptions improves program reliability, while streams enable data operations across files and memory. These concepts are crucial for building fault-tolerant and data-driven applications.