**LAB 11**

**QUESTION 1:**

**This Q1. You are tasked to develop a reusable array class where accessing an invalid index must throw an exception.**

**Requirements:**

**● Create a class template SafeArray<T>.**

**● Add methods:**

**○ void set(int index, T value)**

**○ T get(int index)**

**● Throw an OutOfBoundsException if:**

**○ Index < 0**

**○ Index >= size**

**● Catch the exception in main() and print "Invalid array index access.".**

**● The class should be able to work with different types (int, double, string).**

**PROGRAM**:

#include<iostream>

#include<stdexcept>

using namespace std;

class OutOfBoundsException : public exception{

    public:

    const char\* what() const noexcept override{

        return "out of bound index";

}

};

template<typename T>

class SafeArray{

    private:

    T array[5];

    int size=5;

    public:

    void set(int *index*, T *value*){

            array[*index*]=*value*;

    }

     T get(int *index*){

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

                throw OutOfBoundsException();

               }

            return array[*index*];

     }

};

int main(){

    int value;

    cout<<endl<<"integer"<<endl;

    try{

    class SafeArray<int> s;

    for(int i=0;i<5;i++){

        cout<<"Enter value for "<<i+1<<" :";

        cin>>value;

        s.set(i,value);

    }

     cout<<"3rd index of integer array :"<<s.get(3)<<endl;

     cout<<"exception handling case :" <<s.get(-1)<<endl;

     }catch (const OutOfBoundsException& e) {

        cout << e.what() << endl;

    }

    cout<<endl<<"double "<<endl;

    double  val;

        try{

        SafeArray<double> s2;

        for(int i=0;i<5;i++){

            cout<<"Enter value for "<<i+1<<" :";

            cin>>val;

            s2.set(i,val);

        }

         cout<<"3rd index of integer array :"<<s2.get(3)<<endl;

         cout<<"exception handling case :" <<s2.get(-1)<<endl;

    } catch (const OutOfBoundsException& e) {

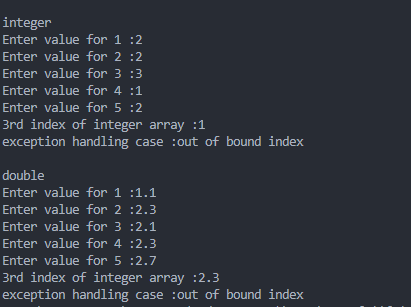
        cout << e.what() << endl;

    }

    return 0;

}

**RESULT:**

****

**QUESTION#2**

**Implement a stack data structure for any data type that throws an error when trying to pop from an empty stack.**

**Requirements:**

**● Create a class template Stack<T>.**

**● Implement methods:**

**○ void push(T item)**

**○ T pop()**

**● Throw a custom StackUnderflowException if pop is called on an empty stack.**

**● Handle the exception in main() and display "Stack is empty. Cannot pop.".**

**PROGRAM**:

#include<iostream>

#include <stdexcept>

using namespace std;

class StackUnderflowException : public exception{

    public:

    const char\* what() const noexcept override{

        return "stack is empty";

}

};

template <class T>

class Stack{

    private:

        T array[5];

        int count;

    public:

        Stack(){

            count=0;

        }

        void push(T *item*){

           array[count]=*item*;

           count++;

        }

        T pop(){

            int temp=0;

                if(count==0){

                    throw StackUnderflowException();

                }

           count--;

           return array[count];

        }

};

int main() {

    Stack<int> s;

    s.push(2);

    s.push(3);

    cout << "Items popped from the stack:" << endl;

    try {

        cout << "Popped: " << s.pop() << endl;

    } catch (StackUnderflowException& e) {

        cout << e.what() << endl;

    }

    try {

        cout << "Popped: " << s.pop() << endl;

    } catch (StackUnderflowException& e) {

        cout << e.what() << endl;

    }

    try {

        cout << "Popped: " << s.pop() << endl;

    } catch (StackUnderflowException& e) {

        cout << e.what() << endl;

    }

    return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.

**QUESTION#3**

**Develop a generic matrix multiplication function. Before multiplying, ensure the**

**matrices are compatible.**

**Requirements:**

**● Create a class template Matrix<T> that holds a 2D array.**

**● Write a function multiply(Matrix<T> a, Matrix<T> b):**

**○ Throw DimensionMismatchException if number of columns in a != number**

**of rows in b.**

**● Catch the exception and print "Matrix dimensions incompatible for multiplication.".**

**● Use the template to multiply matrices of int and float.**

#include<iostream>

#include <stdexcept>

using namespace std;

class DimensionMismatchException : public exception {

public:

    const char\* what() const noexcept override {

        return "Matrix dimensions are not matching";

    }

};

template <typename T>

class Matrix {

private:

    int column, rows;

    T\*\* array;

public:

    Matrix(int *rows*, int *column*) : rows(*rows*), column(*column*) {

        array = new T\*[*rows*];

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

            array[i] = new T[*column*];

        }

    }

    ~Matrix() {

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

            delete[] array[i];

        }

        delete[] array;

    }

    void setMatrix() {

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

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

                cout << "Enter for index [" << i << "][" << j << "] : ";

                cin >> array[i][j];

            }

        }

    }

    T getArray(int *i*, int *j*) {

        return array[*i*][*j*];

    }

    int getRows() const {

        return rows;

    }

    int getColumns() const {

        return column;

    }

    friend Matrix<T> MultiplyMatrix(const Matrix<T>& *a*, const Matrix<T>& *b*) {

        if (*a*.column != *b*.rows) {

            throw DimensionMismatchException();

        }

        Matrix<T> result(*a*.rows, *b*.column);

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

            for (int j = 0; j < *b*.column; j++) {

                result.array[i][j] = 0;

                for (int k = 0; k < *a*.column; k++) {

                    result.array[i][j] += *a*.array[i][k] \* *b*.array[k][j];

                }

            }

        }

        return result;

    }

    void display() const {

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

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

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

            }

            cout << endl;

        }

    }

};

int main() {

    try {

        cout << "For Integer Matrix:" << endl;

        Matrix<int> M(2, 2);

        Matrix<int> M2(2, 2);

        cout << "Matrix 1:" << endl;

        M.setMatrix();

        cout << "Matrix 2:" << endl;

        M2.setMatrix();

        Matrix<int> M3 = MultiplyMatrix(M, M2);

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

        M3.display();

    }

    catch (const DimensionMismatchException& e) {

        cout << e.what() << endl;

    }

    try {

        cout << "For Float Matrix:" << endl;

        Matrix<float> M3(1, 3);

        Matrix<float> M4(3, 2);

        cout << "Matrix 1:" << endl;

        M3.setMatrix();

        cout << "Matrix 2:" << endl;

        M4.setMatrix();

        Matrix<float> M5 = MultiplyMatrix(M3, M4);

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

        M5.display();

    }

    catch (const DimensionMismatchException& e) {

        cout << e.what() << endl;

    }

    return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

**QUESTION#4**

**Q4. Build a file reader that can read and store data of different types from a file.**

**Requirements:**

**● Create a function template readFromFile<T>(string filename).**

**● If the file does not exist or cannot be opened, throw a FileNotFoundException.**

**● Handle the exception by printing "Error: Unable to open file.".**

**● Read data (e.g., integers, floats, or strings) into an array<T>.**

**● Demonstrate with a sample file.**

#include<iostream>

#include <fstream>

#include <stdexcept>

using namespace std;

class FileNotFoundException: public exception{

    public:

    const char\* what() const noexcept override{

        return "unable to open to";

    }

    };

class FileReader{

    public:

template <typename T>

T\* readfromFile(string *filename*,int &*size*){

*size*=0;

    ifstream file(*filename*);

     if(!file){

             throw FileNotFoundException();

     }

     T temp;

    while(file >> temp){

*size*++;

    }

    T\* data=new T[*size*];

    file.clear();

    file.seekg(0);

    for(int i=0;i<*size*;i++){

         file>>data[i];

    }

    file.close();

    return data;

}

};

int main() {

    FileReader F;

    try {

        int size;

        cout << "---int output---" << endl;

        int\* intArray = F.readfromFile<int>("integer.txt", size);

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

            cout << intArray[i] << endl;

        }

        FileReader F2;

        cout << "---string output---" << endl;

        string\* StrArray = F2.readfromFile<string>("string.txt", size);

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

            cout << StrArray[i] << " ";

        }

        cout << endl;

        FileReader F3;

        cout << "---Exception handling case---" << endl;

        double\* DArray = F3.readfromFile<double>("double.txt", size);

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

            cout << DArray[i] << " ";

        }

        cout << endl;

    } catch (FileNotFoundException e) {

        cout << e.what() << endl;

    }

    return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.

**QUESTION#5**

**You are designing a banking application that can handle different types of account**

**balances (float, double).**

**Requirements:**

**● Create a class template BankAccount<T>.**

**● Implement methods:**

**○ void deposit(T amount)**

**○ void withdraw(T amount)**

**● Throw an InsufficientFundsException if withdrawal amount > current balance.**

**● Catch and handle the exception with a message like "Withdrawal failed: Insufficient**

**funds.".**

**● Allow deposit and withdrawal operations through user input.**

#include<iostream>

using namespace std;

class InsufficientFundsException: public exception{

    public:

    const char\* what() const noexcept override{

        return "Insufficient Funds";

    }

    };

template <typename T>

class BankAccount{

    private:

   T amount;

   public:

   BankAccount(){

    amount=0;

   }

   void deposit(T *Amount*){

      amount+=*Amount*;

      cout<<"Amount added sucessfully"<<endl;

   }

   void withdrawal(T *Amount*){

    if(*Amount* > amount){

     throw InsufficientFundsException();

    }

    amount-=*Amount*;

    cout<<"Amount withdrawal sucessfully"<<endl;

   }

};

int main(){

    BankAccount<int> B;

    B.deposit(20000);

    try{

        B.withdrawal(1500);

    }

    catch(InsufficientFundsException e){

        cout<<e.what();

    }

    try{

        B.withdrawal(30000000);

    }

    catch(InsufficientFundsException e){

        cout<<e.what()<<endl;

    }

}

