**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСТИЕТ**

Лабораторная работа №6 по дисциплине «Программирование»

**Обработка исключительных ситуаций**

Группа: **АВТ-342**

Студенты: **Бондаренко А.В.,Фадеев В.А.**

НОВОСИБИРСК 2024

1. **Постановка задачи**

**Вариант 4.**

Добавить в классы и демонстрационную программу обработку исключений при возникновении ошибок: недостатка памяти, выход за пределы диапазона допустимых значений, деление на ноль и т.д. Изменить основную программу так, чтобы она демонстрировала обработку исключений.

1. **Исключительные ситуации**

**-** неправильный ввод длины списка (не число, 0, менее 0, излишне большая длина);

- неправильный ввод типа матрицы (не 0-3, не число);

- неправильный ввод размерностей матрицы (не число, 0, менее 0);

- неправильный ввод содержимого матрицы (не число);

- неправильный ввод пункта в меню операций со списком (не число от 0 до 5, не число).

**3. Определение пользовательских классов с комментариями**

1. Класс Matrix

Описание: Это основной класс для работы с матрицами, который содержит основные операции и методы для управления матрицами.

Поля:

* int rows — количество строк в матрице.
* int cols — количество столбцов в матрице.
* int\*\* data — двумерный массив, хранящий элементы матрицы.

Методы:

* Конструкторы и деструктор:
  + Matrix() — конструктор по умолчанию, инициализирует пустую матрицу.
  + Matrix(int rows, int cols) — конструктор, создающий матрицу заданного размера.
  + Matrix(const Matrix&) — конструктор копирования.
  + ~Matrix() — деструктор для освобождения памяти.
* Операции и методы:
  + Matrix multiply(const Matrix&) const — метод для умножения матриц.
  + Matrix transpose() const — метод для транспонирования матрицы.
  + Matrix operator=(const Matrix&) — оператор присваивания.
  + Matrix operator-(const Matrix&) const — оператор вычитания матриц.
  + int operator()() const — оператор, возвращающий размер матрицы.
  + int& operator()(int row, int column) — оператор индексирования по элементам матрицы.
  + Дружественные операторы:
    - Matrix operator+(const Matrix&, const Matrix&) — оператор сложения двух матриц.
    - istream& operator>>(istream& is, Matrix& matrix) — оператор ввода из стандартного потока.
    - ostream& operator<<(ostream& os, const Matrix& matrix) — оператор вывода в стандартный поток.
    - ofstream& operator<<(ofstream& ofs, const Matrix& matrix) — оператор записи в файл.
    - ifstream& operator>>(ifstream& ifs, Matrix& matrix) — оператор чтения из файла.
  + int\* operator[](int i) const — оператор доступа к строкам матрицы.
  + void saveToBinary(const string& filename) const — метод сохранения матрицы в бинарный файл.
  + void loadFromBinary(const string& filename) — метод загрузки матрицы из бинарного файла.
  + virtual void print() const — виртуальный метод для печати матрицы.
  + void resize(int newRows, int newCols) — метод изменения размера матрицы.
  + bool operator==(const Matrix& other) const — оператор сравнения матриц.
* Приватные методы:
  + void freeMemory() — метод для освобождения памяти, используемой под матрицу.

2. Класс IdentityMatrix

Описание: Наследник класса Matrix, представляющий единичную матрицу.

Методы:

* Наследует все методы базового класса Matrix, но может переопределять поведение методов, например:
  + void print() const — переопределение метода print() для вывода единичной матрицы

3. Класс RectangularMatrix

Описание: Наследник класса Matrix, представляющий прямоугольную матрицу.

Методы:

* Наследует все методы базового класса Matrix.
* Переопределение метода print() для вывода прямоугольной матрицы.

4. Класс Node

Описание: Структура для представления узла связного списка, хранящего указатель на объект Matrix.

Поля:

* Matrix\* data — указатель на данные (объект матрицы).
* Node\* next — указатель на следующий узел.
* Node\* prev — указатель на предыдущий узел.

Конструктор:

* Node(Matrix\* newData = nullptr) — конструктор узла с параметром для инициализации данных.

5. Класс List

Описание: Двусвязный список для хранения объектов Matrix.

Поля:

* Node\* head — указатель на начало списка.
* int size — размер списка (количество узлов).

Методы:

* Конструктор List() — инициализация пустого списка.
* Деструктор ~List() — вызывает функцию удаления списка deleteList(\*this).
* Дружественные функции:
  + void append(List& list, Matrix\* data) — добавление элемента в конец списка.
  + void deleteList(List& list) — удаление всех узлов списка.
  + void insertNode(List& list, int index, Matrix\* newMatrix) — вставка нового узла на заданную позицию.
  + void deleteNode(List& list, int index) — удаление узла на заданной позиции.
  + void displayNode(const List& list, int index) — вывод данных конкретного узла.
  + void display(const List& list) — вывод всех элементов списка.
  + bool isEmptyList(const List& list) — проверка, пустой ли список.
  + int getSizeList(const List& list) — получение размера списка.
  + Node\* findNode(const List& list, int index) — нахождение узла по индексу.

6. Класс Menu\_print

Описание: Класс для работы с меню программы.

Методы:

* void matrixTypes() — вывод меню выбора типа матрицы.
* void mathOptions() — вывод математических операций для выполнения над матрицами.
* void listOptions() — вывод операций для работы со списком матриц.

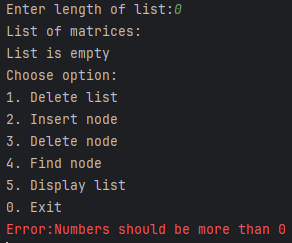
7. Класс OutOfRangeLarge – выводит ошибку, если значение размера списка слишком большое.

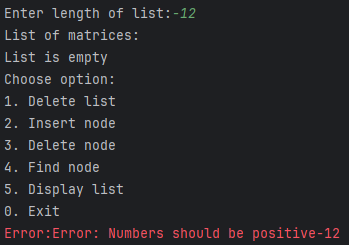
8. Класс OutOfRangeNegative – выводит ошибку, если значение размера списка отрицательное.

9. Класс OutOfRangeNull – выводит ошибку, если значение размера списка равно нулю.

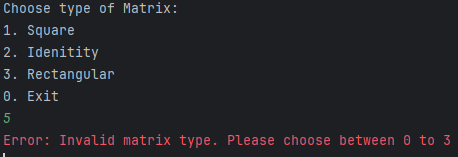
**3. Тестовые примеры выполнения программы**

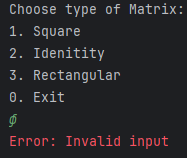
****

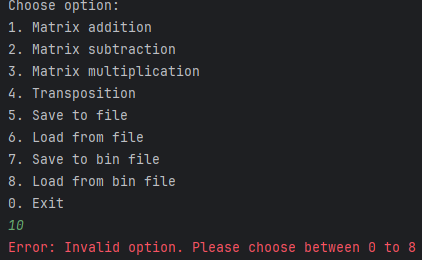
****

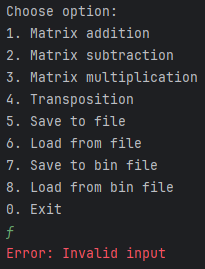
****

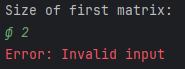
****

****

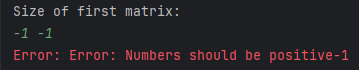
****

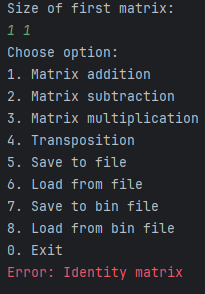
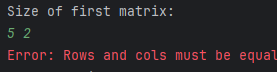
****

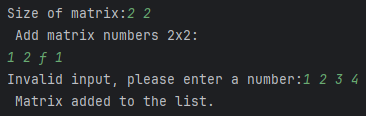
****

****

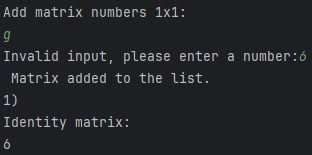
Исключения для квадратной матрицы:



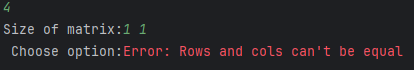
 

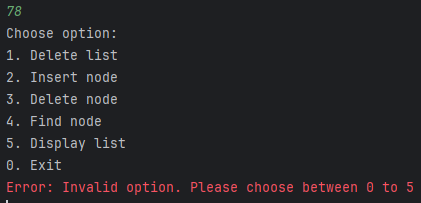


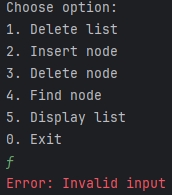
Исключения для единичной матрицы:

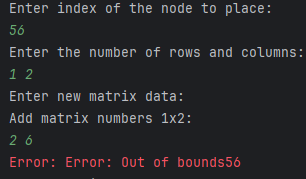


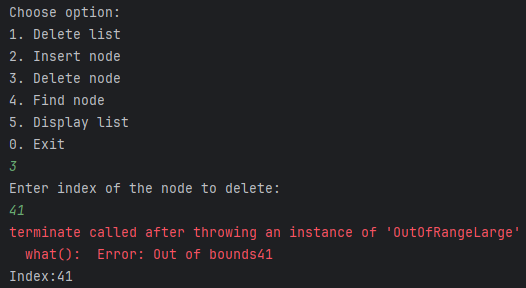
Исключения для прямоугольной матрицы:

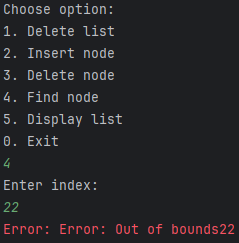


Исключения для меню со списком:  




Вставка элемента:  






**4. Листинг демонстрационной программы**

**main.cpp**

#include "matrix.h"  
#include "OutOfRangeNegative.h"  
#include "OutOfRangeLarge.h"  
#include "OutOfRangeNull.h"  
using namespace std;  
  
int main() {  
 int ListLen;  
 cout << "Enter length of list:";  
 try {  
 cin >> ListLen;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (ListLen < 0) {  
 throw OutOfRangeNegative(ListLen);  
 }  
 if (ListLen == 0) {  
 throw OutOfRangeNull();  
 }  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 return 1;  
 }  
 catch (const exception& e) {  
 cerr << "Error:" << e.what() << '\n';  
 }  
 Menu\_print menu;  
 List list;  
 Node\* head;  
 for (int i = 0; i < ListLen; i++) {  
 Matrix\* result;  
 int NodesMade = 0;  
 while (1) {  
 int MatrixType;  
 bool is\_result\_stored = false;  
 menu.matrixTypes();  
 while (1) {  
 try {  
 cin >> MatrixType;  
 if (cin.fail()) {  
 cin.clear();  
 cin.ignore(1000, '\n');  
 throw invalid\_argument("Invalid input");  
 }  
 if (MatrixType < 0 || MatrixType > 3) {  
 throw out\_of\_range("Invalid matrix type. Please choose between 0 to 3");  
 }  
 break;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 if (MatrixType == 1) {  
 while (1) {  
 int choice;  
 menu.mathOptions();  
 try {  
 cin >> choice;  
 if (cin.fail()) {  
 cin.clear();  
 cin.ignore(1000, '\n');  
 throw invalid\_argument("Invalid input");  
 }  
 if (choice < 0 || choice > 8) {  
 throw out\_of\_range("Invalid option. Please choose between 0 to 8");  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 if (choice == 0) {  
 break;  
 }  
 if (choice == 1) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Enter matrix size:";  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows != cols) {  
 throw logic\_error("Rows and cols must be equal");  
 }  
 if ((rows == 1) && (cols == 1)) {  
 throw logic\_error("Identity matrix");  
 }  
 break;  
 }  
 Matrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 result = new Matrix(matrix1 + matrix2);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 2) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Enter matrix size:" << endl;  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows != cols) {  
 throw logic\_error("Rows and cols must be equal");  
 }  
 if ((rows == 1) && (cols == 1)) {  
 throw logic\_error("Identity matrix");  
 }  
 break;  
 }  
 Matrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 result = new Matrix(matrix1 - matrix2);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 3) {  
 if (is\_result\_stored == false) {  
 int rows1, cols1, rows2, cols2;  
 cout << "Size of first matrix: " << endl;  
 try {  
 while (1) {  
 cin >> rows1;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols1;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows1 != cols1) {  
 throw logic\_error("Rows and cols must be equal");  
 }  
 if ((rows1 == 1) && (cols1 == 1)) {  
 throw logic\_error("Identity matrix");  
 }  
 break;  
 }  
 Matrix matrix1(rows1, cols1);  
 cout << "Size of second matrix: " << endl;  
 while (1) {  
 cin >> rows2;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols2;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows2 != cols2) {  
 throw logic\_error("Rows and cols must be equal");  
 }  
 if ((rows2 == 1) && (cols2 == 1)) {  
 throw logic\_error("Identity matrix");  
 }  
 break;  
 }  
 Matrix matrix2(rows2, cols2);  
 if (cols1 != rows2) {  
 throw logic\_error("Matrices can't be multiplied");  
 }  
 cin >> matrix1 >> matrix2;  
 result = new Matrix(matrix1.multiply(matrix2));  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 4) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Size of matrix: ";  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows != cols) {  
 throw logic\_error("Rows and cols must be equal");  
 }  
 if ((rows == 1) && (cols == 1)) {  
 throw logic\_error("Identity matrix");  
 }  
 break;  
 }  
 Matrix matrix1(rows, cols);  
 cin >> matrix1;  
 result = new Matrix(matrix1.transpose());  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 5) {  
 if (is\_result\_stored) {  
 try {  
 ofstream ofs("matrix.txt");  
 ofs << \*result;  
 ofs.close();  
 cout << "Result saved to file." << endl;  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 if (choice == 6) {  
 try {  
 ifstream ifs("matrix.txt");  
 if (ifs.is\_open()) {  
 ifs >> \*result;  
 cout << "Matrix loaded from file: " << "\n" << \*result << endl;  
 is\_result\_stored = true;  
 ifs.close();  
 }  
 else {  
 throw ios\_base::failure("Unable to open file");  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 if (choice == 7) {  
 try {  
 if (is\_result\_stored) {  
 result->saveToBinary("matrix.bin");  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 if (choice == 8) {  
 try {  
 result->loadFromBinary("matrix.bin");  
 is\_result\_stored = true;  
 result->print();  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Memory allocation error: " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 }  
 }  
 else if (MatrixType == 2) {  
 while (1) {  
 int choice;  
 menu.mathOptions();  
 try {  
 cin >> choice;  
 if (cin.fail()) {  
 cin.clear();  
 cin.ignore(1000, '\n');  
 throw invalid\_argument("Invalid input");  
 }  
 if (choice < 0 || choice > 8) {  
 throw out\_of\_range("Invalid option. Please choose between 0 to 8");  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 if (choice == 0) {  
 break;  
 }  
 if (choice == 1) {  
 try {  
 if (is\_result\_stored == false) {  
 int rows = 1;  
 int cols = 1;  
 IdentityMatrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1 + matrix2;  
 result = new IdentityMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 }  
 if (choice == 2) {  
 try {  
 if (is\_result\_stored == false) {  
 int rows = 1;  
 int cols = 1;  
 IdentityMatrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1 - matrix2;  
 result = new IdentityMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 }  
 if (choice == 3) {  
 try {  
 if (is\_result\_stored == false) {  
 int rows1 = 1;  
 int cols1 = 1;  
 int rows2 = 1;  
 int cols2 = 1;  
 IdentityMatrix matrix1(rows1, cols1);  
 IdentityMatrix matrix2(rows2, cols2);  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1.multiply(matrix2);  
 result = new IdentityMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 }  
 if (choice == 4) {  
 try {  
 if (is\_result\_stored == false) {  
 int rows = 1;  
 int cols = 1;  
 IdentityMatrix matrix1(rows, cols);  
 cin >> matrix1;  
 Matrix temp = matrix1.transpose();  
 result = new IdentityMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 }  
 if (choice == 5) {  
 if (is\_result\_stored) {  
 try {  
 ofstream ofs("matrix.txt");  
 ofs << \*result;  
 ofs.close();  
 cout << "Result saved to file." << endl;  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 if (choice == 6) {  
 try {  
 ifstream ifs("matrix.txt");  
 if (ifs.is\_open()) {  
 ifs >> \*result;  
 cout << "Matrix loaded from file: " << "\n" << \*result << endl;  
 is\_result\_stored = true;  
 ifs.close();  
 }  
 else {  
 throw ios\_base::failure("Unable to open file");  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 if (choice == 7) {  
 try {  
 if (is\_result\_stored) {  
 result->saveToBinary("matrix.bin");  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 if (choice == 8) {  
 try {  
 result->loadFromBinary("matrix.bin");  
 is\_result\_stored = true;  
 result->print();  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 }  
 }  
 else if (MatrixType == 3) {  
 while (1) {  
 int choice;  
 menu.mathOptions();  
 try {  
 cin >> choice;  
 if (cin.fail()) {  
 cin.clear();  
 cin.ignore(1000, '\n');  
 throw invalid\_argument("Invalid input");  
 }  
 if (choice < 0 || choice > 8) {  
 throw out\_of\_range("Invalid option. Please choose between 0 to 8");  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 if (choice == 0) {  
 break;  
 }  
 if (choice == 1) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Enter matrix size:" << endl;  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows == cols) {  
 throw logic\_error("Rows and cols can't be equal");  
 }  
 break;  
 }  
 RectangularMatrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1 + matrix2;  
 result = new RectangularMatrix(temp);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 2) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Enter matrix size:" << endl;  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows == cols) {  
 throw logic\_error("Rows and cols can't be equal");  
 }  
 break;  
 }  
 RectangularMatrix matrix1(rows, cols), matrix2(rows, cols);  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1 - matrix2;  
 result = new RectangularMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 3) {  
 if (is\_result\_stored == false) {  
 try {  
 int rows1, cols1, rows2, cols2;  
 cout << "Size of first matrix: ";  
 while (1) {  
 cin >> rows1;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols1;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows1 == cols1) {  
 throw logic\_error("Rows and cols can't be equal");  
 }  
 break;  
 }  
 RectangularMatrix matrix1(rows1, cols1);  
 cout << "Size of second matrix: ";  
 while (1) {  
 cin >> rows2;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols2;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows2 == cols2) {  
 throw logic\_error("Rows and cols can't be equal");  
 }  
 break;  
 }  
 RectangularMatrix matrix2(rows2, cols2);  
 if (cols1 != rows2) {  
 throw logic\_error("Matrices can't be multiplied");  
 }  
 cin >> matrix1 >> matrix2;  
 Matrix temp = matrix1.multiply(matrix2);  
 result = new RectangularMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 4) {  
 if (is\_result\_stored == false) {  
 int rows, cols;  
 cout << "Size of matrix: ";  
 try {  
 while (1) {  
 cin >> rows;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 cin >> cols;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (rows == cols) {  
 throw logic\_error("Rows and cols can't be equal");  
 }  
 break;  
 }  
 RectangularMatrix matrix1(rows, cols);  
 cin >> matrix1;  
 Matrix temp = matrix1.transpose();  
 result = new RectangularMatrix(temp);  
 list.append(list, result);  
 NodesMade++;  
 is\_result\_stored = true;  
 cout << "Matrix added to the list." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 catch (const logic\_error& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "Impossible to exceed the specified list size" << endl;  
 }  
 }  
 if (choice == 5) {  
 if (is\_result\_stored) {  
 try {  
 ofstream ofs("matrix.txt");  
 ofs << \*result;  
 ofs.close();  
 cout << "Result saved to file." << endl;  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 if (choice == 6) {  
 try {  
 ifstream ifs("matrix.txt");  
 if (ifs.is\_open()) {  
 ifs >> \*result;  
 cout << "Matrix loaded from file: " << "\n" << \*result << endl;  
 is\_result\_stored = true;  
 ifs.close();  
 }  
 else {  
 cout << "Unable to open file" << endl;  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 }  
 if (choice == 7) {  
 try {  
 if (is\_result\_stored) {  
 result->saveToBinary("matrix.bin");  
 }  
 else {  
 cout << "No result to save!" << endl;  
 }  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 if (choice == 8) {  
 try {  
 result->loadFromBinary("matrix.bin");  
 is\_result\_stored = true;  
 result->print();  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "File error: " << e.what() << endl;  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Memory allocation error: " << e.what() << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
 }  
 }  
 }  
 else if (MatrixType == 0) {  
 break;  
 }  
 if (NodesMade >= ListLen) {  
 break;  
 }  
 }  
 }  
 cout << "List of matrices:" << endl;  
 list.display(list);  
 int choice;  
 do {  
 menu.listOptions();  
 try {  
 cin >> choice;  
 if (cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (choice < 0 || choice > 5) {  
 throw out\_of\_range("Invalid option. Please choose between 0 to 5");  
 }  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 switch (choice) {  
 case 0: {  
 break;  
 }  
 case 1: {  
 list.deleteList(list);  
 cout << "List deleted." << endl;  
 break;  
 }  
 case 2: {  
 int index, rows, cols;  
 try {  
 cout << "Enter index of the node to place:" << endl;  
 cin >> index;  
 cout << "Enter the number of rows and columns:" << endl;  
 cin >> rows >> cols;  
 Matrix\* newMatrix = new Matrix(rows, cols);  
 cout << "Enter new matrix data:" << endl;  
 cin >> \*newMatrix;  
 list.insertNode(list, index, newMatrix);  
 cout << "Node placed." << endl;  
 list.display(list);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 catch (const bad\_alloc& e) {  
 cerr << "Error (memory allocation): " << e.what() << '\n';  
 }  
 break;  
 }  
 case 3: {  
 int index1;  
 cout << "Enter index of the node to delete:" << endl;  
 cin >> index1;  
 cout << "Index:" << index1 << endl;  
 list.deleteNode(list, index1);  
 cout << "Node deleted." << endl;  
 list.display(list);  
 break;  
 }  
 case 4: {  
 try {  
 cout << "Enter index:" << endl;  
 int index2;  
 cin >> index2;  
 list.displayNode(list, index2);  
 }  
 catch (const invalid\_argument& e) {  
 cerr << "Error: " << e.what() << '\n';  
 cin.clear();  
 cin.ignore(1000, '\n');  
 }  
 catch (const out\_of\_range& e) {  
 cerr << "Error: " << e.what() << '\n';  
 }  
 break;  
 }  
 case 5: {  
 list.display(list);  
 break;  
 }  
 }  
 } while (choice != 0);  
 return 0;  
}

**matrix.cpp**

#include "matrix.h"  
#include "OutOfRangeNegative.h"  
#include "OutOfRangeLarge.h"  
#include "OutOfRangeNull.h"  
#include <limits>  
using namespace std;  
  
// Конструктор с параметром  
Matrix::Matrix(int rows, int cols) : rows(rows), cols(cols) {  
 if (rows < 0) {  
 throw OutOfRangeNegative(rows);  
 }  
 if (cols < 0) {  
 throw OutOfRangeNegative(cols);  
 }  
 if (rows == 0 || cols == 0) {  
 throw OutOfRangeNull();  
 }  
 data = new int\* [rows];  
 for (int i = 0; i < rows; i++) {  
 data[i] = new int[cols];  
 }  
}  
  
  
Matrix::~Matrix() {  
 for (int i = 0; i < rows; ++i) {  
 delete[] data[i];  
 }  
 delete[] data;  
}  
  
  
void Matrix::freeMemory() {  
 if (data) {  
 for (int i = 0; i < rows; i++) {  
 delete[] data[i];  
 }  
 delete[] data;  
 data = nullptr;  
 }  
}  
  
  
Matrix Matrix::multiply(const Matrix& other) const {  
 Matrix result(rows, other.cols);  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < other.cols; j++) {  
 result.data[i][j] = 0;  
 for (int k = 0; k < cols; k++) {  
 result.data[i][j] += data[i][k] \* other.data[k][j];  
 }  
 }  
 }  
 return result;  
}  
  
  
Matrix Matrix::transpose() const {  
 Matrix result(cols, rows);  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 result.data[j][i] = data[i][j];  
 }  
 }  
 return result;  
}  
  
  
// Перегруженная операция присваивания  
Matrix Matrix::operator=(const Matrix& other) {  
 if (this != &other) {  
 freeMemory();  
 rows = other.rows;  
 cols = other.cols;  
 data = new int\* [rows];  
 for (int i = 0; i < rows; i++) {  
 data[i] = new int[cols];  
 for (int j = 0; j < cols; j++) {  
 data[i][j] = other.data[i][j];  
 }  
 }  
 }  
 return \*this;  
}  
  
// Перегруженная операция вычитания  
Matrix Matrix::operator-(const Matrix& other) const {  
 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;  
}  
  
// Перегруженная операция сложения  
Matrix operator+(const Matrix& m1, const Matrix& m2) {  
 Matrix result(m1.rows, m1.cols);  
 for (int i = 0; i < m1.rows; i++) {  
 for (int j = 0; j < m1.cols; j++) {  
 result.data[i][j] = m1.data[i][j] + m2.data[i][j];  
 }  
 }  
 return result;  
}  
  
// Перегрузка оператора ввода для матриц  
istream& operator>>(istream& is, Matrix& matrix) {  
 cout << "Add matrix numbers " << matrix.rows << "x" << matrix.cols << ":" << endl;  
 for (int i = 0; i < matrix.rows; i++) {  
 for (int j = 0; j < matrix.cols; j++) {  
 while (true) {  
 int value;  
 is >> value;  
 if (is.fail()) {  
 is.clear(); // Сброс состояния  
 is.ignore(numeric\_limits<streamsize>::max(), '\n'); // Игнорируем некорректный ввод  
 cout << "Invalid input, please enter a number: ";  
 } else {  
 matrix.data[i][j] = value;  
 break;  
 }  
 }  
 }  
 }  
 if (is.bad()) {  
 throw ios\_base::failure("Critical error");  
 }  
 return is;  
}  
  
// Перегрузка оператора вывода для матриц  
ostream& operator<<(ostream& os, const Matrix& matrix) {  
 for (int i = 0; i < matrix.rows; i++) {  
 for (int j = 0; j < matrix.cols; j++) {  
 os << matrix.data[i][j] << " ";  
 }  
 os << endl;  
 }  
 return os;  
}  
  
// Сохранение матрицы в текстовый файл  
ofstream& operator<<(ofstream& ofs, const Matrix& matrix) {  
 if (!ofs) {  
 throw ios\_base::failure("Failed to open the file");  
 }  
 ofs << matrix.rows << " " << matrix.cols << endl;  
 for (int i = 0; i < matrix.rows; i++) {  
 for (int j = 0; j < matrix.cols; j++) {  
 if (!ofs) {  
 throw ios\_base::failure("Error during writing data to file");  
 }  
 ofs << matrix.data[i][j] << " ";  
 }  
 if (!ofs) {  
 throw ios\_base::failure("Error during writing data to file");  
 }  
 ofs << endl;  
 }  
 return ofs;  
}  
  
  
// Загрузка матрицы из текстового файла  
ifstream& operator>>(ifstream& ifs, Matrix& matrix) {  
 if (!ifs) {  
 throw ios\_base::failure("Failed to open the file");  
 }  
 ifs >> matrix.rows >> matrix.cols;  
 if (ifs.fail()) {  
 throw ios\_base::failure("Invalid matrix dimensions in file");  
 }  
 matrix.resize(matrix.rows, matrix.cols);  
 for (int i = 0; i < matrix.rows; ++i) {  
 for (int j = 0; j < matrix.cols; ++j) {  
 ifs >> matrix.data[i][j];  
 if (ifs.fail()) {  
 throw ios\_base::failure("Invalid data in file");  
 }  
 }  
 }  
 return ifs;  
}  
  
  
// Метод для сохранения матрицы в бинарный файл  
void Matrix::saveToBinary(const string& filename) const {  
 ofstream ofs(filename, ios::binary );  
 if (!ofs) {  
 throw ios\_base::failure("Failed to open binary file");  
 }  
 ofs << \*this;  
 if (ofs.fail()) {  
 throw ios\_base::failure("Error during binary file writing");  
 }  
 ofs.close();  
}  
  
// Метод для загрузки матрицы из бинарного файла  
void Matrix::loadFromBinary(const string& filename) {  
 ifstream ifs(filename, ios::binary);  
 if (!ifs) {  
 throw ios\_base::failure("Failed to open file");  
 }  
 ifs >> \*this;  
 if (ifs.fail()) {  
 throw ios\_base::failure("Error during binary file reading");  
 }  
 ifs.close();  
}  
  
// Печать матрицы  
void Matrix::print() const {  
 cout << "Square matrix:" << endl;  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 cout << data[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
void IdentityMatrix::print() const {  
 cout << "Identity matrix:" << endl;  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 cout << data[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
void RectangularMatrix::print() const {  
 cout << "Rectangular matrix:" << endl;  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 cout << data[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
void Matrix::resize(int newRows, int newCols) {  
 freeMemory();  
 rows = newRows;  
 cols = newCols;  
 data = new int\*[rows];  
 for (int i = 0; i < rows; i++) {  
 data[i] = new int[cols]();  
 }  
}  
  
bool List::isEmptyList(const List& list) {  
 return head == nullptr;  
}  
  
void List::append(List& list, Matrix\* data) {  
 Node\* newNode = new Node(data);  
 if (isEmptyList(list)) {  
 list.head = newNode;  
 }  
 else {  
 Node\* tail = list.head->prev;  
 tail->next = newNode;  
 newNode->prev = tail;  
 newNode->next = list.head;  
 list.head->prev = newNode;  
 }  
 list.size++;  
}  
  
Node\* List::findNode(const List& list, int index) {  
 if (std::cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (index < 0) {  
 throw OutOfRangeNegative(index);  
 }  
 if (index >= list.size) {  
 throw OutOfRangeLarge(index);  
 }  
 Node\* current = list.head;  
 for (int i = 0; i < index; i++) {  
 current = current->next;  
 }  
 return current;  
}  
  
void List::insertNode(List& list, int index, Matrix\* newMatrix) {  
 if (std::cin.fail()) {  
 throw invalid\_argument("Invalid input");  
 }  
 if (index < 0) {  
 throw OutOfRangeNegative(index);  
 }  
 if (index >= list.size) {  
 throw OutOfRangeLarge(index);  
 }  
 Node\* newNode = new Node(newMatrix);  
  
 if (list.size == 0) {  
 list.head = newNode;  
 newNode->next = newNode->prev = newNode;  
 cout << "Inserted first node. Head data:" << endl;  
 newNode->data->print();  
 } else if (index == 0) {  
 Node\* tail = list.head->prev;  
 newNode->next = list.head;  
 newNode->prev = tail;  
 tail->next = newNode;  
 list.head->prev = newNode;  
 list.head = newNode;  
 cout << "Inserted node at the beginning." << endl;  
 newNode->data->print();  
 } else if (index == list.size) {  
 append(list, newMatrix);  
 return;  
 } else {  
 Node\* current = findNode(list, index);  
 Node\* previous = current->prev;  
 newNode->next = current;  
 newNode->prev = previous;  
 previous->next = newNode;  
 current->prev = newNode;  
 cout << "Inserted node in the middle. Data:" << endl;  
 newNode->data->print();  
 }  
  
 list.size++;  
 cout << "List size after insertion: " << list.size << endl;  
}  
  
void List::deleteNode(List& list, int index) {  
 Node\* removeNode = findNode(list, index);  
 if (list.size == 1) {  
 delete removeNode;  
 list.head = nullptr;  
 cout << "Node deleted \n";  
 }  
 else {  
 Node\* prevNode = removeNode->prev;  
 Node\* nextNode = removeNode->next;  
 prevNode->next = nextNode;  
 nextNode->prev = prevNode;  
 if (removeNode == list.head) {  
 list.head = nextNode;  
 }  
 delete removeNode;  
 cout << "Node deleted \n";  
 }  
 list.size--;  
}  
  
void List::displayNode(const List& list, int index) {  
 if (size == 0) {  
 cout << "List is empty \n";  
 return;  
 }  
 Node\* node = findNode(list, index);  
 if (node != nullptr && node->data != nullptr) {  
 cout << "Matrix at index " << index << ":\n";  
 node->data->print();  
 } else {  
 cout << "Node does not exist or contains no matrix \n";  
 }  
}  
  
void List::deleteList(List& list) {  
 while (size > 0) {  
 deleteNode(list, 0);  
 }  
 cout << "List deleted \n";  
}  
  
void List::display(const List& list) {  
 if (isEmptyList(list)) {  
 cout << "List is empty \n";  
 return;  
 }  
 Node\* current = head;  
 int counter = 1;  
 do {  
 cout << counter << ") " << endl;  
 if (current->data != nullptr) {  
 current->data->print();  
 } else {  
 cout << "Matrix is empty \n";  
 }  
 current = current->next;  
 counter++;  
 } while (current != list.head);  
}  
  
bool Matrix::operator==(const Matrix& other) const {  
 if (this->rows != other.rows || this->cols != other.cols) {  
 return false;  
 }  
 for (int i = 0; i < rows; ++i) {  
 for (int j = 0; j < cols; ++j) {  
 if (this->data[i][j] != other.data[i][j]) {  
 return false;  
 }  
 }  
 }  
  
 return true;  
}  
  
void Menu\_print::matrixTypes() {  
 cout << "Choose type of Matrix:" << endl;  
 cout << "1. Square" << endl;  
 cout << "2. Idenitity" << endl;  
 cout << "3. Rectangular" << endl;  
 cout << "0. Exit" << endl;  
}  
  
void Menu\_print::mathOptions() {  
 cout << "Choose option:" << endl;  
 cout << "1. Matrix addition" << endl;  
 cout << "2. Matrix subtraction" << endl;  
 cout << "3. Matrix multiplication" << endl;  
 cout << "4. Transposition" << endl;  
 cout << "5. Save to file" << endl;  
 cout << "6. Load from file" << endl;  
 cout << "7. Save to bin file" << endl;  
 cout << "8. Load from bin file" << endl;  
 cout << "0. Exit" << endl;  
}  
  
void Menu\_print::listOptions() {  
 cout << "Choose option:" << endl;  
 cout << "1. Delete list" << endl;  
 cout << "2. Insert node" << endl;  
 cout << "3. Delete node" << endl;  
 cout << "4. Find node" << endl;  
 cout << "5. Display list" << endl;  
 cout << "0. Exit" << endl;  
}

**matrix.h**

#ifndef MATRIX\_H  
#define MATRIX\_H  
  
#include <iostream>  
#include <fstream>  
#include <string>  
#include <cctype>  
using namespace std;  
  
class Matrix {  
public:  
 Matrix(int rows, int cols);  
 ~Matrix();  
 Matrix multiply(const Matrix&) const;  
 Matrix transpose() const;  
 Matrix operator=(const Matrix&);  
 Matrix operator-(const Matrix&) const;  
 friend Matrix operator+(const Matrix&, const Matrix&);  
 friend istream& operator>>(istream& is, Matrix& matrix);  
 friend ostream& operator<<(ostream& os, const Matrix& matrix);  
 friend ofstream& operator<<(ofstream& ofs, const Matrix& matrix);  
 friend ifstream& operator>>(ifstream& ifs, Matrix& matrix);  
 void saveToBinary(const string& filename) const;  
 void loadFromBinary(const string& filename);  
 virtual void print() const;  
 void resize(int newRows, int newCols);  
 bool operator==(const Matrix& other) const;  
protected:  
 int rows;  
 int cols;  
 int\*\* data;  
private:  
 void freeMemory();  
};  
  
class IdentityMatrix : public Matrix {  
public:  
 IdentityMatrix(int rows, int cols) : Matrix(rows, cols) {}  
 IdentityMatrix(const IdentityMatrix& other) : Matrix(other) {}  
 IdentityMatrix(const Matrix& other) : Matrix(other) {}  
 ~IdentityMatrix() {}  
 void print() const;  
};  
  
class RectangularMatrix : public Matrix {  
public:  
 RectangularMatrix(int rows, int cols) : Matrix(rows, cols) {}  
 RectangularMatrix(const RectangularMatrix& other) : Matrix(other) {}  
 RectangularMatrix(const Matrix& other) : Matrix(other) {}  
 ~RectangularMatrix() {}  
 void print() const;  
};  
  
class Node {  
public:  
 Matrix\* data;  
 Node\* next;  
 Node\* prev;  
  
 Node(Matrix\* newData = nullptr) : data(newData), next(this), prev(this) {}  
};  
  
class List {  
private:  
 Node\* head;  
 int size;  
public:  
 List() : head(nullptr), size(0) {}  
 ~List() {  
 deleteList(\*this);  
 }  
 void append(List& list, Matrix\* data);  
 void deleteList(List& list);  
 void insertNode(List& list, int index, Matrix\* newMatrix);  
 void deleteNode(List& list, int index);  
 void displayNode(const List& list, int index);  
 void display(const List& list);  
 bool isEmptyList(const List& list);  
 Node\* findNode(const List& list, int index);  
};  
  
class Menu\_print {  
public:  
 void matrixTypes();  
 void mathOptions();  
 void listOptions();  
};  
  
#endif

**OutOfRangeLarge.h**

#ifndef OUT\_OF\_RANGE\_LARGE\_H  
#define OUT\_OF\_RANGE\_LARGE\_H  
  
#include <stdexcept>  
using namespace std;  
  
class OutOfRangeLarge : public out\_of\_range {  
public:  
 explicit OutOfRangeLarge(int value) //LIST!!!  
 : out\_of\_range("Error: Out of bounds" + to\_string(value)) {}  
};  
  
#endif

**OutOfRangeNegative.h**

#ifndef OUT\_OF\_RANGE\_NEGATIVE\_H  
#define OUT\_OF\_RANGE\_NEGATIVE\_H  
  
#include <string>  
#include <stdexcept>  
using namespace std;  
  
class OutOfRangeNegative : public out\_of\_range {  
public:  
 explicit OutOfRangeNegative(int value)  
 : out\_of\_range("Error: Numbers should be positive" + to\_string(value)) {}  
};  
  
#endif

**OutOfRangeNull.h**

#ifndef OUT\_OF\_RANGE\_NULL\_H  
#define OUT\_OF\_RANGE\_NULL\_H  
  
#include <string>  
#include <stdexcept>  
using namespace std;  
  
class OutOfRangeNull : public out\_of\_range {  
public:  
 explicit OutOfRangeNull()   
 : out\_of\_range("Numbers should be more than 0") {}  
};  
  
#endif

**Вывод**В ходе выполнения лабораторной работы была разработана программа для работы с двусвязным циклическим списком матриц, реализующая как иерархию классов для квадратных, единичных и прямоугольных матриц, так и полиморфизм у классов матриц. Программа включает в себя операции сложения, вычитания, умножения, транспонирования, а также функции для сохранения и загрузки матриц в текстовые и бинарные файлы; операции добавления узла в список, удаления узла из списка, поиска узла по индексу.

Создан класс Matrix, который служит базовым для производных классов SingleMatrix и RectangularMatrix. Этот базовый класс предоставляет методы для выполнения основных математических операций и управление данными матрицы. В производных классах были переопределены методы, обеспечивающие специфичное поведение для единичных и прямоугольных матриц, а также была добавлена возможность удобной работы с объектами этих классов через перегрузку операторов.

Создан класс Node – класс, в котором задаётся узел вместе со ссылками на предыдущие узлы и следующие.

Создан класс List – класс списка, где определены операции над списком.

Cозданы классы OutOfRangeLarge, OutOfRangeNegative, OutOfRangeNull для обработки исключений.