Задание 1

Выполнил:	`Захаревич Николай Сергеевич`
Имя входного файла:	`input.txt`
Имя выходного файла:	`Консоль`

- Создать класс матрица (должен быть конструктор и деструктор)
- Выполнить перегрузку операторов +/-/*/+=/-=/*= для матрицы (поэлементно) и скалярного значения
- Выполнить перегрузку оператора () для индексации элементов матрицы
- Добавить метод транспонирования матрицы
- Добавить метод dot для произведения матриц
- Выполнить перегрузку operator<<(>>). Для >> данные для матрицы читаем из файла

Формат входного файла

Входной файл содержит число n ($0 < n < 10^9$)- количество матриц, затем n матриц, заданных через размер $n_i, m_i (0 < n_i, m_i < 10^9)$, и $n_i \cdot m_i$ целых чисел. Затем идет число q - ($0 < q < 10^9$) количество запросов, относящихся к этим матрицам. Существует 5 типов запросов: a(i, j) - вывести заданный элемент матрицы a; a +=/-=/*= C - произвести операцию со всеми элементами матрицы a; a = b +/-/* C - присваивает a модифицированную b; a transpose - транспонирует a, a dot b - перемножает матрицы a и b

Формат выходного файла

Матрицы после исполнения всех запросов

Пример

input.txt	Консоль
2 2 3 1 1 1 2 2 2 2 3 4 4 4 4 4 4 4 1 transpose 0 dot 1 1 += 3 1 *= 2	12 12 24 24 14 14 14 14 14 14

Код программы

main.cpp

```
#include "matrix.h"
#include <vector>
#include <string>

int main()
{
  std::ifstream fin ("input.txt");
  unsigned int matrixAmount;
```

```
  fin >> matrixAmount;

  std::vector <Matrix> matrix(matrixAmount);
  for (unsigned int i = 0; i < matrixAmount; i++) {
    unsigned int nI, mI;
    fin >> nI >> mI;
    matrix[i] = Matrix(nI, mI);
    fin >> matrix[i];
  }

  unsigned int queriesAmount;
  fin >> queriesAmount;

  std::string query;
  std::getline(fin, query); // get '\n' symbol
  while (queriesAmount--) {
    std::getline(fin, query);
    if (query[1] == '(') { // a(i, j)
      std::cout << matrix[query[0] - '0']((query[2] - '0'), (query[4]));
    }
    if (query[2] == '=') { // a = b + C, a = b - C, a = b * C
      if (query[6] == '+') {
        matrix[query[0] - '0'] = matrix[query[4] - '0'] + (query[8] - '0');
      } else if (query[6] == '-') {
        matrix[query[0] - '0'] = matrix[query[4] - '0'] - (query[8] - '0');
      } else if (query[6] == '*') {
        matrix[query[0] - '0'] = matrix[query[4] - '0'] * (query[8] - '0');
      }
    } else if (query[3] == '=') { // a += C, a -= C, a *= C
      if (query[2] == '+') {
        matrix[query[0] - '0'] += (query[5] - '0');
      } else if (query[2] == '-') {
        matrix[query[0] - '0'] -= (query[5] - '0');
      } else if (query[2] == '*') {
        matrix[query[0] - '0'] *= (query[5] - '0');
      }
    } else if (query.find("transpose") != std::string::npos) { // a transpose
      matrix[query[0] - '0'].transpose();
    } else if (query.find("dot") != std::string::npos) { // a dot b
      matrix[query[0] - '0'].dot(matrix[query[6] - '0']);
    }
  }

  for (unsigned int i = 0; i < matrixAmount; i++) {
    std::cout << matrix[i] << "\n";
  }

  return 0;
}
```

matrix.h

```
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED

#include <iostream>
#include <ostream>
#include <fstream>
#include <cstdlib>

class Matrix {
private:
  unsigned int n, m;
  int **matrix;
  void checkSizeEquivalence(const Matrix &first, const Matrix &second);

public:
  Matrix();
  Matrix(unsigned int n, unsigned int m);
  Matrix(const Matrix &other);
  ~Matrix();

  int &operator()(unsigned int row, unsigned int col);
  int &operator()(unsigned int row, unsigned int col) const;
  Matrix &operator=(const Matrix &other);

  Matrix operator+(int value) const;
  Matrix operator-(int value) const;
  Matrix operator*(int value) const;

  Matrix operator+(const Matrix &other) const;
  Matrix operator-(const Matrix &other) const;
  Matrix operator*(const Matrix &other) const;

  Matrix &operator+=(int value);
  Matrix &operator-=(int value);
  Matrix &operator*=(int value);

  Matrix &operator+=(const Matrix &other);
  Matrix &operator-=(const Matrix &other);
  Matrix &operator*=(const Matrix &other);

  void transpose();
  void dot(const Matrix &other);

  friend std::ostream &operator<<(std::ostream &stream, const Matrix &outMatrix);
  friend std::ifstream &operator>>(std::ifstream &stream, const Matrix &inMatrix);
};

#endif // MATRIX_H_INCLUDED
```

matrix.cpp

```
#include "matrix.h"

Matrix::Matrix() {
  n = 0;
  m = 0;
}

Matrix::Matrix(unsigned int sizeN, unsigned int sizeM) {
  n = sizeN;
  m = sizeM;
  matrix = new int *[n];
  for (unsigned int row = 0; row < n; row++) {
    matrix[row] = new int [m];
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] = 0;
    }
  }
}

Matrix::Matrix(const Matrix &other) {
  n = other.n;
  m = other.m;
  matrix = new int *[n];
  for (unsigned int row = 0; row < n; row++) {
    matrix[row] = new int [m];
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] = other.matrix[row][col];
    }
  }
}

Matrix::~Matrix() {
  for (unsigned int row = 0; row < n; row++) {
    delete [] matrix[row];
  }
  delete [] matrix;
}

Matrix &Matrix::operator=(const Matrix &other) {
  this -> ~Matrix();
  new (this) Matrix(other);
  return *this;
}

int &Matrix::operator()(unsigned int row, unsigned int col) {
  if (row < n && col < m) {
    return matrix[row][col];
  } else {
    std::cout << "Bad index";
    exit(EXIT_FAILURE);
  }
}
```

```
int &Matrix::operator()(unsigned int row, unsigned int col) const{
  if (row < n && col < m) {
    return matrix[row][col];
  } else {
    std::cout << "Bad index";
    exit(EXIT_FAILURE);
  }
}

Matrix Matrix::operator+(int value) const {
  Matrix newMatrix = *this;
  newMatrix += value;
  return newMatrix;
}

Matrix Matrix::operator-(int value) const {
  Matrix newMatrix = *this;
  newMatrix -= value;
  return newMatrix;
}

Matrix Matrix::operator*(int value) const {
  Matrix newMatrix = *this;
  newMatrix *= value;
  return newMatrix;
}

Matrix Matrix::operator+(const Matrix &other) const {
  Matrix newMatrix = *this;
  newMatrix += other;
  return newMatrix;
}

Matrix Matrix::operator-(const Matrix &other) const {
  Matrix newMatrix = *this;
  newMatrix -= other;
  return newMatrix;
}

Matrix Matrix::operator*(const Matrix &other) const {
  Matrix newMatrix = *this;
  newMatrix *= other;
  return newMatrix;
}

Matrix &Matrix::operator+=(int value) {
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] += value;
    }
  }
```

```
  return *this;
}

Matrix &Matrix::operator-=(int value) {
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] -= value;
    }
  }
  return *this;
}

Matrix &Matrix::operator*=(int value) {
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] *= value;
    }
  }
  return *this;
}

void Matrix::checkSizeEquivalence(const Matrix &first, const Matrix &second) {
  if (first.n != second.n || first.m != second.m) {
    std::cout << "Bad size of matrix";
    exit(EXIT_FAILURE);
  }
}

Matrix &Matrix::operator+=(const Matrix &other) {
  checkSizeEquivalence(*this, other);
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] += other.matrix[row][col];
    }
  }
  return *this;
}

Matrix &Matrix::operator-=(const Matrix &other) {
  checkSizeEquivalence(*this, other);
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] -= other.matrix[row][col];
    }
  }
  return *this;
}

Matrix &Matrix::operator*=(const Matrix &other) {
  checkSizeEquivalence(*this, other);
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < m; col++) {
```

```
      matrix[row][col] *= other.matrix[row][col];
    }
  }
  return *this;
}

void Matrix::transpose() {
  Matrix temp = *this;
  this -> ~Matrix();

  n = temp.m;
  m = temp.n;
  matrix = new int *[n];
  for (unsigned int row = 0; row < n; row++) {
    matrix[row] = new int [m];
    for (unsigned int col = 0; col < m; col++) {
      matrix[row][col] = temp.matrix[col][row];
    }
  }
}

void Matrix::dot(const Matrix &other) {
  if (m != other.n) {
    std::cout << "Impossible to multiply matrices";
    exit(EXIT_FAILURE);
  }

  Matrix temp(n, other.m);
  for (unsigned int row = 0; row < n; row++) {
    for (unsigned int col = 0; col < other.m; col++) {
      for (unsigned int k = 0; k < m; k++) {
        temp.matrix[row][col] += matrix[row][k] * other.matrix[k][col];
      }
    }
  }
  (*this) = temp;
}

std::ostream &operator<<(std::ostream &stream, const Matrix &outMatrix) {
  for (unsigned int row = 0; row < outMatrix.n; row++) {
    for (unsigned int col = 0; col < outMatrix.m; col++) {
      stream << outMatrix.matrix[row][col] << " ";
    }
    stream << "\n";
  }
  return stream;
}

std::ifstream &operator>>(std::ifstream &stream, const Matrix &inMatrix) {
  for (unsigned int row = 0; row < inMatrix.n; row++) {
    for (unsigned int col = 0; col < inMatrix.m; col++) {
      stream >> inMatrix.matrix[row][col];
```

```
    }
  }
  return stream;
}
```