```
#ifndef LABO_1_MATRIX_H
#define LABO_1_MATRIX_H

#include <iostream>
#include "operations/Operation.h"

using DataType = unsigned int; // Data type of the matrix

/**
 * Matrix class
 * This class is used to represent a matrix.
 * @author Maxime Scharwath
 * @author Nicolas Crausaz
 */
class Matrix {
public:
    /**
     * Output Flux Operators for the Matrix Class
     * @param os The output stream
     * @param m The matrix to output
     * @return The output stream
     */
    friend std::ostream& operator<<(std::ostream& os, const Matrix& m);

    /**
     * Addition operator return a new matrix
     * @param other The matrix to add
     * @return the new matrix
     */
    friend Matrix add(const Matrix& lhs, const Matrix& rhs);

    /**
     * Addition operator dynamic version
     * @param other The matrix to add
     * @return the new matrix pointer
     */
    friend Matrix* addDyn(const Matrix& lhs, const Matrix& rhs);

    /**
     * Subtraction operator return a new matrix
     * @param other The matrix to substract
     * @return the new matrix
     */
    friend Matrix sub(const Matrix& lhs, const Matrix& rhs);

    /**
     * Subtraction operator dynamic version
     * @param other The matrix to substract
     * @return the new matrix pointer
     */
    friend Matrix* subDyn(const Matrix& lhs, const Matrix& rhs);

    /**
     * Multiplication operator return a new matrix
     * @param other The matrix to multiply
     * @return the new matrix
     */
    friend Matrix mult(const Matrix& lhs, const Matrix& rhs);

    /**
     * Multiplication operator dynamic version
     * @param other The matrix to multiply
     * @return the new matrix pointer
     */
    friend Matrix* multDyn(const Matrix& lhs, const Matrix& rhs);

    /**
     * Main Constructor for the Matrix Class
     * @param rows The number of rows
     * @param cols The number of columns
     * @param modulo The modulo to use
```

```
 72         */
 73        Matrix(unsigned int rows, unsigned int cols, unsigned int modulo);
 74
 75        /**
 76         * Square Constructor for the Matrix Class
 77         * @param size The size of the square matrix
 78         * @param modulo The modulo to use
 79         */
 80        Matrix(unsigned int size, unsigned int modulo);
 81
 82        /**
 83         * Allocator operator for the Matrix Class
 84         * @param other The matrix to copy
 85         * @return The new matrix
 86         */
 87        Matrix& operator=(const Matrix& other);
 88
 89        /**
 90         * Copy constructor for the Matrix Class
 91         * @param other The matrix to copy
 92         */
 93        Matrix(const Matrix& other);
 94
 95        /**
 96         * Destructor for the Matrix Class
 97         */
 98        ~Matrix();
 99
100        /**
101         * Addition operator edit the current matrix
102         * @param other The matrix to add
103         * @return the edited matrix
104         */
105        Matrix& add(const Matrix& other);
106
107        /**
108         * Subtraction operator edit the current matrix
109         * @param other The matrix to substract
110         * @return the edited matrix
111         */
112        Matrix& sub(const Matrix& other);
113
114        /**
115         * Multiplication operator edit the current matrix
116         * @param other The matrix to multiply
117         * @return the edited matrix
118         */
119        Matrix& mult(const Matrix& other);
120
121    private:
122        unsigned int rows, cols;
123        unsigned int modulo;
124        DataType** data;
125
126        /**
127         * Allocate a new matrix with random values
128         * @return the matrix data pointer
129         */
130        DataType** allocateMatrixData() const;
131
132        /**
133         * Allocate a new matrix with the values of another matrix
134         * @warning No check is done on the dimensions
135         * @param other The matrix to copy
136         * @return the matrix data pointer
137         */
138        DataType** allocateMatrixData(const Matrix& other) const;
139
140        /**
141         * Deallocate the matrix data
142         * @warning cols and rows must be set before
```

```
143         */
144        void deallocateMatrixData();
145
146        /**
147         * Initialize the matrix with values from another matrix
148         * @details Used by the copy constructor and the operator=
149         * @param other The matrix to copy
150         */
151        void initFrom(const Matrix& other);
152
153        /**
154         * Execute an operation on the current matrix
155         * @param operation The operation to execute
156         * @param other The matrix to use with
157         * @throw std::runtime_error if matrix modulo are different
158         */
159        void operation(const Operation<DataType>& operation, const Matrix& other);
160    };
161
162    #endif //LABO_1_MATRIX_H
163
```

```cpp
#include "Matrix.h"
#include "operations/AdditionOperation.h"
#include "operations/SubstrationOperation.h"
#include "operations/MultiplicationOperation.h"

// Friend

std::ostream& operator<<(std::ostream& os, const Matrix& m) {
    for (unsigned i = 0; i < m.rows; i++) {
        for (unsigned j = 0; j < m.cols; j++) {
            os << m.data[i][j] << " ";
        }
        os << std::endl;
    }
    return os;
}

Matrix add(const Matrix& lhs, const Matrix& rhs) {
    Matrix result(lhs);
    return result.add(rhs);
}

Matrix* addDyn(const Matrix& lhs, const Matrix& rhs) {
    Matrix* result = new Matrix(lhs);
    result->add(rhs);
    return result;
}

Matrix sub(const Matrix& lhs, const Matrix& rhs) {
    Matrix result(lhs);
    return result.sub(rhs);
}

Matrix* subDyn(const Matrix& lhs, const Matrix& rhs) {
    Matrix* result = new Matrix(lhs);
    result->sub(rhs);
    return result;
}

Matrix mult(const Matrix& lhs, const Matrix& rhs) {
    Matrix result(lhs);
    return result.mult(rhs);
}

Matrix* multDyn(const Matrix& lhs, const Matrix& rhs) {
    Matrix* result = new Matrix(lhs);
    result->mult(rhs);
    return result;
}

// Public

Matrix::Matrix(unsigned int rows, unsigned int cols, unsigned int modulo) :
        rows(rows), cols(cols), modulo(modulo) {
    // Verify params
    if (rows <= 0 || cols <= 0) {
        throw std::runtime_error("Matrix dimensions must be greater than 0");
    }

    if (modulo <= 0) {
        throw std::runtime_error("Matrix modulo must be greater than 0");
    }

    data = allocateMatrixData();
}

Matrix::Matrix(unsigned int size, unsigned int modulo) :
        Matrix(size, size, modulo) {}

Matrix::~Matrix() {
    deallocateMatrixData();
```

```
72  }
73
74  Matrix& Matrix::operator=(const Matrix& other) {
75      if (this != &other) {
76          deallocateMatrixData();
77          initFrom(other);
78      }
79      return *this;
80  }
81
82  Matrix::Matrix(const Matrix& other) {
83      initFrom(other);
84  }
85
86  // Private
87
88  void Matrix::initFrom(const Matrix& other) {
89      rows = other.rows;
90      cols = other.cols;
91      modulo = other.modulo;
92      data = allocateMatrixData(other);
93  }
94
95  DataType** Matrix::allocateMatrixData() const {
96      DataType** tmpData = new DataType* [rows];
97
98      for (unsigned i = 0; i < rows; ++i) {
99          tmpData[i] = new DataType[cols];
100         for (unsigned j = 0; j < cols; ++j) {
101             tmpData[i][j] = (DataType) (rand() / (RAND_MAX + 1.0) * modulo);
102         }
103     }
104     return tmpData;
105 }
106
107 DataType** Matrix::allocateMatrixData(const Matrix& other) const {
108     DataType** tmpData = new DataType* [rows];
109
110     for (unsigned i = 0; i < rows; ++i) {
111         tmpData[i] = new DataType[cols];
112         for (unsigned j = 0; j < cols; ++j) {
113             tmpData[i][j] = other.data[i][j];
114         }
115     }
116     return tmpData;
117 }
118
119 void Matrix::deallocateMatrixData() {
120     for (unsigned i = 0; i < rows; ++i) {
121         delete[] this->data[i];
122     }
123     delete[] data;
124 }
125
126 Matrix& Matrix::add(const Matrix& other) {
127     static AdditionOperation<DataType> op;
128     operation(op, other);
129     return *this;
130 }
131
132 Matrix& Matrix::sub(const Matrix& other) {
133     static SubstractionOperation<DataType> op;
134     operation(op, other);
135     return *this;
136 }
137
138 Matrix& Matrix::mult(const Matrix& other) {
139     static MultiplicationOperation<DataType> op;
140     operation(op, other);
141     return *this;
142 }
```

```
143
144  void Matrix::operation(const Operation<DataType>& operation, const Matrix& other) {
145      if (modulo != other.modulo) {
146          throw std::runtime_error("Matrices must have the same modulo");
147      }
148
149      unsigned maxRows = std::max(rows, other.rows);
150      unsigned maxCols = std::max(cols, other.cols);
151
152      DataType** tmp = new DataType* [maxRows];
153
154      for (unsigned i = 0; i < maxRows; ++i) {
155          tmp[i] = new DataType[maxCols];
156          for (unsigned j = 0; j < maxCols; ++j) {
157              DataType a = (i < rows && j < cols) ? data[i][j] : 0;
158              DataType b = (i < other.rows && j < other.cols) ? other.data[i][j] : 0;
159              tmp[i][j] = operation.execute(a, b) % modulo;
160          }
161      }
162      deallocateMatrixData();
163      data = tmp;
164      rows = maxRows;
165      cols = maxCols;
166  }
```

```
1   #ifndef LABO_1_OPERATION_H
2   #define LABO_1_OPERATION_H
3
4
5   /**
6    * Operation class
7    * @brief The Operation class is the base class for all operations.
8    * @tparam T
9    * @author Maxime Scharwath
10   * @author Nicolas Crausaz
11   */
12  template<typename T>
13  class Operation {
14  public:
15      /**
16       * execute the operation
17       * @param a - first operand
18       * @param b - second operand
19       * @return the result of the operation
20       */
21      virtual T execute(T a, T b) const = 0;
22  };
23
24  #endif //LABO_1_OPERATION_H
25
```

```
#ifndef LABO_1_ADDITIONOPERATION_H
#define LABO_1_ADDITIONOPERATION_H

#include "Operation.h"

/**
 * Addition operation.
 * @tparam T
 * @author Maxime Scharwath
 * @author Nicolas Crausaz
 */
template<typename T>
class AdditionOperation : public Operation<T> {
public:
    T execute(T a, T b) const override {
        return a + b;
    }
};

#endif //LABO_1_ADDITIONOPERATION_H

```

```
1   #ifndef LABO_1_SUBSTRATIONOPERATION_H
2   #define LABO_1_SUBSTRATIONOPERATION_H
3
4   #include "Operation.h"
5
6   /**
7    * Subtraction operation.
8    * @tparam T
9    * @author Maxime Scharwath
10   * @author Nicolas Crausaz
11   */
12  template<typename T>
13  class SubstractionOperation : public Operation<T> {
14  public:
15     T execute(T a, T b) const override {
16        return a - b;
17     }
18  };
19
20  #endif //LABO_1_SUBSTRATIONOPERATION_H
21
```

```
1   #ifndef LABO_1_MULTIPLICATIONOPERATION_H
2   #define LABO_1_MULTIPLICATIONOPERATION_H
3
4   #include "Operation.h"
5
6   /**
7    * Multiplication operation.
8    * @tparam T
9    * @author Maxime Scharwath
10   * @author Nicolas Crausaz
11   */
12  template<typename T>
13  class MultiplicationOperation : public Operation<T> {
14  public:
15      T execute(T a, T b) const override {
16          return a * b;
17      }
18  };
19
20  #endif //LABO_1_MULTIPLICATIONOPERATION_H
21
```

```cpp
#include <iostream>
#include <ctime>
#include "Matrix.h"

using namespace std;

/**
 * Unit test for Matrix class.
 */
void unit_tests() {
    const unsigned MOD = 8;
    cout << "TESTS" << endl;
    // TEST 1a
    cout << "TEST 1a" << endl;
    try {
        // Should throw
        Matrix mInvalidModulo2(2, 3, 0);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }

    // TEST 1b
    cout << "TEST 1b" << endl;
    try {
        // Should throw
        Matrix mInvalidModulo1(2, 0);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }

    // TEST 2a
    cout << "TEST 2a" << endl;
    try {
        // Should throw
        Matrix mInvalidRowsAndCols(0, 0, MOD);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }

    // TEST 2b
    cout << "TEST 2b" << endl;
    try {
        // Should throw
        Matrix mInvalidRows(0, 2, MOD);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }

    // TEST 2c
    cout << "TEST 2c" << endl;
    try {
        // Should throw
        Matrix mInvalidCols(2, 0, MOD);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }

    // TEST 2d
    cout << "TEST 2d" << endl;
    try {
        // Should throw
        Matrix mInvalidCols(0, MOD);
    }
    catch (const std::exception& e) {
        cout << e.what() << endl;
    }
```

```
 72
 73        // TEST 3a
 74        cout << "TEST 3a" << endl;
 75        Matrix validMatrix = Matrix(4, 5, MOD);
 76        cout << validMatrix << endl;
 77
 78        // TEST 3b
 79        cout << "TEST 3b" << endl;
 80        Matrix validSquareMatrix = Matrix(4, MOD);
 81        cout << validSquareMatrix << endl;
 82
 83        // TEST 4a
 84        cout << "TEST 4a" << endl;
 85        Matrix mOneRow(1, 2, MOD);
 86        cout << mOneRow << endl;
 87
 88        // TEST 4b
 89        cout << "TEST 4b" << endl;
 90        Matrix mOneCol(3, 1, MOD);
 91        cout << mOneCol << endl;
 92
 93        // TEST 5a
 94        cout << "TEST 5a" << endl;
 95        Matrix m1(3, 4, MOD);
 96        Matrix m2(m1);
 97        cout << m1 << endl << m2 << endl;
 98
 99        // TEST 5b
100        cout << "TEST 5b" << endl;
101        Matrix m3(3, MOD);
102        cout << m1 << endl << m3 << endl;
103        m1 = m3;
104        cout << m1 << endl << m3 << endl;
105
106        // TEST 6a
107        cout << "TEST 6a" << endl;
108        Matrix add1 = Matrix(4, 5, MOD);
109        Matrix toAdd1 = Matrix(4, 5, MOD);
110        cout << add1 << "+" << endl << toAdd1 << "=" << endl;
111        add1.add(toAdd1);
112        cout << add1 << endl;
113
114        // TEST 6b
115        cout << "TEST 6b" << endl;
116        Matrix add2 = Matrix(2, 4, MOD);
117        Matrix toAdd2 = Matrix(3, 2, MOD);
118        cout << add2 << "+" << endl << toAdd2 << "=" << endl << add2.add(toAdd2) << endl;
119
120        // TEST 6c
121        cout << "TEST 6c" << endl;
122        Matrix addCopy1 = Matrix(4, 5, MOD);
123        Matrix toAddCopy1 = Matrix(4, 5, MOD);
124        cout << addCopy1 << "+" << endl << toAddCopy1 << "=" << endl << add(addCopy1,
           toAddCopy1) << endl;
125
126        // TEST 6d
127        cout << "TEST 6d" << endl;
128        Matrix addCopy2 = Matrix(2, 4, MOD);
129        Matrix toAddCopy2 = Matrix(3, 2, MOD);
130        cout << addCopy2 << "+" << endl << toAddCopy2 << "=" << endl << add(addCopy2,
           toAddCopy2) << endl;
131
132        // TEST 6e
133        cout << "TEST 6e" << endl;
134        Matrix addDyn1 = Matrix(4, 5, MOD);
135        Matrix toAddDyn1 = Matrix(4, 5, MOD);
136        Matrix* dyn1 = addDyn(addDyn1, toAddDyn1);
137        cout << addDyn1 << "+" << endl << toAddDyn1 << "=" << endl << *dyn1 << endl;
138        delete dyn1;
139
140        // TEST 6f
```

```
141         cout << "TEST 6f" << endl;
142         Matrix addDyn2 = Matrix(2, 4, MOD);
143         Matrix toAddDyn2 = Matrix(3, 2, MOD);
144         Matrix* dyn2 = addDyn(addDyn2, toAddDyn2);
145         cout << addDyn2 << "+" << endl << toAddDyn2 << "=" << endl << *dyn2 << endl;
146         delete dyn2;
147
148         // TEST 7a
149         cout << "TEST 7a" << endl;
150         Matrix sub1 = Matrix(4, 5, MOD);
151         Matrix toSub1 = Matrix(4, 5, MOD);
152         cout << sub1 << "-" << endl << toSub1 << "=" << endl << endl;
153         sub1.sub(toSub1);
154         cout << sub1 << endl;
155
156         // TEST 7b
157         cout << "TEST 7b" << endl;
158         Matrix sub2 = Matrix(2, 4, MOD);
159         Matrix toSub2 = Matrix(3, 2, MOD);
160         cout << sub2 << "-" << endl << toSub2 << "=" << endl << endl;
161         sub2.sub(toSub2);
162         cout << sub2 << endl;
163
164         // TEST 7c
165         cout << "TEST 7c" << endl;
166         Matrix subCopy1 = Matrix(4, 5, MOD);
167         Matrix toSubCopy1 = Matrix(4, 5, MOD);
168         cout << subCopy1 << "-" << endl << toSubCopy1 << "=" << endl << sub(subCopy1,
            toSubCopy1) << endl;
169
170         // TEST 7d
171         cout << "TEST 7d" << endl;
172         Matrix subCopy2 = Matrix(2, 4, MOD);
173         Matrix toSubCopy2 = Matrix(3, 2, MOD);
174         cout << subCopy2 << "-" << endl << toSubCopy2 << "=" << endl << sub(subCopy2,
            toSubCopy2) << endl;
175
176         // TEST 7e
177         cout << "TEST 7e" << endl;
178         Matrix subDyn1 = Matrix(4, 5, MOD);
179         Matrix toSubDyn1 = Matrix(4, 5, MOD);
180         Matrix* dyn3 = subDyn(subDyn1, toSubDyn1);
181         cout << subDyn1 << "-" << endl << toSubDyn1 << "=" << endl << *dyn3 << endl;
182         delete dyn3;
183
184         // TEST 6f
185         cout << "TEST 6f" << endl;
186         Matrix subDyn2 = Matrix(2, 4, MOD);
187         Matrix toSubDyn2 = Matrix(3, 2, MOD);
188         Matrix* dyn4 = subDyn(subDyn2, toSubDyn2);
189         cout << subDyn2 << "-" << endl << toSubDyn2 << "=" << endl << *dyn4 << endl;
190         delete dyn4;
191
192         // TEST 8a
193         cout << "TEST 8a" << endl;
194         Matrix mult1 = Matrix(4, 5, MOD);
195         Matrix toMult1 = Matrix(4, 5, MOD);
196         cout << mult1 << "*" << endl << toMult1 << "=" << endl << endl;
197         mult1.mult(toMult1);
198         cout << mult1 << endl;
199
200         // TEST 8b
201         cout << "TEST 8b" << endl;
202         Matrix mult2 = Matrix(4, 5, MOD);
203         Matrix toMult2 = Matrix(4, 5, MOD);
204         cout << mult2 << "*" << endl << toMult2 << "=" << endl << endl;
205         mult2.mult(toMult2);
206         cout << mult2 << endl;
207
208         // TEST 8c
209         cout << "TEST 8c" << endl;
```

```
210         Matrix multCopy1 = Matrix(4, 5, MOD);
211         Matrix toMultCopy1 = Matrix(4, 5, MOD);
212         cout << multCopy1 << "*" << endl << toMultCopy1 << "=" << endl <<
            mult(multCopy1, toMultCopy1) << endl;
213
214         // TEST 8d
215         cout << "TEST 8d" << endl;
216         Matrix multCopy2 = Matrix(2, 4, MOD);
217         Matrix toMultCopy2 = Matrix(3, 2, MOD);
218         cout << multCopy2 << "*" << endl << toMultCopy2 << "=" << endl <<
            mult(multCopy2, toMultCopy2) << endl;
219
220         // TEST 8e
221         cout << "TEST 8e" << endl;
222         Matrix multDyn1 = Matrix(4, 5, MOD);
223         Matrix toMultDyn1 = Matrix(4, 5, MOD);
224         Matrix* dyn5 = multDyn(multDyn1, toMultDyn1);
225         cout << multDyn1 << "*" << endl << toMultDyn1 << "=" << endl << *dyn5 << endl;
226         delete dyn5;
227
228         // TEST 8f
229         cout << "TEST 8f" << endl;
230         Matrix multDyn2 = Matrix(2, 4, MOD);
231         Matrix toMultDyn2 = Matrix(3, 2, MOD);
232         Matrix* dyn6 = multDyn(multDyn2, toMultDyn2);
233         cout << multDyn2 << "*" << endl << toMultDyn2 << "=" << endl << *dyn6 << endl;
234         delete dyn6;
235     }
236
237     /**
238      * Main program entry point
239      * @author Maxime Scharwath
240      * @author Nicolas Crausaz
241      * @return
242      */
243     int main() {
244         srand(time(nullptr)); // Initialize random seed
245
246         const unsigned MOD = 5;
247
248         cout << "The modulus is " << MOD << endl;
249         cout << "one" << endl;
250         Matrix one = Matrix(3, 4, MOD);
251         cout << one << endl;
252
253         cout << "two" << endl;
254         Matrix two = Matrix(3, 5, MOD);
255         cout << two << endl;
256
257         cout << "one + two" << endl << add(one, two) << endl;
258
259         cout << "one - two" << endl << sub(one, two) << endl;
260
261         cout << "one x two" << endl << mult(one, two) << endl;
262
263         // More specific tests
264         unit_tests();
265         return 0;
266     }
267
```