```cpp
#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
     */
    Matrix(unsigned int rows, unsigned int cols, unsigned int modulo);

    /**
     * Square Constructor for the Matrix Class
     * @param size The size of the square matrix
     * @param modulo The modulo to use
     */
    Matrix(unsigned int size, unsigned int modulo);

    /**
     * Allocator operator for the Matrix Class
     * @param other The matrix to copy
     * @return The new matrix
     */
    Matrix& operator=(const Matrix& other);

    /**
     * Copy constructor for the Matrix Class
     * @param other The matrix to copy
     */
    Matrix(const Matrix& other);

    /**
     * Destructor for the Matrix Class
     */
    ~Matrix();

    /**
     * Addition operator edit the current matrix
     * @param other The matrix to add
     * @return the edited matrix
     */
    Matrix& add(const Matrix& other);

    /**
     * Subtraction operator edit the current matrix
     * @param other The matrix to substract
     * @return the edited matrix
     */
    Matrix& sub(const Matrix& other);

    /**
     * Multiplication operator edit the current matrix
     * @param other The matrix to multiply
     * @return the edited matrix
     */
    Matrix& mult(const Matrix& other);

private:
    unsigned int rows, cols;
    unsigned int modulo;
    DataType** data;

    /**
     * Allocate a new matrix with random values
     * @return the matrix data pointer
     */
    DataType** allocateMatrixData() const;

    /**
     * Allocate a new matrix with the values of another matrix
     * @warning No check is done on the dimensions
     * @param other The matrix to copy
     * @return the matrix data pointer
     */
    DataType** allocateMatrixData(const Matrix& other) const;

    /**
     * Deallocate the matrix data
     * @warning cols and rows must be set before
     */
    void deallocateMatrixData();

    /**
     * Initialize the matrix with values from another matrix
     * @details Used by the copy constructor and the operator=
     * @param other The matrix to copy
     */
    void initFrom(const Matrix& other);

    /**
     * Execute an operation on the current matrix
     * @param operation The operation to execute
     * @param other The matrix to use with
     * @throw std::runtime_error if matrix modulo are different
     */
    void operation(const Operation<DataType>& operation, const Matrix& other);
};

#endif //LABO_1_MATRIX_H

```

```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();
}

Matrix& Matrix::operator=(const Matrix& other) {
    if (this != &other) {
        deallocateMatrixData();
        initFrom(other);
    }
    return *this;
}

Matrix::Matrix(const Matrix& other) {
    initFrom(other);
}

// Private

void Matrix::initFrom(const Matrix& other) {
    rows = other.rows;
    cols = other.cols;
    modulo = other.modulo;
    data = allocateMatrixData(other);
}

DataType** Matrix::allocateMatrixData() const {
    DataType** tmpData = new DataType* [rows];

    for (unsigned i = 0; i < rows; ++i) {
        tmpData[i] = new DataType[cols];
        for (unsigned j = 0; j < cols; ++j) {
            tmpData[i][j] = (DataType) (rand() / (RAND_MAX + 1.0) * modulo);
        }
    }
    return tmpData;
}

DataType** Matrix::allocateMatrixData(const Matrix& other) const {
    DataType** tmpData = new DataType* [rows];

    for (unsigned i = 0; i < rows; ++i) {
        tmpData[i] = new DataType[cols];
        for (unsigned j = 0; j < cols; ++j) {
            tmpData[i][j] = other.data[i][j];
        }
    }
    return tmpData;
}

void Matrix::deallocateMatrixData() {
    for (unsigned i = 0; i < rows; ++i) {
        delete[] this->data[i];
    }
    delete[] data;
}

Matrix& Matrix::add(const Matrix& other) {
    static AdditionOperation<DataType> op;
    operation(op, other);
    return *this;
}

Matrix& Matrix::sub(const Matrix& other) {
    static SubstractionOperation<DataType> op;
    operation(op, other);
    return *this;
}

Matrix& Matrix::mult(const Matrix& other) {
    static MultiplicationOperation<DataType> op;
    operation(op, other);
    return *this;
}
```

```cpp

void Matrix::operation(const Operation<DataType>& operation, const Matrix& other) {
    if (modulo != other.modulo) {
        throw std::runtime_error("Matrices must have the same modulo");
    }

    unsigned maxRows = std::max(rows, other.rows);
    unsigned maxCols = std::max(cols, other.cols);

    DataType** tmp = new DataType* [maxRows];

    for (unsigned i = 0; i < maxRows; ++i) {
        tmp[i] = new DataType[maxCols];
        for (unsigned j = 0; j < maxCols; ++j) {
            DataType a = (i < rows && j < cols) ? data[i][j] : 0;
            DataType b = (i < other.rows && j < other.cols) ? other.data[i][j] : 0;
            tmp[i][j] = operation.execute(a, b) % modulo;
        }
    }
    deallocateMatrixData();
    data = tmp;
    rows = maxRows;
    cols = maxCols;
}
```

```cpp
#ifndef LABO_1_OPERATION_H
#define LABO_1_OPERATION_H


/**
 * Operation class
 * @brief The Operation class is the base class for all operations.
 * @tparam T
 * @author Maxime Scharwath
 * @author Nicolas Crausaz
 */
template<typename T>
class Operation {
public:
    /**
     * execute the operation
     * @param a - first operand
     * @param b - second operand
     * @return the result of the operation
     */
    virtual T execute(T a, T b) const = 0;
};

#endif //LABO_1_OPERATION_H

```

```
#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

```

```cpp
#ifndef LABO_1_SUBSTRATIONOPERATION_H
#define LABO_1_SUBSTRATIONOPERATION_H

#include "Operation.h"

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

#endif //LABO_1_SUBSTRATIONOPERATION_H

```

```
#ifndef LABO_1_MULTIPLICATIONOPERATION_H
#define LABO_1_MULTIPLICATIONOPERATION_H

#include "Operation.h"

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

#endif //LABO_1_MULTIPLICATIONOPERATION_H

```

```
#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;
    }

    // TEST 3a
    cout << "TEST 3a" << endl;
    Matrix validMatrix = Matrix(4, 5, MOD);
    cout << validMatrix << endl;

    // TEST 3b
    cout << "TEST 3b" << endl;
    Matrix validSquareMatrix = Matrix(4, MOD);
    cout << validSquareMatrix << endl;

    // TEST 4a
    cout << "TEST 4a" << endl;
    Matrix mOneRow(1, 2, MOD);
    cout << mOneRow << endl;

    // TEST 4b
    cout << "TEST 4b" << endl;
    Matrix mOneCol(3, 1, MOD);
    cout << mOneCol << endl;

    // TEST 5a
    cout << "TEST 5a" << endl;
    Matrix m1(3, 4, MOD);
    Matrix m2(m1);
    cout << m1 << endl << m2 << endl;

    // TEST 5b
    cout << "TEST 5b" << endl;
    Matrix m3(3, MOD);
    cout << m1 << endl << m3 << endl;
    m1 = m3;
    cout << m1 << endl << m3 << endl;

    // TEST 6a
    cout << "TEST 6a" << endl;
    Matrix add1 = Matrix(4, 5, MOD);
    Matrix toAdd1 = Matrix(4, 5, MOD);
    cout << add1 << "+" << endl << toAdd1 << "=" << endl;
    add1.add(toAdd1);
    cout << add1 << endl;

    // TEST 6b
    cout << "TEST 6b" << endl;
    Matrix add2 = Matrix(2, 4, MOD);
    Matrix toAdd2 = Matrix(3, 2, MOD);
    cout << add2 << "+" << endl << toAdd2 << "=" << endl << add2.add(toAdd2) << endl;

    // TEST 6c
    cout << "TEST 6c" << endl;
    Matrix addCopy1 = Matrix(4, 5, MOD);
    Matrix toAddCopy1 = Matrix(4, 5, MOD);
    cout << addCopy1 << "+" << endl << toAddCopy1 << "=" << endl << add(addCopy1, 
    toAddCopy1) << endl;

    // TEST 6d
    cout << "TEST 6d" << endl;
    Matrix addCopy2 = Matrix(2, 4, MOD);
    Matrix toAddCopy2 = Matrix(3, 2, MOD);
    cout << addCopy2 << "+" << endl << toAddCopy2 << "=" << endl << add(addCopy2, 
    toAddCopy2) << endl;

    // TEST 6e
    cout << "TEST 6e" << endl;
    Matrix addDyn1 = Matrix(4, 5, MOD);
    Matrix toAddDyn1 = Matrix(4, 5, MOD);
    Matrix* dyn1 = addDyn(addDyn1, toAddDyn1);
    cout << addDyn1 << "+" << endl << toAddDyn1 << "=" << endl << *dyn1 << endl;
    delete dyn1;

    // TEST 6f
```

```cpp
    cout << "TEST 6f" << endl;
    Matrix addDyn2 = Matrix(2, 4, MOD);
    Matrix toAddDyn2 = Matrix(3, 2, MOD);
    Matrix* dyn2 = addDyn(addDyn2, toAddDyn2);
    cout << addDyn2 << "+" << endl << toAddDyn2 << "=" << endl << *dyn2 << endl;
    delete dyn2;

    // TEST 7a
    cout << "TEST 7a" << endl;
    Matrix sub1 = Matrix(4, 5, MOD);
    Matrix toSub1 = Matrix(4, 5, MOD);
    cout << sub1 << "-" << endl << toSub1 << "=" << endl << endl;
    sub1.sub(toSub1);
    cout << sub1 << endl;

    // TEST 7b
    cout << "TEST 7b" << endl;
    Matrix sub2 = Matrix(2, 4, MOD);
    Matrix toSub2 = Matrix(3, 2, MOD);
    cout << sub2 << "-" << endl << toSub2 << "=" << endl << endl;
    sub2.sub(toSub2);
    cout << sub2 << endl;

    // TEST 7c
    cout << "TEST 7c" << endl;
    Matrix subCopy1 = Matrix(4, 5, MOD);
    Matrix toSubCopy1 = Matrix(4, 5, MOD);
    cout << subCopy1 << "-" << endl << toSubCopy1 << "=" << endl << sub(subCopy1, toSubCopy1) << endl;

    // TEST 7d
    cout << "TEST 7d" << endl;
    Matrix subCopy2 = Matrix(2, 4, MOD);
    Matrix toSubCopy2 = Matrix(3, 2, MOD);
    cout << subCopy2 << "-" << endl << toSubCopy2 << "=" << endl << sub(subCopy2, toSubCopy2) << endl;

    // TEST 7e
    cout << "TEST 7e" << endl;
    Matrix subDyn1 = Matrix(4, 5, MOD);
    Matrix toSubDyn1 = Matrix(4, 5, MOD);
    Matrix* dyn3 = subDyn(subDyn1, toSubDyn1);
    cout << subDyn1 << "-" << endl << toSubDyn1 << "=" << endl << *dyn3 << endl;
    delete dyn3;

    // TEST 6f
    cout << "TEST 6f" << endl;
    Matrix subDyn2 = Matrix(2, 4, MOD);
    Matrix toSubDyn2 = Matrix(3, 2, MOD);
    Matrix* dyn4 = subDyn(subDyn2, toSubDyn2);
    cout << subDyn2 << "-" << endl << toSubDyn2 << "=" << endl << *dyn4 << endl;
    delete dyn4;

    // TEST 8a
    cout << "TEST 8a" << endl;
    Matrix mult1 = Matrix(4, 5, MOD);
    Matrix toMult1 = Matrix(4, 5, MOD);
    cout << mult1 << "*" << endl << toMult1 << "=" << endl << endl;
    mult1.mult(toMult1);
    cout << mult1 << endl;

    // TEST 8b
    cout << "TEST 8b" << endl;
    Matrix mult2 = Matrix(4, 5, MOD);
    Matrix toMult2 = Matrix(4, 5, MOD);
    cout << mult2 << "*" << endl << toMult2 << "=" << endl << endl;
    mult2.mult(toMult2);
    cout << mult2 << endl;

    // TEST 8c
    cout << "TEST 8c" << endl;
    Matrix multCopy1 = Matrix(4, 5, MOD);
    Matrix toMultCopy1 = Matrix(4, 5, MOD);
    cout << multCopy1 << "*" << endl << toMultCopy1 << "=" << endl << mult(multCopy1, toMultCopy1) << endl;

    // TEST 8d
    cout << "TEST 8d" << endl;
    Matrix multCopy2 = Matrix(2, 4, MOD);
    Matrix toMultCopy2 = Matrix(3, 2, MOD);
    cout << multCopy2 << "*" << endl << toMultCopy2 << "=" << endl << mult(multCopy2, toMultCopy2) << endl;

    // TEST 8e
    cout << "TEST 8e" << endl;
    Matrix multDyn1 = Matrix(4, 5, MOD);
    Matrix toMultDyn1 = Matrix(4, 5, MOD);
    Matrix* dyn5 = multDyn(multDyn1, toMultDyn1);
    cout << multDyn1 << "*" << endl << toMultDyn1 << "=" << endl << *dyn5 << endl;
    delete dyn5;

    // TEST 8f
    cout << "TEST 8f" << endl;
    Matrix multDyn2 = Matrix(2, 4, MOD);
    Matrix toMultDyn2 = Matrix(3, 2, MOD);
    Matrix* dyn6 = multDyn(multDyn2, toMultDyn2);
    cout << multDyn2 << "*" << endl << toMultDyn2 << "=" << endl << *dyn6 << endl;
    delete dyn6;
}

/**
 * Main program entry point
 * @author Maxime Scharwath
 * @author Nicolas Crausaz
 * @return
 */
int main() {
    srand(time(nullptr)); // Initialize random seed

    const unsigned MOD = 5;

    cout << "The modulus is " << MOD << endl;
    cout << "one" << endl;
    Matrix one = Matrix(3, 4, MOD);
    cout << one << endl;

    cout << "two" << endl;
    Matrix two = Matrix(3, 5, MOD);
    cout << two << endl;

    cout << "one + two" << endl << add(one, two) << endl;

    cout << "one - two" << endl << sub(one, two) << endl;

    cout << "one x two" << endl << mult(one, two) << endl;

    // More specific tests
    unit_tests();
    return 0;
}

```