```
cmake_minimum_required(VERSION 3.10)
project(LabMatrix)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

include(FetchContent)
FetchContent_Declare(
        googletest
        URL https://github.com/google/googletest/archive/03597a01ee50ed33e9dfd640b249b4be3799d395.zip
)

FetchContent_MakeAvailable(googletest)

include_directories(.)

enable_testing()

add_executable(matrix src/Matrix.cpp src/Matrix.hpp src/main.cpp
        src/operations/Operation.hpp
        src/operations/Subtraction.cpp
        src/operations/Subtraction.hpp
        src/operations/Multiplication.cpp
        src/operations/Multiplication.hpp
        src/operations/Addition.cpp
        src/operations/Addition.hpp
        src/utils/utils.cpp
        src/utils/utils.hpp)

add_executable(testMatrix test/test.cpp src/Matrix.hpp src/Matrix.cpp
        src/operations/Addition.hpp
        src/operations/Addition.cpp
        src/operations/Operation.hpp
        src/operations/Subtraction.cpp
        src/operations/Subtraction.hpp
        src/operations/Multiplication.cpp
        src/operations/Multiplication.hpp
        src/utils/utils.hpp
        src/utils/utils.cpp)

set_target_properties(matrix PROPERTIES LINKER_LANGUAGE CXX)

target_link_libraries(
        testMatrix
        GTest::gtest_main
)

include(GoogleTest)

gtest_discover_tests(testMatrix)

if(MSVC)
    target_compile_options(matrix PRIVATE /W4 /WX)
else()
    target_compile_options(matrix PRIVATE -Wall -Wpedantic -Werror)
endif()
```

```cpp
#pragma once

#include <cstddef>

/**
 * @brief The Operation class represents a generic arithmetic operation.
 */
class Operation {
public:
    /**
     * @brief Destructor for Operation.
     */
    virtual ~Operation() = default;

    /**
     * @brief Calculates the result of the operation.
     * @param a The first operand.
     * @param b The second operand.
     * @return The result of the operation.
     */
    [[nodiscard]] virtual int calculate(const int& a,
                                        const int& b) const = 0;

protected:
    /**
     * @brief Default constructor for Operation.
     */
    Operation() = default;
};
```

```cpp
#pragma once
#include "Operation.hpp"

/**
 * @brief The Addition class represents an addition operation.
 */
class Addition : public Operation {
public:
    /**
     * @brief Default constructor for Addition.
     */
    Addition() = default;

    /**
     * @brief Calculates the result of addition.
     * @param a The first operand.
     * @param b The second operand.
     * @return The result of addition.
     */
    [[nodiscard]] int calculate(const int& a,
                                const int& b) const override;
};
```

```cpp
#include "Addition.hpp"

int Addition::calculate(const int& a,
                        const int& b)const {
   return a+b;
}
```

```
#pragma once
#include "Operation.hpp"

/**
 * @brief The Multiplication class represents a multiplication operation.
 */
class Multiplication : public Operation {
public:
    /**
     * @brief Default constructor for Multiplication.
     */
    Multiplication() = default;

    /**
     * @brief Calculates the result of multiplication.
     * @param a The first operand.
     * @param b The second operand.
     * @return The result of multiplication.
     */
    [[nodiscard]] int calculate(const int& a,
                                const int& b) const override;
};
```

```cpp
#include "Multiplication.hpp"

int Multiplication::calculate(const int& a,
                              const int& b) const {
   return a*b;
}
```

```cpp
#pragma once
#include "Operation.hpp"

/**
 * @brief The Subtraction class represents a subtraction operation.
 */
class Subtraction : public Operation {
public:
    /**
     * @brief Default constructor for Subtraction.
     */
    Subtraction() = default;

    /**
     * @brief Calculates the result of subtraction.
     * @param a The first operand.
     * @param b The second operand.
     * @return The result of subtraction.
     */
     [[nodiscard]] int calculate(const int& a,
                                 const int& b) const override;
};
```

```cpp
//
// Created by Quentin Surdez on 02.03.24.
//

#include "Subtraction.hpp"

int Subtraction::calculate(const int& a,
                                    const int& b) const {
   return a-b;
}
```

```cpp
#pragma once

#include <ostream>
#include <random>
#include <cmath>
#include "operations/Operation.hpp"

class Matrix {

private:
   unsigned int modulo;      /**< Modulo value for the matrix. */
   unsigned int row;         /**< Number of rows in the matrix. */
   unsigned int column;      /**< Number of columns in the matrix. */
   unsigned int** matrix;    /**< Pointer to the matrix data. */

   /**
    * @brief Helper function to get the value at a specific position in the matrix
    *        if it's in bounds.
    * @param matrixOfInterest Matrix to get the value from.
    * @param i Row index.
    * @param j Column index.
    * @return Value at the specified position, or 0 if out of bounds.
    */
   [[nodiscard]] static unsigned int getValueIfInBound(const Matrix &matrixOfInterest,
                                                       const unsigned int &i, const unsigned int &j);

   /**
    * @brief Performs an operation between two matrices and stores the result
    *       in a third matrix.
    * @param op2 Second matrix for the operation.
    * @param result Resultant matrix to store the operation result.
    * @param operation Operation to be performed.
    */
   void performOperationResult(const Matrix &op2, Matrix &result,
                               const Operation &operation) const;

   /**
    * @brief Helper function to get the maximum rows and columns between two
    *        matrices.
    * @param op2 Second matrix for comparison.
    * @param maxRow Maximum rows.
    * @param maxColumn Maximum columns.
    */
   void getMaxRowMaxColumn(const Matrix &op2, unsigned int &maxRow,
                           unsigned int &maxColumn) const;

public:
   /**
    * @brief Constructor to initialize a matrix with specified modulo, rows,
    *       and columns.
    * @param _modulo Modulo value for the matrix.
    * @param _row Number of rows in the matrix.
    * @param _column Number of columns in the matrix.
    */
   Matrix(const unsigned int &_modulo, const unsigned int &_row, const unsigned int &_column);

   /**
    * @brief Destructor to release memory allocated for the matrix.
    */
   ~Matrix();

   /**
    * @brief Copy constructor to perform deep copy of another matrix.
    * @param matrix1 Matrix to be copied.
    */
   Matrix(const Matrix& matrix1);

   /**
    * @brief Copy assignment operator to perform deep copy of another matrix.
    * @param op2 Matrix to be assigned.
    * @return Reference to the assigned matrix.
    */
```

```
    Matrix& operator=(const Matrix& op2);

    /**
     * @brief Move constructor.
     * @param moved_matrix Matrix to be moved.
     */
    Matrix(Matrix&& moved_matrix) noexcept;

    /**
     * @brief Move assignment operator.
     * @param moved_matrix Matrix to be moved.
     * @return Reference to the assigned matrix.
     */
    Matrix& operator=(Matrix&& moved_matrix) noexcept;

    /**
     * @brief Applies a static operation on a matrix with a given operation.
     * @param b Second matrix for the operation.
     * @param op Operation to be performed.
     * @return Resultant matrix after applying the operation.
     */
    Matrix applyOpStaticVal(const Matrix& b, const Operation& op);

    /**
      * @brief Applies an operation on the current matrix and modifies it.
      * @param b Second matrix for the operation.
      * @param op Operation to be performed.
      */
    void applyOpModify(const Matrix& b, const Operation& op);

    /**
      * @brief Applies a dynamic operation on a matrix with a given operation.
      * @param b Second matrix for the operation.
      * @param op Operation to be performed.
      * @return Pointer to the resultant matrix after applying the operation.
      */
    Matrix* applyOpDynaPtr(const Matrix& b, const Operation& op);

    /**
     * @brief Overloaded stream insertion operator to print the matrix.
     * @param os Output stream.
     * @param matrix Matrix to be printed.
     * @return Reference to the output stream.
     */
    friend std::ostream& operator<<(std::ostream& os, const Matrix& matrix);

};
```

```cpp
#include "Matrix.hpp"
#include <random>
#include <iostream>
#include "operations/Operation.hpp"
#include "utils/utils.hpp"

std::ostream &operator<<(std::ostream &os, const Matrix &matrix) {

   for (unsigned int i = 0; i < matrix.row; i++) {
      for (unsigned int j = 0; j < matrix.column; j++) {
         os << matrix.matrix[i][j] << " ";
      }
      os << std::endl;
   }
   return os;
}

Matrix::Matrix(const unsigned int &_modulo, const unsigned int &_row, const unsigned int &_column) :
modulo(_modulo),
                                                                  row(_row),
                                                                  column(_column) {
   if (_modulo <= 0 || _row <= 0 || _column <= 0) {
      throw std::runtime_error(
                "The modulo, row or column cannot be equal or lower than zero"
                );
      return;
   }

   try {
      // Allocate memory for the matrix
      matrix = new unsigned int *[row];
      matrix[0] = new unsigned int[row * column];
      for (unsigned int i = 0; i < row; i++) {
         matrix[i] = matrix[0] + i * column;
      }
   } catch (const std::bad_alloc& e) { // todo neccessary or we don't care ?
      // Memory allocation failed, handle the error here
      std::cerr << "Memory allocation failed: " << e.what() << std::endl;
      throw;
   }
   // Initialize matrix with random values
   std::random_device rd;
   std::mt19937 gen(rd()); // Mersenne_twister_engine init with rd()
   std::uniform_int_distribution<> distrib(0, static_cast<int>(modulo) - 1);
   for (unsigned int i = 0; i < row; i++) {
      for (unsigned int j = 0; j < column; j++) {
         matrix[i][j] = static_cast<unsigned int>(distrib(gen));
      }
   }

}

// The new value 1 is assigned to ensure a valid state of the object after the move.
// This allows the user to potentially reuse the object, although not recommended.
Matrix::Matrix(Matrix &&moved_matrix) noexcept
                           : modulo{std::exchange(moved_matrix.modulo, 1)},
                              row{std::exchange(moved_matrix.row, 1)},
                              column{std::exchange(moved_matrix.column,1)},
                              matrix{std::exchange(moved_matrix.matrix, nullptr)} {}

Matrix &Matrix::operator=(Matrix &&moved_matrix) noexcept {
   if (this == &moved_matrix) {
      return *this;
   }

   // Free the matrix - Neo is that u ? □_□ // todo check pls
   if (matrix != nullptr) {
      if (matrix[0] != nullptr) {
         delete[] matrix[0];
      }
      delete[] matrix;
   }
```

```
   // Assign new values and ensure moved_matrix is in a valid state
   modulo = std::exchange(moved_matrix.modulo, 1);
   row = std::exchange(moved_matrix.row, 1);
   column = std::exchange(moved_matrix.column, 1);
   matrix = std::exchange(moved_matrix.matrix, nullptr);

   return *this;
}

Matrix::~Matrix() {
   // todo check pls
   if (matrix != nullptr) {
      if (matrix[0] != nullptr) {
         delete[] matrix[0];
      }
      delete[] matrix;
   }

}

Matrix::Matrix(const Matrix &matrix1) : modulo(matrix1.modulo),
                                        row(matrix1.row),
                                        column(matrix1.column) {
   // Allocate memory
   matrix = new unsigned int *[row];
   matrix[0] = new unsigned int[row * column];
   for (unsigned int i = 0; i < row; i++) {
      matrix[i] = matrix[0] + i * column;
   }
   // Copy values from matrix1
   for (unsigned int i = 0; i < row; i++) {
      for (unsigned int j = 0; j < column; j++) {
         matrix[i][j] = matrix1.matrix[i][j];
      }
   }
}

Matrix &Matrix::operator=(const Matrix &op2) {
   if (this == &op2) {
      return *this;
   }
   this->row = op2.row;
   this->column = op2.column;
   this->modulo = op2.modulo;

   // Allocate memory
   matrix = new unsigned int *[row];
   matrix[0] = new unsigned int[row * column];
   for (unsigned int i = 0; i < row; i++) {
      matrix[i] = matrix[0] + i * column;
   }
   // Copy values from op2
   for (unsigned int i = 0; i < row; i++) {
      for (unsigned int j = 0; j < column; j++) {
         this->matrix[i][j] = op2.matrix[i][j];
      }
   }

   return *this;
}

void Matrix::performOperationResult(const Matrix &op2, Matrix &result,
                                    const Operation &operation) const {
   if (this->modulo != op2.modulo) {
      throw std::invalid_argument("The modulii of the matrix must be equal");
      return;
   }
   // Perform an operation between two matrices
   // and stores the result in a third matrix
   for (unsigned int i = 0; i < result.row; i++) {
      for (unsigned int j = 0; j < result.column; j++) {
```

```
            unsigned int num1 = getValueIfInBound(*this, i, j);
            unsigned int num2 = getValueIfInBound(op2, i, j);
            unsigned int temp_int = floorMod(operation.calculate(
               static_cast<int>(num1),
               static_cast<int>(num2)
            ), static_cast<int>(result.modulo));
            result.matrix[i][j] = temp_int;
         }
      }
   }

Matrix Matrix::applyOpStaticVal(const Matrix &b, const Operation &op) {
   unsigned int maxRows, maxColumns;
   getMaxRowMaxColumn(b, maxRows, maxColumns);

   // Create a Matrix object with statically allocated memory
   Matrix result = Matrix(modulo, maxRows, maxColumns);
   // Perform the operation and store the result
   performOperationResult(b, result, op);

   return result;
}

Matrix *Matrix::applyOpDynaPtr(const Matrix &b, const Operation &op) {
   unsigned int maxRows, maxColumns;
   getMaxRowMaxColumn(b, maxRows, maxColumns);
   // Create a Matrix object with dynamically allocated memory
   auto *result = new Matrix(modulo, maxRows, maxColumns);
   // Perform the operation and store the result
   performOperationResult(b, *result, op);

   return result;
}

void Matrix::applyOpModify(const Matrix &b, const Operation &op) {
   unsigned int maxRows, maxColumns;
   getMaxRowMaxColumn(b, maxRows, maxColumns);
   // Create a temporary Matrix object with dynamically allocated memory
   auto *result = new Matrix(modulo, maxRows, maxColumns);
   // Perform the operation and store the result
   performOperationResult(b, *result, op);
   // Move the result to the current matrix
   *this = std::move(*result);
}

void Matrix::getMaxRowMaxColumn(const Matrix &op2, unsigned int &maxRow,
                                unsigned int &maxColumn) const {
   maxRow = std::max(row, op2.row);
   maxColumn = std::max(column, op2.column);

}

unsigned int Matrix::getValueIfInBound(const Matrix &matrixOfInterest,
                                       const unsigned int &i, const unsigned int &j) {
   return (matrixOfInterest.row <= i || matrixOfInterest.column <= j) ?
            0 : matrixOfInterest.matrix[i][j];
}
```

```cpp
#include "utils.hpp"

unsigned int floorMod(const int &number, const int &mod){
    int result = number % mod;
    if (result < 0){
        return static_cast<unsigned int>(result + mod);
    }
    return static_cast<unsigned int>(result);
}
```

```
#pragma once

#include <cstdlib>

/**
 * Caclulate number % mod and ensure that the result is positive
 * @param number the number on which to apply the modulo
 * @param mod the modulo
 * @return number % mod (>= 0)
 */
unsigned int floorMod(const int &number, const int &mod);
```