File - C:\Heig-vd\POA\POA_L1\src\Add.cpp

```cpp
#include "Add.h"

int Add::calculate(unsigned a, unsigned b) const {
    return a + b;
}

```

```cpp
//
// Created by André on 10.03.2022.
//

#ifndef POA_L1_ADD_H
#define POA_L1_ADD_H


#include "Operation.h"

/**
 * Add operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Add : public Operation{
public:


    /**
     * Add two numbers together
     *
     * @param a The first number to add.
     * @param b The second operand.
     * @return The result of the addition.
     */
    int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_ADD_H

```

```cpp
#include <iostream>
#include <sstream>
#include "Matrix.h"
using std::cout;
using std::endl;

/**
 * Test the matrix class
 *
 * @param n1 number of rows of the first matrix
 * @param m1 the number of columns of the first matrix
 * @param n2 number of rows of the second matrix
 * @param m2 the number of columns of the second matrix
 * @param n the modulo of the matrix
 */
void testMatrix(unsigned n1, unsigned m1, unsigned n2, unsigned m2, unsigned n){
    cout << "Test initialisation" << endl;

    Matrix* matrix1 = new Matrix(n1, m1, n);

    Matrix* matrix2 = new Matrix(n2, m2, n);

    Matrix* matrix3 = new Matrix(n1+1, m1+1, n);

    Matrix* matrix4 = new Matrix(n1, m1, n+1);

    cout << "matrix1 : " << endl << matrix1 << endl;
    cout << "matrix2 : " << endl << matrix2 << endl;
    cout << "matrix3 : " << endl << matrix3 << endl;
    cout << "matrix4 : " << endl << matrix4 << endl;

    cout << "test constructor copy : " << endl;
    Matrix c(*matrix1);
    cout << "c : " << endl << c << endl;

    cout << "test operator = : " << endl;
    Matrix d = *matrix2;
    cout << "d : " << endl << d << endl;

    cout << "test operator : " << endl;

    cout << "ADD with same modulus and size : " << endl;
    cout << "Inline : " << endl;

    matrix1->add(*matrix2);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->addByValue(*matrix2);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    c = matrix1->addByPtr(*matrix2);
    cout << "d : " << endl << d << endl;

    cout << "ADD with same modulus and different size : " << endl;
    cout << "Inline : " << endl;
    matrix1->add(*matrix3);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->addByValue(*matrix3);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    d = matrix1->addByPtr(*matrix3);
    cout << "d : " << endl << d << endl;

    /*cout << "ADD with different modulus : " << endl;
    cout << "Inline : " << endl;
    matrix1->add(*matrix4);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->addByValue(*matrix4);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    d = matrix1->addByPtr(*matrix4);
    cout << "d : " << endl << d << endl;*/

    cout << "SUB with same modulus and size : " << endl;
    cout << "Inline : " << endl;
    matrix1->sub(*matrix2);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->subByValue(*matrix2);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    c = matrix1->subByPtr(*matrix2);
    cout << "d : " << endl << d << endl;

    cout << "SUB with same modulus and different size : " << endl;
    cout << "Inline : " << endl;
    matrix1->sub(*matrix3);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->subByValue(*matrix3);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    d = matrix1->subByPtr(*matrix3);
    cout << "d : " << endl << d << endl;

    /*cout << "SUB with different modulus : " << endl;
    cout << "Inline : " << endl;
    matrix1->sub(*matrix4);
```

```cpp
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->subByValue(*matrix4);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    d = matrix1->subByPtr(*matrix4);
    cout << "d : " << endl << d << endl;*/

    cout << "MULTIPLY with same modulus and size : " << endl;
    cout << "Inline : " << endl;
    matrix1->mult(*matrix2);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->multByValue(*matrix2);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    c = matrix1->multByPtr(*matrix2);
    cout << "d : " << endl << d << endl;

    cout << "MULTIPLY with same modulus and different size : " << endl;
    cout << "Inline : " << endl;
    matrix1->mult(*matrix3);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->multByValue(*matrix3);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    d = matrix1->multByPtr(*matrix3);
    cout << "d : " << endl << d << endl;

    /*cout << "MULTIPLY with different modulus : " << endl;
    cout << "Inline : " << endl;
    matrix1->mult(*matrix4);
    cout << "matrix1 : " << endl << matrix1 << endl;
    matrix1 = &c;
    cout << "by value : " << endl;
    d = matrix1->multByValue(*matrix4);
    cout << "d : " << endl << d << endl;
    cout << "by pointer : " << endl;
    c = matrix1->multByPtr(*matrix4);
    cout << "d : " << endl << d << endl;*/

    delete matrix1;
    delete matrix2;
    delete matrix3;
    delete matrix4;
}

int main(int argc, char* argv[]) {

    if(argc != 6){
        std::cerr << "Usage: " << argv[0] << " N1 M1 N2 M2 N" << std::endl;
        return 1;
    }

    std::istringstream issn1(argv[1]), issm1(argv[2]), issn2(argv[3]), issm2(argv[4]), issn(argv[5]);
    unsigned n1, m1, n2, m2, n;

    if(issn1 >> n1 and issm1 >> m1 and issn2 >> n2 and issm2 >> m2 and issn >> n) {
        testMatrix(n1, m1, n2, m2, n);
        return 0;
    }
    std::cerr << "Error: " << argv[0] << " Arguments are not integers" << std::endl;
    return 1;
}
```

```cpp
#include "Matrix.h"
#include "iostream"
#include "Add.h"
#include "Sub.h"
#include "Multiply.h"
#include "Random.h"
#include <cmath>


static const Add ADD = Add();
static const Sub SUB = Sub();
static const Multiply MUL = Multiply();

Matrix::Matrix() {
    this->n = 0;
    this->m = 0;
    this->modulo = 0;
    this->data = nullptr;
}


Matrix::Matrix(unsigned int n, unsigned int m, unsigned int modulo, bool initRandom) {
    if(n < MIN_N)
        throw std::runtime_error("number of rows is out of bounds");
    if(m < MIN_M)
        throw std::runtime_error("number of columns is out of bounds");
    if(modulo < MIN_MODULO)
        throw std::runtime_error("modulus is out of bounds");

    Random* rand = Random::getInstance();
    this->n = n;
    this->m = m;
    this->modulo = modulo;
    this->data = new unsigned* [m];

    for (int i = 0; i < m; ++i) {
        this->data[i] = new unsigned [n];
    }
    if(initRandom) {
        for (int i = 0; i < this->n; ++i) {
            for (int j = 0; j < this->m; ++j) {
                this->data[i][j] = mod(rand->getRandom(this->modulo), this->modulo);
            }
        }
    }
}

Matrix::Matrix(unsigned int n, unsigned int m, unsigned int modulo) : Matrix(n, m, modulo, true) {}

```

```cpp
Matrix::Matrix(const Matrix &other) : Matrix(other.n, other.m, other.modulo, false){
    std::cout << "copy Matrix" << std::endl;
    for (unsigned i = 0; i < this->m; ++i) {
        for (unsigned j = 0; j < this->n; ++j) {
            this->data[i][j] = other.data[i][j];
        }
    }
}

/**
 * Deletes the dynamically allocated memory for the data member
 */
Matrix::~Matrix() {
    deleteValues();
}

void Matrix::deleteValues(){
    for (int i = 0; i < m; ++i) {
        delete[] this->data[i];
    }
    delete[] this->data;
}

std::ostream &operator<<(std::ostream &os, const Matrix &dt) {
    for (int i = 0; i < dt.m; ++i) {
        for (int j = 0; j < dt.n; ++j) {
            os << dt.data[i][j];
            if(j + 1 < dt.n)
                os << " ";
        }
        os << std::endl;
    }
    return os;
}

std::ostream &operator<<(std::ostream &os, Matrix* dt) {
    os << *dt;
    return os;
}

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

Matrix & Matrix::operator=(const Matrix *other) {

    if(other != this){
        // We use a temporary variable to not leave the object in a broken state
        // in case the allocation throws an exception.
        unsigned** tmpData = new unsigned* [other->n];
        for (int i = 0; i < other->n; ++i) {
            tmpData[i] = new unsigned [other->m];
        }

        deleteValues();

        this->n = other->n;
        this->m = other->m;
        this->modulo = other->modulo;

        this->data = tmpData;
        for (int i = 0; i < this->n; ++i) {
            for (int j = 0; j < this->m; ++j) {
                this->data[i][j] = other->data[i][j];
            }
        }
    }

    return *this;
}

unsigned int Matrix::getValueOrZero(unsigned i, unsigned j) const {
    return i < this->n && j < this->m ? this->data[i][j] : 0;
}

Matrix& Matrix::operation(const Matrix &other, const Operation &op) {
    if(other.modulo != this->modulo)
        throw std::invalid_argument("Error : Not the same modulus");

    unsigned newN = std::max(this->n, other.n);
    unsigned newM = std::max(this->m, other.m);

    unsigned** tmpData = new unsigned* [newN];
    for (int i = 0; i < newN; ++i) {
        tmpData[i] = new unsigned [newM];
    }

    for (unsigned i = 0; i < newM; ++i) {
        for (unsigned j = 0; j < newN; ++j) {
            tmpData[i][j] = mod(op.calculate(this->getValueOrZero(i,j), other.getValueOrZero(i,j)),modulo);
        }
    }

    this->n = newN;
    this->m = newM;

    deleteValues();
    this->data = tmpData;
    return *this;
}

Matrix *Matrix::operationByPtr(const Matrix &other, const Operation &op) const {
    if(other.modulo != this->modulo)
        throw std::invalid_argument("Error : Not the same modulus");

    Matrix* res = new Matrix(std::max(this->n, other.n), std::max(this->m, other.m), this->modulo, false);

    for (unsigned i = 0; i < n; ++i) {
        for (unsigned j = 0; j < m; ++j) {
            res->setValue(i, j, op.calculate(this->getValueOrZero(i,j), other.getValueOrZero(i,j)));
        }
    }

    return res;
}

Matrix Matrix::operationByValue(const Matrix &other, const Operation &op) const {
    if(other.modulo != this->modulo)
        throw std::invalid_argument("Error : Not the same modulus");

    Matrix res = Matrix(std::max(this->n, other.n), std::max(this->m, other.m), this->modulo, false);

    for (unsigned i = 0; i < n; ++i) {
        for (unsigned j = 0; j < m; ++j) {
            res.setValue(i, j, op.calculate(this->getValueOrZero(i,j), other.getValueOrZero(i,j)));
        }
    }

    return res;
}

void Matrix::setValue(unsigned int i, unsigned int j, unsigned int value) {
    if(i >= n || j >= m)
        throw std::runtime_error("Error out of bounds");

    data[i][j] = mod(value, modulo);
}

Matrix& Matrix::add(const Matrix &other) {
    return this->operation(other, ADD);
}

Matrix Matrix::addByValue(const Matrix &other) const {
    return this->operationByValue(other, ADD);
}

Matrix *Matrix::addByPtr(const Matrix &other) const {
    return this->operationByPtr(other, ADD);
}
```

```

Matrix& Matrix::sub(const Matrix &other) {
    return this->operation(other, SUB);
}

Matrix Matrix::subByValue(const Matrix &other) const {
    return this->operationByValue(other, SUB);
}

Matrix *Matrix::subByPtr(const Matrix &other) const {
    return this->operationByPtr(other, SUB);
}

Matrix& Matrix::mult(const Matrix &other) {
    return this->operation(other, SUB);
}

Matrix Matrix::multByValue(const Matrix &other) const {
    return this->operationByValue(other, MUL);
}

Matrix *Matrix::multByPtr(const Matrix &other) const {
    return this->operationByPtr(other, MUL);
}

unsigned int Matrix::mod(int a, int b){
   return (unsigned int) std::abs(a - ((floor((a / b))) * b));
}

```

```
#ifndef POA_L1_MATRIX_H
#define POA_L1_MATRIX_H

#include "iostream"
#include "Operation.h"

/**
* Matrix of unsigned int.
*
* @author Nelson Jeanrenaud
* @author André Marques Nora
* @version 1.0
*/
class Matrix {

   /**
    * Minimal value for n
    */
   static const int MIN_N = 1;

   /**
    * Minimal value for m
    */
   static const int MIN_M = 1;

   /**
    * Minimal value for modulo
    */
   static const int MIN_MODULO = 1;

   /**
   * the number of rows in the matrix
   */
   unsigned int n;

   /**
   * the number of columns in the matrix
   */
   unsigned int m;

   /**
   * Modulo for the numbers contained in the matrix
   */
   unsigned int modulo;

   /**
   * Array containing the numbers of the matrix
   */
   unsigned int** data;

   Matrix();

   /**
```

```
    * Create a matrix with n rows and m columns
    *
    * @param n the number of rows in the matrix
    * @param m the number of rows in the matrix
    * @param modulo the modulo for the numbers contained in the matrix
    * @param initRandom if true, the matrix will be initialized with random values.
    */
    Matrix(unsigned int n, unsigned int m, unsigned int modulo, bool initRandom);

    /**
    * If the indices are valid, return the value at the specified indices. Otherwise, return 0
    *
    * @param i The row of the matrix.
    * @param j The column index.
    * @return The value of the element at position (i, j) or zero if the position is out of bounds.
    */
    unsigned int getValueOrZero(unsigned i, unsigned j) const;

    /**
    * Destroy the value array in memory
    */
    void deleteValues();

    /**
    * Methode to calculate the modulus of two given numbers
    * @param a The first operand
    * @param b The second operand
    * @return Unsigned int representing the result of the modulus' calcul
    */
    static unsigned int mod(int a, int b);

public:

    /**
    * Operator for the stream output
    * @param os The output stream
    * @param dt A matrix
    * @return A stream output corresponding to a matrix
    */
    friend std::ostream& operator<<(std::ostream& os, const Matrix& dt);

    /**
    * Operator for the stream output
    * @param os The output stream
    * @param dt A pointer to a matrix
    * @return A stream output corresponding to a matrix
    */
    friend std::ostream& operator<<(std::ostream& os, Matrix* dt);

    /**
    * Operator = for a matrix
    * @param other A matrix
    * @return A Matrix
    */
    Matrix& operator= (const Matrix& other);

    /**
    * Operator = for a matrix
    * @param other A pointer to a matrix
    * @return A Matrix
    */
    Matrix& operator= (const Matrix *other);

    /**
     * Create a matrix with n rows and m columns
     *
     * @param n the number of rows in the matrix
     * @param m the number of rows in the matrix
     * @param modulo the modulo for the numbers contained in the matrix
     */
    Matrix(unsigned n, unsigned m, unsigned modulo);

    /**
    * The function creates a new matrix with the same dimensions as the other matrix and copies the values
    * from the other matrix into the new matrix
    *
    * @param other the matrix to copy
    */
    Matrix(const Matrix& other);

    /**
    * Matrix destructor
    */
    virtual ~Matrix();

    /**
    * This function adds the values of the two matrices and returns the result
    *
    * @param other The matrix to add to this one.
    * @return a reference to this object
    */
    Matrix& add(const Matrix& other);

    /**
    * Add two matrices together by value
    *
    * @param other The matrix to add to the current matrix.
    * @return The result of the addition operation.
    */

    Matrix addByValue(const Matrix& other) const;
    /**
    * This function adds two matrices together and returns a pointer to the resulting matrix
    * The resulting matrix is created dynamically
    *
    * @param other the other matrix to be added to this matrix.
    * @return A new Matrix object.
    */
    Matrix* addByPtr(const Matrix& other) const;

    /**
    * Subtracts the other matrix from this matrix
    *
    * @param other The matrix to subtract from this matrix.
    * @return a reference to this object.
    */
    Matrix& sub(const Matrix& other);

    /**
    * Subtracts the values of the other matrix from the values of this matrix
    *
    * @param other The matrix to subtract from this matrix.
    * @return The result of the subtraction operation.
    */
    Matrix subByValue(const Matrix& other) const;

    /**
    * This function subtracts the values of the other matrix from the values of this matrix and returns a
    * new dynamically allocated matrix
    *
    * @param other the matrix to subtract from this matrix
    * @return A new Matrix object.
    */
    Matrix* subByPtr(const Matrix& other) const;

    /**
    * Multiply the matrix by another matrix
    *
    * @param other The matrix to subtract from this matrix.
    * @return a reference to this object.
    */
    Matrix& mult(const Matrix& other);

    /**
     * Multiply the matrix by a another matrix
     *
     * @param other The matrix to multiply by.
     * @return The result of the multiplication.
     */
    Matrix multByValue(const Matrix& other) const;

    /**
     * Multiply the matrix by another matrix
     *
     * @param other the matrix to multiply by
     * @return A pointer to a new Matrix object.
     */
    Matrix* multByPtr(const Matrix& other) const;

    /**
    * Given a matrices, this function returns the result of the operation between the matrix and this
    *
    * @param other the matrix to be added
    * @param op the operation to perform
    * @return A reference to this object
    */
    Matrix& operation(const Matrix& other, const Operation& op);

    /**
    * Given a matrices, this function returns the result of the operation between the matrix and this
    *
    * @param other the matrix to be added
    * @param op the operation to perform
    * @return A pointer to a new Matrix object.
    */
    Matrix* operationByPtr(const Matrix& other, const Operation& op) const;

    /**
    * Given a matrices, this function returns the result of the operation between the matrix and this
    *
    * @param other the matrix to be added
    * @param op the operation to perform
    * @return A new Matrix object.
    */
    Matrix operationByValue(const Matrix& other, const Operation& op) const;

    /**
    * Set the value of the element at row i and column j to value
    *
    * @param i The row of the matrix.
    * @param j The column index of the value to be set.
    * @param value the value to be set
    */
    void setValue(unsigned i, unsigned j, unsigned value);
};
```

```


#endif //POA_L1_MATRIX_H

```

```cpp
#include "Multiply.h"

int Multiply::calculate(unsigned a, unsigned b) const {
    return a * b;
}

```

```
#ifndef POA_L1_MULTIPLY_H
#define POA_L1_MULTIPLY_H

#include "Operation.h"

/**
 * Multiplication operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Multiply : public Operation{
public:

   /**
    * Multiply the two unsigned integers
    *
    * @param a The first number to multiply.
    * @param b The number of times to multiply a by b.
    * @return The result of the multiplication.
    */
   int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_MULTIPLY_H

```

```cpp
#ifndef POA_L1_OPERATION_H
#define POA_L1_OPERATION_H

/**
 * Operation abstract class.
 * Extended to calculate an operation between two unsigned int.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Operation {
public:

    /**
     * Calculate the operation between two unsigned int.
     * @param a first operand.
     * @param b second operand..
     * @return result of the operation
     */
   virtual int calculate(unsigned a, unsigned b) const = 0;
};


#endif //POA_L1_OPERATION_H

```

```cpp
#include "Random.h"
#include <ctime>

Random* Random::instance = nullptr;
/**
 * The constructor initializes the random number generator with the current time
 */
Random::Random() {
    srand(time(nullptr));
}

Random* Random::getInstance() {
    if(!instance)
        instance = new Random;
    return instance;
}

unsigned Random::getRandom(unsigned n) {
    return (unsigned) (1 + rand() / (RAND_MAX + 1.0) * n);
}

```

```cpp
#ifndef POA_L1_RANDOM_H
#define POA_L1_RANDOM_H

#include <cstdlib>

/**
 * Singleton Random unsigned int generator.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Random {

    /**
     * Single instance of the random class
     */
    static Random* instance;
    /**
     * Default constructor
     */
    Random();

public:

    /**
     * If there is no instance of Random, create one
     *
     * @return The Random class is a singleton.  The getInstance() method returns the single instance of
     * the class.
     */
    static Random* getInstance();

    /**
      * Generate a random number between 1 and n
      *
      * @param n The upper bound of the number generated.
      * @return A random number between 1 and n.
      */
    unsigned int getRandom(unsigned n);

    /**
     * Singleton class can't be copied
     */
    Random(Random &other) = delete;
    /**
     * Singleton class can't be assigned
     */
    void operator=(const Random &) = delete;
};


```

```
#endif //POA_L1_RANDOM_H

```

```cpp
#include "Sub.h"

int Sub::calculate(unsigned a, unsigned b) const {
    return a - b;
}

```

```cpp
//
// Created by André on 10.03.2022.
//

#ifndef POA_L1_SUB_H
#define POA_L1_SUB_H


#include "Operation.h"

/**
 * Substraction operation.
 *
 * @author Nelson Jeanrenaud
 * @author André Marques Nora
 * @version 1.0
 */
class Sub : public Operation{
public:

    /**
     * "Subtract b from a and return the result."
     *
     * @param a The first number to be subtracted.
     * @param b The number of times to run the test.
     * @return The result of the substraction
     */
    int calculate(unsigned a, unsigned b) const override;
};


#endif //POA_L1_SUB_H

```