**Lab Description:**

\*\*In README file

\*\*Not on lab write up

**Sample Output iomodel:**

Leontif Economic I/O Model Calculations

===========================

Amount of each product needed:

Coal Mining: 133.724 units

Steelworks: 236.45 units

Gem Hunting: 287.607 units

===========================

===========================

Amount of each product needed:

Leatherworks: -18.9297 units

Alchemy: -85.3043 units

Shell Sorting: -11.0886 units

SteelHouse: -96.5608 units

Isotope Farming House: -38.4621 units

===========================

===========================

Amount of each product needed:

Aircraft: -76.3621 units

Pharmaceutical: -62.0765 units

Automobile: -93.3138 units

Food Service: -96.0215 units

===========================

**Sample Output MatrixDriver:**

Testing printMatrix Function

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

0.6 0.16 0.16

1 0.6 0.16

1 1 0.6

Testing Overloaded Constructor

Creating 5x4 matrix

3 2 3 2

3 2 2 3

2 1 1 1

1 3 3 1

2 1 1 3

Testing Scalar multiplication overloaded multiplication (BONUS)

Test3

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Multiplying test3 by 4

1.2 0.32 0.32

2 1.2 0.32

2 2 1.2

Copy constructor

Test3

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Test5 copied over from test3

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Overloaded assignment

test6 assigned as test5

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Preventing self assignment

test6

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

test2

3 2 3 2

3 2 2 3

2 1 1 1

1 3 3 1

2 1 1 3

Copy test2 over to test6

test6

3 2 3 2

3 2 2 3

2 1 1 1

1 3 3 1

2 1 1 3

Testing overloaded addition

test7

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

test8

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Result

0.6 0.16 0.16

1 0.6 0.16

1 1 0.6

Testing overloaded subtraction

test7

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

test8

0.3 0.08 0.08

0.5 0.3 0.08

0.5 0.5 0.3

Result

0 0 0

0 0 0

0 0 0

Testing overloaded multiplication

test10

2 1

2 3

test11

3

3

Result

9

15

Testing Diagonal Matrix creation

0 5 5

2 0 5

4 4 0

Testing Triangle Matrix creation

Upper

5 3 3

0 4 1

0 0 2

Lower

3 0 0

4 5 0

1 2 3

Testing Identity Matrix creation

1 0 0

0 1 0

0 0 1

Testing the transpose function

Current working matrix:

2 1 1 3

3 3 1 2

3 3 3 3

1 1 2 2

Transpose is...

2 3 3 1

1 3 3 1

1 1 3 2

3 2 3 2

Testing inverse function

Current working matrix...

3 1

2 1

Inverse of matrix...

1 -1

-2 3

Multiplied should be identity matrix...

1 -8.88178e-16

-1.77636e-15 1

**Data.txt:**

Coal Mining

Steelworks

Gem Hunting

---

0.6 0.02 0.1

0.3 0.2 0.4

0.2 0.4 0.3

---

20

34

80

---

Leatherworks

Alchemy

Shell Sorting

SteelHouse

Isotope Farming House

---

0.2 0.01 0.5 0.7 0.03

0.4 0.8 0.06 0.7 0.5

0.2 0.6 0.01 0.4 0.4

0.7 0.9 0.8 0.5 0.4

0.7 0.7 0.1 0.2 0.6

---

60

78

98

66

78

---

Aircraft

Pharmaceutical

Automobile

Food Service

---

0.2 0.4 0.5 0.1

0.4 0.3 0.6 0.02

0.2 0.6 0.78 0.5

0.6 0.7 0.1 0.9

---

20

45

80

89

---

**Iomodel.cpp:**

#include <iostream> // cout, endl

#include <stdio.h> // printf

#include "matrix.h" // Matrix class

#include <fstream> // ifstream

Matrix calculateData(Matrix&, Matrix&); // Does necessary calculations for problem

int main(int argc, char\*\* argv) {

srand(time(NULL));

int argCount = argc;

if (argCount != 2) { // Error Code for incorrect arguments

std::perror("Cannot Execute Program: Error Code\n\t--Amount of arguments incorrect");

exit(1);

}

std::ifstream inFile;

try {

inFile.open(argv[1]);

} catch (...) {

std::perror("Error: File input not found"); // Error Code for incorrect file

exit(1);

}

int companyCount = 0;

std::string\* compNames = new std::string[50]; // Company limit of 50

std::cout << "Leontif Economic I/O Model Calculations" << std::endl;

while (inFile.peek() != EOF) {

for (int i = 0; inFile.peek() != '-'; i++) {

std::getline(inFile, compNames[i]);

companyCount++;

}

inFile.ignore(3,'\n');

Matrix sample(companyCount, companyCount);

double num = 0;

for (int i = 0; i < companyCount; i++) {

for (int j = 0; j < companyCount; j++) {

inFile >> num;

sample.setElement(i, j, num);

}

}

inFile.ignore(5, '\n');

inFile.ignore(5, '\n');

Matrix demand(companyCount, 1);

for (int i = 0; i < companyCount; i++) {

inFile >> num;

demand.setElement(i, 0, num);

}

Matrix result = calculateData(sample, demand);

std::cout << "===========================" << std::endl;

std::cout << "Amount of each product needed: " << std::endl;

for (int i = 0; i < companyCount; i++) {

std::cout << compNames[i] << ": " << result.getElement(i, 0) << " units" <<

std::endl;

}

std::cout << "===========================" << std::endl;

inFile.ignore(5, '\n');

inFile.ignore(5, '\n');

num = 0;

companyCount = 0;

}

delete [] compNames;

inFile.close();

return 0;

}

/\*

\* calculateData Function:

\* Does the necessary calculations to get the output for the given problem, returns

\* a matrix

\*/

Matrix calculateData(Matrix& input, Matrix& demand) {

Matrix identity = input;

identity.fillMatrixIdentity();

Matrix rtnMe = (identity - input).inverse() \* demand;

return rtnMe;

}

**MatrixDriver.cpp:**

#include <iostream> // cout, endl

#include <stdio.h> // printf

#include "matrix.h" // Matrix class

int main() {

srand(time(NULL));

std::cout << "Testing printMatrix Function" << std::endl;

Matrix test;

test.printMatrix();

(test \* 2).printMatrix();

std::cout << "Testing Overloaded Constructor" << std::endl;

std::cout << "Creating 5x4 matrix" << std::endl;

Matrix test2(5,4);

test2.printMatrix();

std::cout << "Testing Scalar multiplication overloaded multiplication (BONUS)" << std::endl;

Matrix test3;

Matrix test4;

std::cout << "Test3\n";

test3.printMatrix();

std::cout << "Multiplying test3 by 4" << std::endl;

(test3 \* 4).printMatrix();

std::cout << "Copy constructor" << std::endl;

std::cout << "Test3" << std::endl;

test3.printMatrix();

std::cout << "Test5 copied over from test3" << std::endl;

Matrix test5(test3);

test5.printMatrix();

std::cout << "Overloaded assignment" << std::endl;

Matrix test6 = test5;

std::cout << "test6 assigned as test5" << std::endl;

test6.printMatrix();

std::cout << "Preventing self assignment" << std::endl;

test6 = test6;

std::cout << "test6" << std::endl;

test6.printMatrix();

std::cout << "test2" << std::endl;

test2.printMatrix();

std::cout << "Copy test2 over to test6" << std::endl;

test6 = test2;

std::cout << "test6" << std::endl;

test6.printMatrix();

std::cout << "Testing overloaded addition" << std::endl;

Matrix test7;

std::cout << "test7" << std::endl;

test7.printMatrix();

Matrix test8;

std::cout << "test8" << std::endl;

test8.printMatrix();

Matrix test9 = test7 + test8;

std::cout << "Result" << std::endl;

test9.printMatrix();

std::cout << "Testing overloaded subtraction" << std::endl;

std::cout << "test7" << std::endl;

test7.printMatrix();

std::cout << "test8" << std::endl;

test8.printMatrix();

test9 = test7 - test8;

std::cout << "Result" << std::endl;

test9.printMatrix();

std::cout << "Testing overloaded multiplication" << std::endl;

Matrix test10(2,2);

Matrix test11(2,1);

std::cout << "test10" << std::endl;

test10.printMatrix();

std::cout << "test11" << std::endl;

test11.printMatrix();

Matrix test12 = test10 \* test11;

std::cout << "Result" << std::endl;

test12.printMatrix();

std::cout << "Testing Diagonal Matrix creation" << std::endl;

Matrix test13;

test13.fillMatrixDiagonal();

test13.printMatrix();

std::cout << "Testing Triangle Matrix creation" << std::endl;

std::cout << "Upper" << std::endl;

test13.fillMatrixTriangle(true);

test13.printMatrix();

std::cout << "Lower" << std::endl;

test13.fillMatrixTriangle(false);

test13.printMatrix();

std::cout << "Testing Identity Matrix creation" << std::endl;

test13.fillMatrixIdentity();

test13.printMatrix();

std::cout << "Testing the transpose function" << std::endl;

std::cout << "Current working matrix: " << std::endl;

Matrix norm(4,4);

norm.printMatrix();

std::cout << "Transpose is..." << std::endl;

Matrix trans = norm.transpose();

trans.printMatrix();

std::cout << "Testing inverse function" << std::endl;

std::cout << "Current working matrix..." << std::endl;

Matrix m(2,2);

m.printMatrix();

std::cout << "Inverse of matrix..." << std::endl;

Matrix inv = m.inverse();

inv.printMatrix();

std::cout << "Multiplied should be identity matrix..." << std::endl;

(m \* inv).printMatrix();

return 0;

}

**Matrix.h:**

#ifndef \_MATRIX\_H\_

#define \_MATRIX\_H\_

#include <iostream> // cout, endl

#include <stdio.h> // printf

#include <math.h> // log

class Matrix {

private:

double\*\* arr;

int rowLength;

int columnLength;

void fillMatrix(); // Fills the matrix with random values

public:

Matrix(); // Default constructor

Matrix(int, int); // Overloaded constructor that takes a length and width

Matrix(const Matrix&); // Copy constructor

~Matrix(); // Destructor

Matrix& operator=(const Matrix&); // Overloaded assignment

void setElement(int, int, double); // Sets a cell to a specified element

double getElement(int, int); // Gets an element at a specifed cell

bool isSymmetric(); // Checks for symmetry

Matrix pad(int); // Pads the matrix to be a power of 2

Matrix inverse(); // Finds the inverse of the function and returns it

Matrix transpose(); // Returns a matrix that is the transpose of it

void fillMatrixDiagonal(); // Creates a diagonal matrix

void fillMatrixTriangle(bool); // Creates a upper/lower triangular matrix

void fillMatrixIdentity(); // Creates an identity matrix

Matrix addMatrices(Matrix&); // Adds two matrices together

Matrix subtractMatrices(Matrix&); // Subtracts two matrices from each other

Matrix multMatrices(Matrix&); // Multiplies two matrices together

Matrix operator\*(int); // Used for multiplying a matrix by a scalar

Matrix operator\*(Matrix&); // Used for multiplying matrices together

Matrix operator+(Matrix&); // Used for more simple adding of two matrices

Matrix operator-(Matrix&); // Used for more simple subtracting of two matrices

void printMatrix(); // Prints out the corresponding matrix

};

#endif

**Matrix.cpp:**

#include "matrix.h"

/\*

\* Default Constructor:

\* Initializes the matrix and uses a fill function to fill the matrix completely

\*/

Matrix::Matrix() {

arr = new double\*[3];

for (int i = 0; i < 3; i++) {

arr[i] = new double[3];

}

rowLength = 3;

columnLength = 3;

fillMatrix();

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (i == j) {

arr[i][j] = 0.3;

} else if (i > j){

arr[i][j] = 0.5;

} else if (j > i) {

arr[i][j] = 0.08;

} else if (i > j && j >=2) {

arr[i][j] = 0.46;

}

}

}

}

/\*

\* setElement Function:

\* Takes in indeces and data and then assigns it to the correct cell

\*/

void Matrix::setElement(int i, int j, double num) {

if (i < 0 || i > rowLength || j < 0 || j > columnLength) {

throw "Incorrect indeces!";

}

arr[i][j] = num;

}

/\*

\* getElement Function:

\* Takes indeces and returns the element at the specified cell

\*/

double Matrix::getElement(int i, int j) {

if (i < 0 || i > rowLength || j < 0 || j > columnLength) {

throw "Incorrect indeces!";

}

return arr[i][j];

}

/\*

\* Overloaded Constructor:

\* Initializes the matrix to specified row and column values and fills the matrix

\*/

Matrix::Matrix(int r, int c) {

if (r < 1 || c < 1) {

throw "Dimensions are not positive real numbers!";

}

arr = new double\*[r];

for (int i = 0; i < r; i++) {

arr[i] = new double[c];

}

rowLength = r;

columnLength = c;

fillMatrix();

}

/\*

\* fillMatrix Function:

\* Fills the matrix with values from 1 to 5 for testing purposes

\*/

void Matrix::fillMatrix() {

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

arr[i][j] = 1 + rand() % 3;

}

}

}

/\*

\* fillMatrixDiagonal Function:

\* Creates a diagonal matrix

\*/

void Matrix::fillMatrixDiagonal() {

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (i == j) {

arr[i][j] = 0;

} else {

arr[i][j] = 1 + rand() % 5;

}

}

}

}

/\*

\* fillMatrixTriangle Function:

\* Takes a bool to determine if the matrix will be an upper or lower triangle matrix

\* and then creates that type of matrix accordingly

\*/

void Matrix::fillMatrixTriangle(bool isUpper) {

if (isUpper) {

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (j >= i) {

arr[i][j] = 1 + rand() % 5;

} else {

arr[i][j] = 0;

}

}

}

} else {

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (j <= i) {

arr[i][j] = 1 + rand() % 5;

} else {

arr[i][j] = 0;

}

}

}

}

}

/\*

\* fillMatrixIdentity Function:

\* Creates an identity matrix

\*/

void Matrix::fillMatrixIdentity() {

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (i == j) {

arr[i][j] = 1;

} else {

arr[i][j] = 0;

}

}

}

}

/\*

\* Copy Constructor:

\* Performs deep copy on matrix

\*/

Matrix::Matrix(const Matrix& rhs) {

arr = new double\*[rhs.rowLength];

for (int i = 0; i < rhs.rowLength; i++) {

arr[i] = new double[rhs.columnLength];

}

rowLength = rhs.rowLength;

columnLength = rhs.columnLength;

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

arr[i][j] = rhs.arr[i][j];

}

}

}

/\*

\* Destructor:

\* Performs memory deallocation

\*/

Matrix::~Matrix() {

for (int i = 0; i < rowLength; i++) {

delete [] arr[i];

}

delete [] arr;

}

/\*

\* Overloaded assignment operator:

\* Performs a deep copy of a matrix to another matrix

\*/

Matrix& Matrix::operator=(const Matrix& rhs) {

if (this == &rhs) {

return \*this;

}

for (int i = 0; i < rowLength; i++) {

delete [] arr[i];

}

delete [] arr;

arr = new double\*[rhs.rowLength];

for (int i = 0; i < rhs.rowLength; i++) {

arr[i] = new double[rhs.columnLength];

}

rowLength = rhs.rowLength;

columnLength = rhs.columnLength;

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

arr[i][j] = rhs.arr[i][j];

}

}

return \*this;

}

/\*

\* transpose Function:

\* Takes the matrix of the specified class and finds the tranpose and then returns

\* a new matrix that is the transpose of the passed matrix

\*/

Matrix Matrix::transpose() {

Matrix trans(columnLength, rowLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

trans.arr[j][i] = arr[i][j];

}

}

return trans;

}

/\*

\* isSymmetric Function:

\* Checks to see if the matrix is symmetric or not

\*/

bool Matrix::isSymmetric() {

Matrix tmp1 = \*this;

Matrix tmp2 = tmp1.transpose();

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (tmp1.arr[i][j] != tmp2.arr[i][j]) {

return false;

}

}

}

return true;

}

/\*

\* pad Function:

\* Pads the matrix to become a power of 2

\*/

Matrix Matrix::pad(int dimension) {

int count = 0;

while (log2(dimension + count) - (int)log2(dimension + count) != 0) {

count++;

}

Matrix rtnMe(dimension + count, dimension + count);

Matrix identity(count, count);

identity.fillMatrixIdentity();

for (int i = 0; i < rtnMe.rowLength; i++) {

for (int j = 0; j < rtnMe.columnLength; j++) {

if (i < dimension && j < dimension) {

rtnMe.arr[i][j] = arr[i][j];

}

if (i >= dimension && j >= dimension) {

rtnMe.arr[i][j] = identity.arr[i - dimension][j - dimension];

}

if ((i >= dimension && j < dimension) || (i < dimension && j >= dimension)) {

rtnMe.arr[i][j] = 0;

}

}

}

return rtnMe;

}

/\*

\* inverse Function:

\* Finds the inverse of the function and then returns a matrix with that is the inverse

\*/

Matrix Matrix::inverse() {

if (rowLength == 1 || columnLength == 1) {

Matrix rtnMe(1,1);

if (arr[0][0] != 0) {

rtnMe.arr[0][0] = 1.0 / arr[0][0];

} else {

rtnMe.arr[0][0] = 0;

}

return rtnMe;

}

if (rowLength != columnLength) {

throw "Matrix not square!";

}

int originalRow = rowLength;

if (log2(rowLength) - (int)log2(rowLength) != 0) { // Checking for power of 2

Matrix fixCurr = pad(rowLength);

Matrix rtnMe = fixCurr.inverse();

Matrix newRtnMe(originalRow, originalRow);

for (int i = 0; i < originalRow; i++) {

for (int j = 0; j < originalRow; j++) {

newRtnMe.arr[i][j] = rtnMe.arr[i][j];

}

}

return newRtnMe;

}

if (!isSymmetric()) { // Checking for symmetry

Matrix tmp = \*this;

Matrix tmp2 = tmp.transpose();

Matrix tmp3 = tmp2 \* tmp;

Matrix tmp4 = tmp3.inverse() \* tmp2;

return tmp4;

}

Matrix B(rowLength / 2, columnLength / 2);

Matrix C(rowLength / 2, columnLength / 2);

Matrix CT(rowLength / 2, columnLength / 2);

Matrix D(rowLength / 2, columnLength / 2);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (i < rowLength / 2 && j < columnLength / 2) {

B.arr[i][j] = arr[i][j];

}

if (i < rowLength / 2 && j >= columnLength / 2) {

CT.arr[i][j - columnLength / 2] = arr[i][j];

}

if (i >= rowLength / 2 && j < columnLength / 2) {

C.arr[i - rowLength / 2][j] = arr[i][j];

}

if (i >= rowLength / 2 && j >= columnLength / 2) {

D.arr[i - rowLength / 2][j - columnLength / 2] = arr[i][j];

}

}

}

Matrix newB = B.inverse();

Matrix W = C \* newB;

Matrix Wtrans = newB \* CT;

Matrix X = W \* CT;

Matrix S = D - X;

Matrix V = S.inverse();

Matrix Y = V \* W;

Matrix Ytrans = Y.transpose();

Matrix T = Ytrans \* -1;

Matrix U = Y \* -1;

Matrix Z = Wtrans \* Y;

Matrix R = newB + Z;

Matrix rtnMe(rowLength, columnLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

if (i < rowLength / 2 && j < columnLength / 2) {

rtnMe.arr[i][j] = R.arr[i][j];

}

if (i < rowLength / 2 && j >= columnLength / 2) {

rtnMe.arr[i][j] = T.arr[i][j - columnLength / 2];

}

if (i >= rowLength / 2 && j < columnLength / 2) {

rtnMe.arr[i][j] = U.arr[i - rowLength / 2][j];

}

if (i >= rowLength / 2 && j >= columnLength / 2) {

rtnMe.arr[i][j] = V.arr[i - rowLength / 2][j - columnLength / 2];

}

}

}

return rtnMe;

}

/\*

\* addMatrices Function:

\* Takes a matrix as an input and adds the two matrices together, returning a third matrix

\*/

Matrix Matrix::addMatrices(Matrix& rhs) {

if (rowLength != rhs.rowLength || columnLength != rhs.columnLength) {

throw "Matrix dimensions aren't equal!";

}

Matrix rtnMe(rowLength, columnLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

rtnMe.arr[i][j] = arr[i][j] + rhs.arr[i][j];

}

}

return rtnMe;

}

/\*

\* Overloaded addition operator:

\* Allows for easy addition instead of using matrix functions

\*/

Matrix Matrix::operator+(Matrix& rhs) {

return addMatrices(rhs);

}

/\*

\* subtractMatrices Function:

\* Takes a matrix as an input and then subtracts it from the current class matrix

\*/

Matrix Matrix::subtractMatrices(Matrix& rhs) {

if (rowLength != rhs.rowLength || columnLength != rhs.columnLength) {

throw "Matrix dimensions aren't equal!";

}

Matrix rtnMe(rowLength, columnLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

rtnMe.arr[i][j] = arr[i][j] - rhs.arr[i][j];

}

}

return rtnMe;

}

/\*

\* Overloaded subtraction operator:

\* Allows for easy subtraction instead of using matrix functions

\*/

Matrix Matrix::operator-(Matrix& rhs) {

return subtractMatrices(rhs);

}

/\*

\* multMatrices Function:

\* Takes a matrix as an input and then multipliese it with the current class matrix

\*/

Matrix Matrix::multMatrices(Matrix& rhs) {

if (columnLength != rhs.rowLength) {

throw "Matrix dimensions incorrect for algorithm! NxM & MxK not satisfies";

}

Matrix rtnMe(rowLength, rhs.columnLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < rhs.columnLength; j++) {

double sum = 0;

for (int l = 0; l < columnLength; l++) {

sum += arr[i][l] \* rhs.arr[l][j];

}

rtnMe.arr[i][j] = sum;

}

}

return rtnMe;

}

/\*

\* Overloaded multiplication operator:

\* Takes a scalar (single int) and a matrix and multiplies them together

\*/

Matrix Matrix::operator\*(int scalar) {

Matrix rtnMe(rowLength, columnLength);

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

rtnMe.arr[i][j] = scalar \* arr[i][j];

}

}

return rtnMe;

}

/\*

\* Overloaded multiplication operator:

\* Takes a matrix and then multiplies the matrices together

\*/

Matrix Matrix::operator\*(Matrix& rhs) {

return multMatrices(rhs);

}

/\*

\* printMatrix Function:

\* Prints the current matrix of the class out to the screen

\*/

void Matrix::printMatrix() {

std::cout << std::endl;

for (int i = 0; i < rowLength; i++) {

for (int j = 0; j < columnLength; j++) {

std::cout << arr[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}