**SOURCE CODE**

**MATRIX OPERATIONS.h**

#pragma once

#ifndef MATRIX\_OPERATIONS\_H

#define MATRIX\_OPERATIONS\_H

#include <iostream>

static const int SIZE = 100;

class Matrix

{

friend std::ostream &operator<<(std::ostream&, Matrix &);

private:

double doubleMatrix[SIZE][SIZE]; //Matrix will be the square and the size defined above

public:

Matrix operator-(Matrix&); //Subtraction

Matrix operator+(Matrix&); //Addition

Matrix operator\*(Matrix&); //Multiplication

void zero(); //Zeroes out a matrix

void initialize(); //Sets matrix to default values

Matrix(); //Only Need Default Constructor

};

#endif //MATRIX\_MULTIPLICATION\_MATRIX\_H

**MatrixOperations.cpp**

//This is responsible for matrix operations

#include "stdafx.h"

#include "MatrixOperations.h"

/\*-------------------------

\* Constructors

\*----------------------\*/

Matrix::Matrix()

{ }

/\*-----------------

\* Operations

\*----------------\*/

void Matrix::zero() //zeroes out a matrix

{

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

this->doubleMatrix[i][j] = 0;

}

}

}

void Matrix::initialize() //initializes matrix with diagonal values as 2.00001

//and everything else as 1.00001

{

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

(i == j) ? this->doubleMatrix[i][j] = 2.00001 : this->doubleMatrix[i][j] = 1.00001;

}

}

}

Matrix Matrix::operator+(Matrix& otherMatrix) //addition operation

{

Matrix resultingMatrix = Matrix();

resultingMatrix.zero();

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

resultingMatrix.doubleMatrix[i][j]=this->doubleMatrix[i][j] +otherMatrix.doubleMatrix[i][j];

}

}

return resultingMatrix;

}

Matrix Matrix::operator-(Matrix& otherMatrix) //subtraction operation

{

Matrix resultingMatrix = Matrix();

resultingMatrix.zero();

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

resultingMatrix.doubleMatrix[i][j] = this->doubleMatrix[i][j] - otherMatrix.doubleMatrix[i][j];

}

}

return resultingMatrix;

}

Matrix Matrix::operator\*(Matrix& otherMatrix) //multiplication operation

{

Matrix resultingMatrix = Matrix();

for (int i = 0; i < SIZE; ++i)

{

for (int j = 0; j < SIZE; ++j)

{

resultingMatrix.doubleMatrix[i][j] = 0;

for (int k = 0; k < SIZE; ++k)

{

resultingMatrix.doubleMatrix[i][j] += this->doubleMatrix[i][k] \* otherMatrix.doubleMatrix[k][j];

}

}

}

return resultingMatrix;

}

std::ostream &operator<<(std::ostream &output, Matrix &matrix) //output

{

for (int rowIndex = 0; rowIndex < SIZE; rowIndex++)

{

output << "\n";

for (int colIndex = 0; colIndex < SIZE; colIndex++)

{

output << matrix.doubleMatrix[rowIndex][colIndex] << "\t";

}

}

output << "\n";

return output;

}

**SortingOperation.h**

#pragma once

#ifndef SORTING\_OPERATION\_H

#define SORTING\_OPERATION\_H

#include <iostream>

#pragma once

class SortingOperation

{

friend std::ostream &operator<<(std::ostream&, SortingOperation &);

private:

static const int SIZE = 6000; //Set size to do around same ops/sec as matrix multiplication

int sortingArray[SIZE]; //Allocating memory for array that will be sorted

public:

void initializeArray(); //Initializes array with values

void commenceSort(); //Begins the sort operation with the initialized array

bool checkArray(); //Verifies array was sorted correctly

static void sort( int[], int lowerBound, int upperBound ); //Main sorting method

static void merge(int arrayToSort[], int lowerBound, int midpoint, int upperBound); //Merging . //method within sort method

};

#endif

**SortingOperation.cpp**

//Merge Sort for Integer Array

#include "stdafx.h"

#include "SortingOperation.h"

/\*-------------------------

\* Array Maintenance

\*------------------------\*/

void SortingOperation::initializeArray() //Seeds values into array; repeats 0, 1, 2, 3

{

int counter = 0;

for (int i = 0; i < SIZE; i++)

{

sortingArray[i] = counter;

counter = (counter + 1) % 4;

}

}

bool SortingOperation::checkArray() //Verifies that array is sorted

{

int temp = sortingArray[0];

for (int i = 0; i < SIZE; i++)

{

if (sortingArray[i] < temp) return false;

temp = sortingArray[i];

}

return true;

}

void SortingOperation::commenceSort() //Calls the sorting operations on the generated array

{

SortingOperation::sort(sortingArray, 0, SIZE - 1);

}

/\*-------------------------

\* Output of Array

\*------------------------\*/

std::ostream &operator<<(std::ostream &output, SortingOperation &SortingOperation) //output

{

output << "Array: \n";

for (int i = 0; i < SortingOperation::SIZE; i++)

{

output << SortingOperation.sortingArray[i] << " ";

}

output << "\n";

return output;

}

/\*-------------------------

\* The Merge Sort

\*----------------------\*/

void SortingOperation::sort(int arrayToSort[], int lowerBound, int upperBound ) //Primary Sort …………. //Operation

{

if (lowerBound < upperBound)

{

int midPoint = (upperBound + lowerBound) / 2; //Declare Midpoint

sort(arrayToSort, lowerBound, midPoint); //Recursive Call for Left Side

sort(arrayToSort, midPoint + 1, upperBound); //Recursive Call for Right Side

merge(arrayToSort, lowerBound, midPoint,upperBound ); //Merging function

}

}

void SortingOperation::merge(int arrayToSort[], int lowerBound, int midpoint, int upperBound) //Merging Operation

{

int i, j, k;

const int leftSize = midpoint - lowerBound + 1; //Size of Left Subarray

const int rightSize = upperBound - midpoint; //Size of Right Subarray

int \*leftArray = new int[leftSize]; //Left Side Subarray

int \*rightArray = new int[rightSize]; //Right Side Subarray

for (i = 0; i < leftSize; i++) leftArray[i] = arrayToSort[lowerBound + i]; //Transfer Values

for (j = 0; j < rightSize; j++) rightArray[j] = arrayToSort[midpoint + 1 + j]; //Transfer Values

i = 0;

j = 0;

k = lowerBound;

while (i < leftSize && j < rightSize) //This section compares and places values from the //two subarrays

{

if (leftArray[i] <= rightArray[j])

{

arrayToSort[k] = leftArray[i]; //If leftArray component smaller add it to //the original array

i++;

}

else

{

arrayToSort[k] = rightArray[j]; //If rightArray component smaller add it to //original array

j++;

}

k++;

}

while (i < leftSize) //Adds any remnant values from leftArray

{

arrayToSort[k] = leftArray[i];

i++;

k++;

}

while (j < rightSize) //Adds any remnant values from rightArray

{

arrayToSort[k] = rightArray[j];

j++;

k++;

}

delete leftArray;

delete rightArray;

}

**CPUben.cpp**

#include "stdafx.h"

#include <ctime>

#include <iostream>

#include "MatrixOperations.h"

#include "SortingOperation.h"

using namespace std;

int main()

{

double clockStart, clockEnd, runTime, harmonicMean, doubleScore, integerScore;

const int TEST\_TIME = 10, SCORE\_MODIFIER = 4;

int integerCounter = 0, doubleCounter = 0;

int endTime;

Matrix matrixOne, matrixTwo;

matrixOne.initialize();

matrixTwo.initialize();

SortingOperation integerSortingOperation;

cout << "========================================================================================\n"

<< "This is a simple benchmark program utilizing double and integer operations.\n"

<< "========================================================================================\n"

<< "Integer operations are checked using merge sort.\n"

<< "Double operations are checked using matrix multiplication.\n"

<< TEST\_TIME << " seconds will be allocated to each type of operation.\n\n\n";

/\*=======================

\*Begin Double Test

=========================\*/

clockStart = clock();

endTime = clockStart + TEST\_TIME \* double(CLOCKS\_PER\_SEC);

while ( clock() < endTime )

{

matrixOne\*matrixTwo;

doubleCounter++;

}

clockEnd = clock();

runTime = (clockEnd - clockStart) / double(CLOCKS\_PER\_SEC);

doubleScore = double(doubleCounter) / runTime;

printf("Double Values:");

printf("A total of %d operations were performed in %.0lf seconds. \n", doubleCounter, runTime);

printf("Operations per minute: %.0lf \n", doubleScore\*60);

printf("Operations per second: %.0lf \n\n\n", doubleScore);

/\*=======================

\*Begin Integer Test

=========================\*/

clockStart = clock();

endTime = clockStart + TEST\_TIME \* double(CLOCKS\_PER\_SEC);

while (clock() < endTime)

{

integerSortingOperation.initializeArray();

integerSortingOperation.commenceSort();

integerCounter++;

}

clockEnd = clock();

runTime = (clockEnd - clockStart) / double(CLOCKS\_PER\_SEC);

integerScore = double(integerCounter) / runTime;

printf("Integer Values:");

printf("A total of %d operations were performed in %.0lf seconds. \n", integerCounter, runTime);

printf("Operations per minute: %.0lf \n", 60 \* integerCounter / runTime);

printf("Operations per second: %.0lf \n\n", integerCounter / runTime);

/\*=======================

\*Display Results

=========================\*/

harmonicMean = 2.0 / (1.0/integerScore + 1.0/doubleScore);

printf("Harmonic mean: %.0lf \n",harmonicMean );

printf("===================================================\n");

printf("Benchmark Score: %.0lf \n", harmonicMean / SCORE\_MODIFIER);

printf("===================================================\n");

return 0;

}