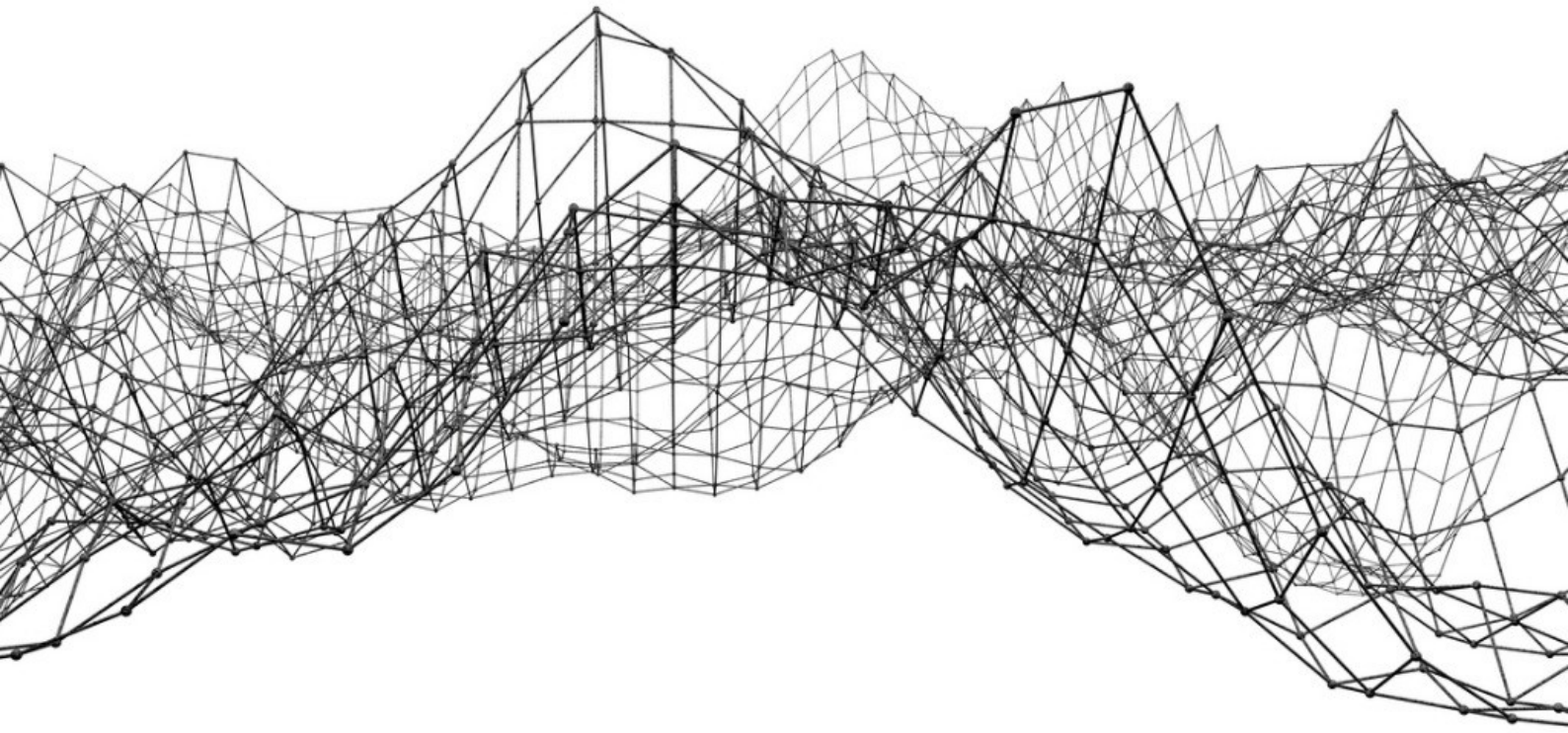
Linear Algebra

Assignment # 01

Saad Bin Khalid

20k-0161



Task 2

This task has used C++ programming language to find the echelon and reduced echelon form of a matrix of any order. The program contains general purpose custom defined classes which are used to implement the working of matrix. Note that each block of code has been implemented by the author himself and no third party resource has been used in it.

Organization:

The complete project consists of following folders:

1. bin - to store the driving or main files.
2. include - to store the custom header files.
3. Lib - to store the definitions of custom header files.

The program consists of following classes:

4. Fraction - to implement mathematical fractions.
5. Row - to implement matrix rows.
6. Matrix - to implement a matrix.

The program consists of following files. Note that each class is separated into .h (headers) and .cpp (definitions) files.

1. main.cpp - the program where main() resides and all classes are used.
2. Fraction.h & Fraction.cpp.
3. Row.h & Row.cpp.
4. Matrix.h & Matrix.cpp.
5. input.txt - to fetch user input.

Usage

1. If not already compiled, compile the program using g++ compiler.
    1. Install the g++ compiler and create its global path variable.
    2. Open ‘bin’ folder in ‘Command Prompt’ (cmd).
    3. Enter : “g++ main.cpp - o main”.
2. You will find a “input.txt” file along attached with the code (if the file is not provided, you can go ahead and create one).
3. Specify the order of the matrix in the first line.
4. Specify each row in a separate new line.
5. Now type ‘main’ in cmd.
6. If you want to see each step of transformations, do the specified changes mentioned in the comments in main.cpp

input.txt

```
4 3
1   1   -2
0   5   -9
2   8   72
1   2   56
```

```
saad@Saad-desktop:~/Fall 2021/Linear Algebra/Assignments/Matrix/bin$ g++ main.cpp -o main
saad@Saad-desktop:~/Fall 2021/Linear Algebra/Assignments/Matrix/bin$ ./main
* Original Matrix:
[         1         1        -2   ]
[         0         5        -9   ]
[         2         8        72   ]
[         1         2        56   ]

* Echelon Form:
[         1         1        -2   ]
[         0         1      38/3   ]
[         0         0         1   ]
[         0         0         0   ]

* Reduced Echelon Form:
[         1         0         0   ]
[         0         1         0   ]
[         0         0         1   ]
[         0         0         0   ]

saad@Saad-desktop:~/Fall 2021/Linear Algebra/Assignments/Matrix/bin$
```

Fraction.h

```h
#pragma once
#include <iostream>
#include <iomanip>
#include <string>
#include <math.h>
#include <exception>
using namespace std;

/**
 * A class to implement the working of fractions in Math.
 * Supports all arithmetic functions applicable to traditional fractions.
 * Any numeric value is convertible to Fraction object
 */
class Fraction
{
    //? friend functions:
    /**
     * To display the fraction in a formatted fashion.
     * @param ostream The output stream to write to.
     * @param Fraction The fraction to display.
     * @return The same output stream for cascading the function call.
     */
    friend ostream &operator<<(ostream &os, const Fraction &other);
    /**
     * To input the fraction from an input stream.
     * @param istream The input stream to read from.
     * @param Fraction The fraction to initialize.
     * @return The same input stream for cascading the function call.
     */
    friend istream &operator>>(istream &is, Fraction &other);

private:
    // The numerator of the fraction.
    int numerator;
    // The denominator of the fraction.
    int denominator;

    //? private functions:
    /**
     * To find the greatest common divisor (GCD) of two numbers.
     * @param int number 1.
     * @param int number 2.
     * @return the GCD of the numbers.
     */
    int gcd(int a, int b) const;

public:
    //? constructor and destructor:
    // To initialize the default value of fraction as 0.
    Fraction();
    // To initialize the value of fraction from a number.
    Fraction(const int &value);
    // To initialize the value of fraction.
    Fraction(const int &numerator, const int &denominator);
    // To initialize the value of fraction from another fraction.
    Fraction(const Fraction &other);

    //? access modifiers:
    /**
     * To return the numerator of object.
     * @param void
     * @return The copy of this object's numerator.
     */
    int getNumerator() const;
    /**
     * To return the denominator of fraction.
     * @param void
     * @return The copy of this fraction's denominator.
     */
    int getDenominator() const;
    /**
     * To set the value of fraction.
     * @param int numerator of the fraction.
     * @param int denominator of the fraction.
     * @return void
     */
    void setValue(int numerator, int denominator);

    //? helper functions:
    /**
     * To reduce the fraction to a decimal value.
     * @param void.
     * @return The decimal equivalent of fraction.
     */
    double toDouble() const;
    /**
     * To transform the fraction into a formatted string.
     * @param void.
     * @return The string representation of fraction.
     */
    string toString() const;

    //? operator overloading :
    /**
     * To assign a new fraction to the current fraction.
     * @param Fraction the source fraction.
     * @return A reference to the current fraction for cascading function calls .
     */
    Fraction &operator=(const Fraction &other);
    /**
     * To sum two fractions.
     * @param Fraction the second fraction.
     * @return The sum of fractions.
     */
    Fraction operator+(const Fraction &other) const;
    /**
     * To add a fraction in the current fraction.
     * @param Fraction the second fraction.
     * @return A reference to the current fraction for cascading function calls.
     */
    Fraction &operator+=(const Fraction &other);
    /**
     * To subtract two fractions.
     * @param Fraction the second fraction.
     * @return The difference between the fractions.
     */
    Fraction operator-(const Fraction &other) const;
    /**
     * To subtract a fraction from the current fraction.
     * @param Fraction the second fraction.
     * @return A reference to the current fraction for cascading function calls.
     */
    Fraction &operator-=(const Fraction &other);
    /**
     * To multiply two fractions.
     * @param Fraction the second fraction.
     * @return The product of fractions.
     */
    Fraction operator*(const Fraction &other) const;
    /**
     * To multiply a fraction with the current fraction.
     * @param Fraction the second fraction.
     * @return A reference to the current fraction for cascading function calls.
     */
    Fraction &operator*=(const Fraction &other);
    /**
     * To divide two fractions.
     * @param Fraction the second fraction.
     * @return The result of division of fractions.
     */
    Fraction operator/(const Fraction &other) const;
    /**
     * To divide a fraction with the current fraction.
     * @param Fraction the second fraction.
     * @return A reference to the current fraction for cascading function calls.
     */
    Fraction &operator/=(const Fraction &other);
    /**
     * To compare two fractions.
     * @param Fraction the second fraction.
     * @return True if both fractions are equal; otherwise, false.
     */
    bool operator==(const Fraction &other) const;
    /**
     * To compare two fractions.
     * @param Fraction the second fraction.
     * @return True if both fractions are not equal; otherwise, false.
     */
    bool operator!=(const Fraction &other) const;
};
```

Row.h

```h
#pragma once
#include "Fraction.h"
using namespace std;

/**
 * A class to implement the working of a row of a matrix.
 * It can also be used as an alternative to dynamic safe array.
 * It uses 'Fractions' as its elements .
*/
class Row
{
    //? friend functions:
    /**
     * To display the row in a formatted fashion.
     * @param ostream The output stream to write to.
     * @param Row The row to display.
     * @return The same output stream for cascading the function call.
     */
    friend ostream &operator<<(ostream &os, const Row &row);
    /**
     * To input the row from an input stream  .
     * @param istream The input stream to read from.
     * @param Fraction The row to initialize.
     * @return The same input stream for cascading the function call.
     */
    friend istream &operator>>(istream &is, Row &row);

private:
    // The maximum capacity of the row.
    // Fractions cannot exceed this length.
    unsigned capacity = 0;
    // The current length occupied by the elements of row.
    // It never exceeds the 'capacity' of the row.
    unsigned length = 0;
    // The pointer to elements of row.
    // The elements are implements using 'Fraction.h'.
    Fraction *elements = nullptr;

    //? private functions:
    /**
     * To allocate memory for the elements of row.
     * @param int the maximum capacity of the row.
     * @return void
     */
    void allocate(const int &capacity);
    /**
     * To deallocate memory of row.
     * @param void
     * @return void
     */
    void deallocateRow();

public:
    //? constructors and destructor:
    // To initialize the default row as an empty row.
    Row();
    // To initialize the row by memory only.
    // No elements are inserted.
    Row(const int &capacity);
    // To initialize the row from a list of fractions.
    Row(const int &capacity, Fraction fractions[]);
    // To initialize the row from another row.
    Row(const Row &other);
    // To destroy the row when exiting.
    ~Row();

    //? access modifiers:
    /**
     * To insert an element at the end of row.
     * @param Fraction the fraction to be inserted (also supports numeric values).
     * @return The index of inserted element.
     */
    int push(const Fraction &fraction);
    /**
     * To remove an element from the end of row.
     * @param void
     * @return The removed Fraction.
     */
    Fraction pop();
    /**
     * To return the maximum available capacity of row.
     * @param void
     * @return The capacity.
     */
    unsigned getCapacity() const;
    /**
     * To return the length of row which is currently used by elements.
     * @param void
     * @return The length.
     */
    unsigned getLength() const;
    /**
     * To populate the row from given Fractions.
     * The previous elements of the row are overriden.
     * @param int the number of fractions to be inserted.
     * @param values[] the list of fractions to insert from.
     * @return void
     */
    void setFractions(const int &length, Fraction fractions[]);
    /**
     * To fill the specified elements of row with the given Fraction.
     * The previous elements of the row are overriden.
     * @param int the number of elements to override.
     * @param Fraction the fraction to fill.
     * @return void
     */
    void fillRow(const int &length, const Fraction &fraction);
    /**
     * To remove all elements of row.
     * The resulting row is now empty, but it still retains its capacity.
     * @param void
     * @return void
     */
    void clear();

    //? helpers
    /**
     * To return the number of starting 0s.
     * @param void
     * @return The number of elements after starting zeros.
     */
    unsigned startingZeroCount() const;

    //? operators:
    /**
     * To fill all elements with a new fraction.
     * @param Fraction the source fraction.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator=(const Fraction &frac);
    /**
     * To assign a new row to the current row.
     * @param Row the source row.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator=(const Row &other);
    /**
     * To find the sum of two rows.
     * @param Row the second row.
     * @return The sum of two rows.
     */
    Row operator+(const Row &other) const;
    /**
     * To add a row into the current row (element-wise).
     * @param Row the second row.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator+=(const Row &other);
    /**
     * To find the difference of two rows.
     * @param Row the second row.
     * @return The difference of two rows.
     */
    Row operator-(const Row &other) const;
    /**
     * To subtract a row from the current row (element-wise).
     * @param Row the second row.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator-=(const Row &other);
    /**
     * To find the product of a row and a fraction.
     * @param Fraction the source fraction.
     * @return The product row whose each element is a [element*fraction].
     */
    Row operator*(const Fraction &frac) const;
    /**
     * To multiply every element of row by a Fraction.
     * @param Fraction the source fraction.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator*=(const Fraction &frac);
    /**
     * To find the division of a row and a fraction.
     * @param Fraction the source fraction.
     * @return The resulting row whose each element is a [element/fraction].
     */
    Row operator/(const Fraction &frac) const;
    /**
     * To divide every element of row by a Fraction.
     * @param Fraction the source fraction.
     * @return A reference to the current row for cascading function calls .
     */
    Row &operator/=(const Fraction &frac);
    /**
     * To compare two rows.
     * @param Row the second row.
     * @return True if both rows are equal; otherwise, false.
     */
    bool operator==(const Row &other) const;
    /**
     * To compare two rows.
     * @param Row the second row.
     * @return false if both rows are equal; otherwise, true.
     */
    bool operator!=(const Row &other) const;
    /**
     * To acces an element of row.
     * @param int the index of element.
     * @return the Fraction element at the index, if any.
     */
    Fraction operator[](const int &index) const;
};
```

Matrix.h

```h
#pragma once
#include "Row.h"
using namespace std;

// A structure to hold attributes of an element of a matrix
struct Element
{
    // the fraction value of element
    Fraction frac = 0;
    // the row number of element
    unsigned row = 0;
    // the column number of element
    unsigned col = 0;
};

/**
 * A class to implement the working of a general matrix.
 * It supports all workings of a matrix.
*/
class Matrix
{
    //? frind functions:
    /**
     * To display the Matrix in a formatted fashion.
     * @param ostream The output stream to write to.
     * @param Row The Matrix to display.
     * @return The same output stream for cascading the function call.
     */
    friend ostream &operator<<(ostream &os, const Matrix &matrix);

public:
    // the body to hold rows of matrix
    Row *body = nullptr;
    // the number of of matrix
    unsigned rows = 0;
    // the number of columns of matrix
    unsigned cols = 0;
    // a flag to control the ouput of each step
    bool steps = false;

    //? private functions:
    /**
     * To allocate memory for rows of the matrix
     * @param int the number of rows.
     * @return void
     */
    void allocateRows(const int &rows);
    /**
     * To allocate memory for columns of the matrix
     * @param int the number of columns.
     * @param Fraction[][] the elements of matrix.
     * @return void
     */
    void allocateColumns(const int &cols, Fraction *fractions[]);
    /**
     * To deallocate memory for matrix.
     * @param void.
     * @return void.
     */
    void deallocateMatrix();

public:
    //? constructors and destructor:
    // to initialize the matrix of given dimensions with the given Fractions
    Matrix(const int &rows, const int &cols, Fraction **fractions);
    // to destroy the matrix when exiting
    ~Matrix();

    //? access modifiers:
    /**
     * To fill the specified number of rows with given fraction.
     * @param int number of rows.
     * @param Fraction the source fraction.
     * @return void
     */
    void fillRow(const int &row, const Fraction &frac);
    /**
     * To fill all rows with given fraction.
     * @param Fraction the source fraction.
     * @return void
     */
    void fill(const Fraction &frac);

    //? EROs (Elementary row operations):
    /**
     * To add the source row into destination row after multiplying it a by a factor, if any.
     * @param int destination row.
     * @param int source row.
     * @param Fraction the factor (by default, it is 1).
     * @return void
     */
    void add(const int &rowDest, const int &rowSrc, const Fraction &multiplier = 1);
    /**
     * To multiply the specified row by given factor.
     * @param int row number.
     * @param Fraction the factor.
     * @return void
     */
    void multiply(const int &row, const Fraction &factor);
    /**
     * To divide the specified row by given fraction.
     * @param int row number.
     * @param Fraction the fraction.
     * @return void
     */
    void divide(const int &row, const Fraction &factor);
    /**
     * To interchange two rows.
     * @param int row1 number.
     * @param int row2 number.
     * @return void
     */
    void change(const int &row1, const int &row2);

    //? helper functions:
    /**
     * To sort matrix rows according to their number of starting 0s.
     * @param void
     * @return void
     */
    void sortRows();
    /**
     * To transform current matrix into echelon form.
     * @param bool an indicator to control transformation to reduced echelon form
     * @return void
     */
    void toEchelon(const bool &reduced = false);
    /**
     * To show steps whenever the matrix is transformed to a new state
     * @param bool a flad to control ouptut
     * @return void
     */
    void showSteps(const bool &show)
    {
        this->steps = show;
    }
};

```

Fraction.cpp

```cpp
#include "../includes/Fraction.h"

//? constructor and destructor:

Fraction::Fraction()
{
    setValue(0, 1);
}

Fraction::Fraction(const int &value)
{
    setValue(value, 1);
}

Fraction::Fraction(const int &numerator, const int &denominator)
{
    setValue(numerator, denominator);
}

Fraction::Fraction(const Fraction &other)
{
    setValue(other.numerator, other.denominator);
}

//? private functions:

int Fraction::gcd(int a, int b) const
{
    int r;
    while (b > 0)
    {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

//? access modifiers:

int Fraction::getNumerator() const
{
    return numerator;
}

int Fraction::getDenominator() const
{
    return denominator;
}

void Fraction::setValue(int numerator, int denominator)
{
    // handling an undefined state
    if (denominator == 0)
    {
        throw logic_error("Denominator cannot be zero.");
    }
    else if (denominator < 0)
    {
        // transfer - sign to numerator
        numerator = -numerator;
        denominator = -denominator;
    }
    // simplifying the fraction
    int GCD = gcd(abs(numerator), abs(denominator));
    this->numerator = numerator / GCD;
    this->denominator = denominator / GCD;
}

//? helper functions:

double Fraction::toDouble() const
{
    return static_cast<double>(this->numerator) / this->denominator;
}

string Fraction::toString() const
{
    string s;
    s += to_string(this->numerator);
    if (this->denominator != 1)
    {
        s += "/";
        s += to_string(this->denominator);
    }
    return s;
}

//? operator overloading :

Fraction &Fraction::operator=(const Fraction &other)
{
    setValue(other.numerator, other.denominator);
    return *this;
}

Fraction Fraction::operator+(const Fraction &other) const
{
    const int num = this->numerator * other.denominator + this->denominator * other.numerator;
    const int denom = this->denominator * other.denominator;
    return Fraction(num, denom);
}

Fraction &Fraction::operator+=(const Fraction &other)
{
    *this = *this + other;
    return *this;
}

Fraction Fraction::operator-(const Fraction &other) const
{
    return *this + Fraction(-other.numerator, other.denominator);
}

Fraction &Fraction::operator-=(const Fraction &other)
{
    *this = *this - other;
    return *this;
}

Fraction Fraction::operator*(const Fraction &other) const
{
    return Fraction(numerator * other.numerator, denominator * other.denominator);
}

Fraction &Fraction::operator*=(const Fraction &other)
{
    // avoiding " *this = *this * other " to save some function calls
    setValue(numerator * other.numerator, denominator * other.denominator);
    return *this;
}

Fraction Fraction::operator/(const Fraction &other) const
{
    return Fraction(numerator * other.denominator, denominator * other.numerator);
}

Fraction &Fraction::operator/=(const Fraction &other)
{
    // avoiding " *this = *this / other " to save some functions' calls
    setValue(numerator * other.denominator, denominator * other.numerator);
    return *this;
}

bool Fraction::operator==(const Fraction &other) const
{
    return this->toDouble() == other.toDouble();
}

bool Fraction::operator!=(const Fraction &other) const
{
    return this->toDouble() != other.toDouble();
}

//? friend functions:

ostream &operator<<(ostream &os, const Fraction &other)
{
    os << other.toString();
    return os;
}
istream &operator>>(istream &is, Fraction &other)
{
    is >> other.numerator;
    is >> other.denominator;
    return is;
}
```

Row.cpp

```cpp
#include "../includes/Row.h"

//? constructors and destrutor:

Row::Row() = default;

Row::Row(const int &capacity)
{
    allocate(capacity);
}

Row::Row(const int &capacity, Fraction fractions[])
{
    allocate(capacity);
    setFractions(capacity, fractions);
}

Row::Row(const Row &other)
{
    allocate(other.capacity);
    setFractions(other.capacity, other.elements);
}

Row::~Row()
{
    deallocateRow();
}

//? private functions:

void Row::allocate(const int &capacity)
{
    if (capacity <= 0)
        throw out_of_range("The capacity must be a greater than 0");

    this->length = 0;
    this->capacity = capacity;
    this->elements = new Fraction[capacity];
}

void Row::deallocateRow()
{
    delete[] this->elements;
    this->elements = nullptr;
    this->length = 0;
    this->capacity = 0;
}

//? access modifiers:

int Row::push(const Fraction &fraction)
{
    // if row is already full
    if (this->length == this->capacity)
        throw overflow_error("The row is already full");

    const unsigned newIndex = this->length++;
    elements[newIndex] = fraction;
    return newIndex;
}

Fraction Row::pop()
{
    // if row is already empty
    if (this->length == 0)
        throw underflow_error("The row is already empty");

    return elements[--length];
}

unsigned Row::getCapacity() const
{
    return capacity;
}

unsigned Row::getLength() const
{
    return length;
}

void Row::setFractions(const int &length, Fraction fractions[])
{
    if (length <= 0)
        throw underflow_error("The specified length should be greater than 0");
    if (length > this->capacity)
        throw overflow_error("The specified length is more than the available capacity");

    this->length = 0;
    for (unsigned i = 0; i < length; i++)
        push(fractions[i]);
}

void Row::fillRow(const int &length, const Fraction &fraction)
{
    if (length <= 0)
        throw underflow_error("The specified length should be greater than 0");
    if (length > this->capacity)
        throw overflow_error("The specified length is more than the available capacity");

    this->length = 0;
    for (unsigned i = 0; i < length; i++)
        push(fraction);
}

void Row::clear()
{
    this->length = 0;
}

//? helper functions:

unsigned Row::startingZeroCount() const
{
    unsigned count;
    for (count = 0; count < this->length; count++)
        if (elements[count] != 0)
            break;
    return count;
}

//? operations:

Row &Row::operator=(const Fraction &frac)
{
    fillRow(this->length, Fraction(frac));
    return *this;
}

Row &Row::operator=(const Row &other)
{
    if (other.capacity != this->capacity)
    {
        deallocateRow();
        allocate(other.capacity);
    }
    setFractions(other.capacity, other.elements);
    return *this;
}

Row Row::operator+(const Row &other) const
{
    if (this->capacity != other.capacity)
        throw runtime_error("In order to add two rows, there capacity must be same");

    Row temp(this->capacity);
    for (unsigned col = 0; col < this->capacity; col++)
        temp.push(elements[col] + other.elements[col]);
    return temp;
}

Row &Row::operator+=(const Row &other)
{
    if (this->capacity != other.capacity)
        throw runtime_error("In order to add two rows, there capacity must be same");

    for (unsigned i = 0; i < this->length; i++)
        elements[i] += other.elements[i];
    return *this;
}

Row Row::operator-(const Row &other) const
{
    if (this->capacity != other.capacity)
        throw runtime_error("In order to subtract two rows, there capacity must be same");

    Row temp(this->capacity);
    for (unsigned col = 0; col < this->capacity; col++)
        temp.push(elements[col] - other.elements[col]);
    return temp;
}

Row &Row::operator-=(const Row &other)
{
    if (this->capacity != other.capacity)
        throw runtime_error("In order to subtract two rows, there capacity must be same");

    for (unsigned i = 0; i < this->length; i++)
        elements[i] -= other.elements[i];
    return *this;
}

Row Row::operator*(const Fraction &frac) const
{
    Row temp(this->capacity);
    for (unsigned col = 0; col < this->capacity; col++)
        temp.push(elements[col] * frac);
    return temp;
}

Row &Row::operator*=(const Fraction &frac)
{
    for (unsigned i = 0; i < this->length; i++)
        elements[i] *= frac;
    return *this;
}

Row Row::operator/(const Fraction &frac) const
{
    Row temp(this->capacity);
    for (unsigned col = 0; col < this->capacity; col++)
        temp.push(elements[col] / frac);
    return temp;
}

Row &Row::operator/=(const Fraction &frac)
{
    for (unsigned i = 0; i < this->length; i++)
        elements[i] /= frac;
    return *this;
}

bool Row::operator==(const Row &other) const
{
    if (this->capacity != other.capacity)
        throw runtime_error("In order to compare two rows, there capacity must be same");

    bool areEqual = true;
    for (unsigned col = 0; col < this->capacity; col++)
    {
        if (elements[col] != other.elements[col])
        {
            areEqual = false;
            break;
        }
    }
    return areEqual;
}

bool Row::operator!=(const Row &other) const
{
    return !(*this == other);
}

Fraction Row::operator[](const int &index) const
{
    if (index < 0 || index >= this->length)
        throw out_of_range("The index is out of range of the available length");
    return elements[index];
}

//? friend functions:

ostream &operator<<(ostream &os, const Row &row)
{
    os << "[";
    for (unsigned col = 0; col < row.length; col++)
    {
        os << setw(10);
        os << row.elements[col];
        os << setw(4);
    }
    os << "]";
    return os;
}
istream &operator>>(istream &is, Row &row)
{
    for (unsigned col = 0; col < row.getCapacity(); col++)
    {
        is >> row.elements[col];
    }
    return is;
}
```

Matrix.cpp

```cpp
#include "../includes/Matrix.h"

//? constructors and destructor:

Matrix::Matrix(const int &rows, const int &cols, Fraction **fractions)
{
    allocateRows(rows);
    allocateColumns(cols, fractions);
}

Matrix::~Matrix()
{
    deallocateMatrix();
}

//? private functions:

void Matrix::allocateRows(const int &rows)
{
    if (rows <= 0)
        throw range_error("The number of rows must be greater than 0");

    this->rows = rows;
    this->body = new Row[rows];
}

void Matrix::allocateColumns(const int &cols, Fraction *fractions[])
{
    if (cols <= 0)
        throw range_error("The number of cols must be greater than 0");

    this->cols = cols;
    for (unsigned i = 0; i < this->rows; i++)
        body[i] = Row(cols, fractions[i]);
}

void Matrix::deallocateMatrix()
{
    delete[] body;
    this->rows = 0;
    this->cols = 0;
}

//? access modifiers:

void Matrix::fillRow(const int &row, const Fraction &frac)
{
    if (row <= 0 || row > this->rows)
        throw out_of_range("The row to be filled is out of range ");

    body[row - 1].clear();
    body[row - 1].fillRow(this->cols, frac);
}
void Matrix::fill(const Fraction &frac)
{
    for (unsigned i = 0; i < this->rows; i++)
        fillRow(i, frac);
}

//? EROs (Elementary row operations):

void Matrix::add(const int &rowDest, const int &rowSrc, const Fraction &multiplier)
{
    if (rowDest <= 0 || rowDest > this->rows)
        throw out_of_range("The destination row is out of range");
    if (rowSrc <= 0 || rowSrc > this->rows)
        throw out_of_range("The source row is out of range");

    body[rowDest - 1] += body[rowSrc - 1] * multiplier;
}

void Matrix::multiply(const int &row, const Fraction &factor)
{
    if (row <= 0 || row > this->rows)
        throw out_of_range("The row is out of range");

    body[row - 1] *= factor;
}

void Matrix::divide(const int &row, const Fraction &factor)
{
    if (row <= 0 || row > this->rows)
        throw out_of_range("The row is out of range");

    body[row - 1] /= factor;
}

void Matrix::change(const int &row1, const int &row2)
{
    if (row1 <= 0 || row1 > this->rows)
        throw out_of_range("The row is out of range");
    if (row2 <= 0 || row2 > this->rows)
        throw out_of_range("The row is out of range");

    Row row1Copy(body[row1 - 1]);
    body[row1 - 1] = body[row2 - 1];
    body[row2 - 1] = row1Copy;
}

//? helper functions:

void Matrix::sortRows()
{
    for (unsigned pass = 1; pass < rows; pass++)
    {
        for (unsigned i = 0; i < rows - 1; i++)
        {
            unsigned currRow = body[i].startingZeroCount();
            unsigned nextRow = body[i + 1].startingZeroCount();

            if (currRow > nextRow)
            {
                change(i + 1, i + 2);
                if (this->steps)
                {
                    cout << "Change R" << i + 1 << " with R" << i + 2 << "\n";
                    cout << *this << "\n";
                }
            }
            // if equal num of levels, change to put 1 on the top
            else if (currRow == nextRow)
            {
                // skip if there is no nonzero values in the row
                if (currRow >= this->cols || nextRow >= this->cols)
                    continue;

                // checking the first non zero values
                if (body[i][currRow] != 1 && body[i][nextRow] == 1)
                {
                    change(i + 1, i + 2);
                    if (this->steps)
                    {
                        cout << "Change R" << i + 1 << " with R" << i + 2 << "\n";
                        cout << *this << "\n";
                    }
                }
            }
        }
    }
}

void Matrix::toEchelon(const bool &reduced)
{
    // array to hold target elements of each column
    // target element is the element after which all elements are 0
    Element targetElements[cols];

    // maximum row of the target elements
    unsigned maxRow = 0;
    // maximum col of the target elements
    unsigned maxCol = 0;

    // sorting the rows
    sortRows();

    // iterating through the matrix column-wise
    for (unsigned col = 0; col < cols; col++)
    {
        Element &target = targetElements[col];
        for (unsigned row = 0; row < rows; row++)
        {
            const Fraction element = body[row][col];
            // skip if element is 0
            if (element == 0)
                continue;
            // if it is the first column
            if (col == 0)
            {
                target.row = row + 1;
                target.col = col + 1;
                target.frac = element;
                break;
            }

            // if the element is to the far right of the previous target elements
            const Element &prevTarget = targetElements[col - 1];
            if (col > (prevTarget.col - 1) && row > (prevTarget.row - 1))
            {
                target.row = row + 1;
                target.col = col + 1;
                target.frac = element;
                break;
            };
            // if there is no taget element in this column
            target.frac = 0;
            target.row = rows;
            target.col = cols;
        }

        // transforming the matrix
        if (target.frac != 0)
        {
            // make target value 1
            if (target.frac != 1)
            {
                divide(target.row, target.frac);
                if (this->steps)
                {
                    cout << "Dividing R" << target.row << " by " << target.frac << endl;
                    cout << *this << "\n";
                }
            }

            // determine whether the result should be a reduced echelon or not
            const unsigned startingRow = reduced ? 0 : target.row;
            // make all elements of the column 0
            for (unsigned row = startingRow; row < rows; row++)
            {
                const Fraction oldVal = body[row][col];
                const Fraction newVal = oldVal * -1;

                // skip the element if it is the target element
                if (row == target.row - 1)
                    continue;
                // skip the element if it is already 0
                if (oldVal == 0)
                    continue;

                add(row + 1, target.row, newVal);

                if (this->steps)
                {
                    cout << "Adding " << newVal << "*R" << target.row << " in R" << row + 1 << endl;
                    cout << *this << "\n";
                }
            }
        }
    }
}

//? friend functions:

ostream &operator<<(ostream &os, const Matrix &matrix)
{
    for (unsigned row = 0; row < matrix.rows; row++)
        os << matrix.body[row] << "\n";
    return os;
}
```

main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <exception>
#include "../includes/Matrix.h"
#include "./defs.cpp"
using namespace std;

int main()
{
    // opening the input file
    ifstream inputFile = ifstream("input.txt");
    if (!inputFile.is_open())
        throw runtime_error("Exception: Input file not found. Try again.");

    // reading the dimensions of matrix
    unsigned rows = 0;
    unsigned cols = 0;
    inputFile >> rows >> cols;
    // allocate memory for the matrix
    Fraction **matrix = new Fraction *[rows];
    for (unsigned i = 0; i < rows; i++)
        matrix[i] = new Fraction[cols];

    // reading the elements
    for (unsigned row = 0; row < rows; row++)
    {
        for (unsigned col = 0; col < cols; col++)
        {
            double x;
            inputFile >> x;
            matrix[row][col] = Fraction(x);

            if (inputFile.eof())
            {
                row = rows; // to break outer loop
                break;
            }
        }
    }

    // declaring the Matrix
    Matrix m(rows, cols, matrix);

    // add this line if you want to see each step
    // m.showSteps(true);

    cout << "* Original Matrix:\n"
         << m << "\n";

    m.toEchelon();
    cout << "* Echelon Form:\n"
         << m << "\n";

    // pass true to transform to reduce form
    m.toEchelon(true);
    cout << "* Reduced Echelon Form:\n"
         << m << "\n";

    // deallocating memory
    for (unsigned i = 0; i < rows; i++)
        delete[] matrix[i];
    delete[] matrix;
    return 0;
}
```