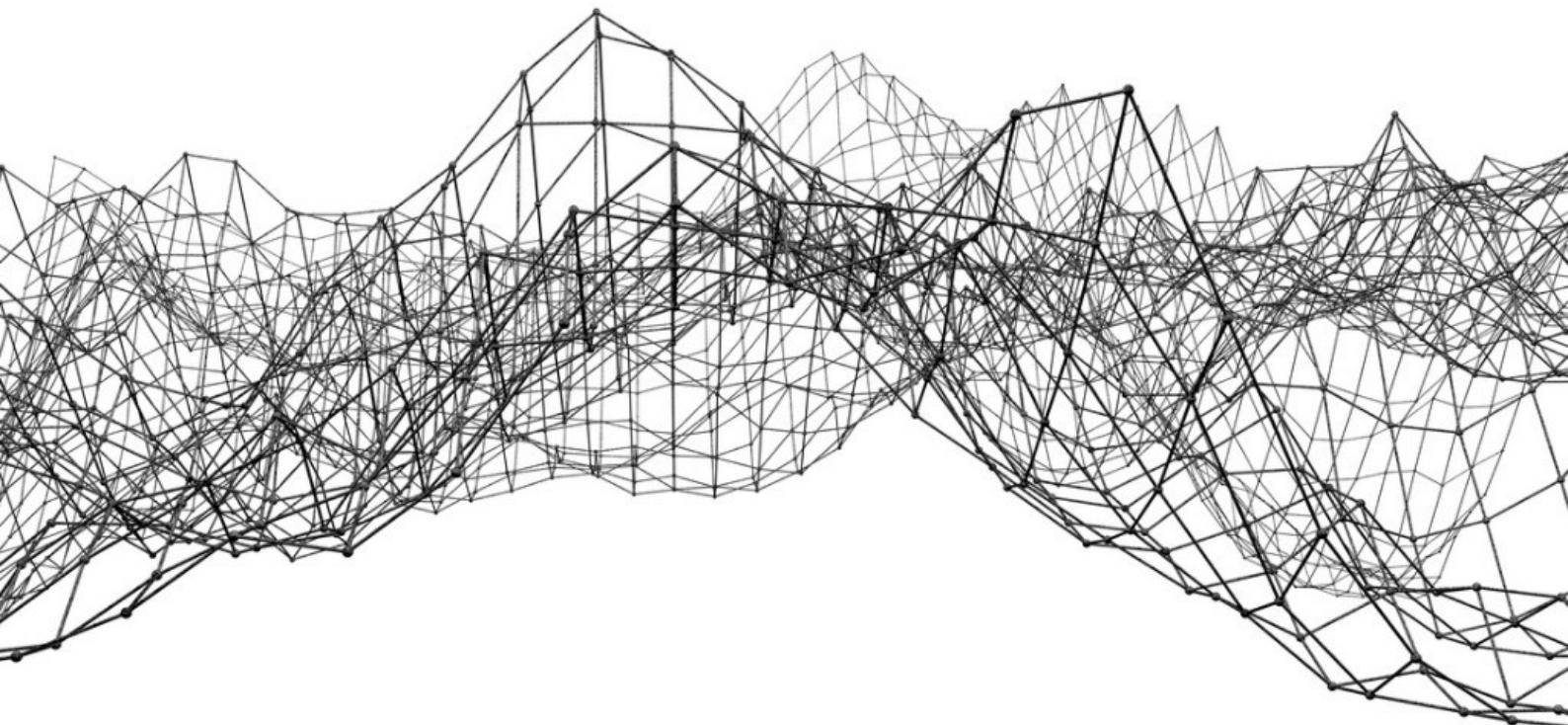


Linear Algebra

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

```

bin > input.txt
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$ 

```

Ln 2, Col 11 Spaces: 4 UTF-8 LF Plain Text

Fraction.h

```

1 #pragma once
2 #include <iostream>
3 #include <iomanip>
4 #include <string>
5 #include <math.h>
6 #include <exception>
7 using namespace std;
8
9 /**
10  * A class to implement the working of fractions in Math.
11  * Supports all arithmetic functions applicable to traditional fractions.
12  * Any numeric value is convertible to Fraction object
13  */
14 class Fraction
15 {
16     // friend functions:
17     /**
18      * To display the fraction in a formatted fashion.
19      * @param ostream The output stream to write to.
20      * @param Fraction The fraction to display.
21      * @return The same output stream for cascading the function call.
22      */
23     friend ostream &operator<<(ostream &os, const Fraction &other);
24     /**
25      * To input the fraction from an input stream.
26      * @param istream The input stream to read from.
27      * @param Fraction The fraction to initialize.
28      * @return The same input stream for cascading the function call.
29      */
30     friend istream &operator>>(istream &is, Fraction &other);
31
32 private:
33     // The numerator of the fraction.
34     int numerator;
35     // The denominator of the fraction.
36     int denominator;
37
38     //? private functions:
39     /**
40      * To find the greatest common divisor (GCD) of two numbers.
41      * @param int number 1.
42      * @param int number 2.
43      * @return the GCD of the numbers.
44      */
45     int gcd(int a, int b) const;
46
47 public:
48     //? constructor and destructor:
49     // To initialize the default value of fraction as 0.
50     Fraction();
51     // To initialize the value of fraction from a number.
52     Fraction(const int &value);
53     // To initialize the value of fraction.
54     Fraction(const int &numerator, const int &denominator);
55     // To initialize the value of fraction from another fraction.
56     Fraction(const Fraction &other);
57
58     //? access modifiers:
59     /**
60      * To return the numerator of object.
61      * @param void
62      * @return The copy of this object's numerator.
63      */
64     int getNumerator() const;
65     /**
66      * To return the denominator of fraction.
67      * @param void
68      * @return The copy of this fraction's denominator.
69      */
70     int getDenominator() const;
71     /**
72      * To set the value of fraction.
73      * @param int numerator of the fraction.
74      * @param int denominator of the fraction.
75      * @return void
76      */
77     void setValue(int numerator, int denominator);
78
79     //? helper functions:
80     /**
81      * To reduce the fraction to a decimal value.
82      * @param void
83      * @return The decimal equivalent of fraction.
84      */
85     double toDouble() const;
86     /**
87      * To transform the fraction into a formatted string.
88      * @param void
89      * @return The string representation of fraction.
90      */
91     string toString() const;
92
93     //? operator overloading :
94     /**
95      * To assign a new fraction to the current fraction.
96      * @param Fraction the source fraction.
97      * @return A reference to the current fraction for cascading function calls .
98      */
99     Fraction &operator=(const Fraction &other);
100    /**
101      * To sum two fractions.
102      * @param Fraction the second fraction.
103      * @return The sum of fractions.
104      */
105     Fraction operator+(const Fraction &other) const;
106    /**
107      * To add a fraction in the current fraction.
108      * @param Fraction the second fraction.
109      * @return A reference to the current fraction for cascading function calls.
110      */
111     Fraction &operator+=(const Fraction &other);
112    /**
113      * To subtract two fractions.
114      * @param Fraction the second fraction.
115      * @return The difference between the fractions.
116      */
117     Fraction operator-(const Fraction &other) const;
118    /**
119      * To subtract a fraction from the current fraction.
120      * @param Fraction the second fraction.
121      * @return A reference to the current fraction for cascading function calls.
122      */
123     Fraction &operator-=(const Fraction &other);
124    /**
125      * To multiply two fractions.
126      * @param Fraction the second fraction.
127      * @return The product of fractions.
128      */
129     Fraction operator*(const Fraction &other) const;
130    /**
131      * To multiply a fraction with the current fraction.
132      * @param Fraction the second fraction.
133      * @return A reference to the current fraction for cascading function calls.
134      */
135     Fraction &operator*=(const Fraction &other);
136    /**
137      * To divide two fractions.
138      * @param Fraction the second fraction.
139      * @return The result of division of fractions.
140      */
141     Fraction operator/(const Fraction &other) const;
142    /**
143      * To divide a fraction with the current fraction.
144      * @param Fraction the second fraction.
145      * @return A reference to the current fraction for cascading function calls.
146      */
147     Fraction &operator/=(const Fraction &other);
148    /**
149      * To compare two fractions.
150      * @param Fraction the second fraction.
151      * @return True if both fractions are equal; otherwise, false.
152      */
153     bool operator==(const Fraction &other) const;
154    /**
155      * To compare two fractions.
156      * @param Fraction the second fraction.
157      * @return True if both fractions are not equal; otherwise, false.
158      */
159     bool operator!=(const Fraction &other) const;
160 };

```

Row.h

```
1 #pragma once
2 #include "Fraction.h"
3 using namespace std;
4
5 /**
6 * A class to implement the working of a row of a matrix.
7 * It can also be used as an alternative to dynamic safe array.
8 * It uses 'Fractions' as its elements .
9 */
10 class Row
11 {
12     //? friend functions:
13     /**
14     * To display the row in a formatted fashion.
15     * @param ostream The output stream to write to.
16     * @param Row The row to display.
17     * @return The same output stream for cascading the function call.
18     */
19     friend ostream &operator<<(ostream &os, const Row &row);
20     /**
21     * To input the row from an input stream .
22     * @param istream The input stream to read from.
23     * @param Fraction The row to initialize.
24     * @return The same input stream for cascading the function call.
25     */
26     friend istream &operator>>(istream &is, Row &row);
27
28 private:
29     // The maximum capacity of the row.
30     // Fractions cannot exceed this length.
31     unsigned capacity = 0;
32     // The current length occupied by the elements of row.
33     // It never exceeds the 'capacity' of the row.
34     unsigned length = 0;
35     // The pointer to elements of row.
36     // The elements are implements using 'Fraction.h'.
37     Fraction *elements = nullptr;
38
39     //? private functions:
40     /**
41     * To allocate memory for the elements of row.
42     * @param int the maximum capacity of the row.
43     * @return void
44     */
45     void allocate(const int &capacity);
46     /**
47     * To deallocate memory of row.
48     * @param void
49     * @return void
50     */
51     void deallocateRow();
52
53 public:
54     //? constructors and destructor:
55     // To initialize the default row as an empty row.
56     Row();
57     // To initialize the row by memory only.
58     // No elements are inserted.
59     Row(const int &capacity);
60     // To initialize the row from a list of fractions.
61     Row(const int &capacity, Fraction fractions[]);
62     // To initialize the row from another row.
63     Row(const Row &other);
64     // To destroy the row when exiting.
65     ~Row();
66
67     //? access modifiers:
68     /**
69     * To insert an element at the end of row.
70     * @param Fraction the fraction to be inserted (also supports numeric values).
71     * @return The index of inserted element.
72     */
73     int push(const Fraction &fraction);
74     /**
75     * To remove an element from the end of row.
76     * @param void
77     * @return The removed Fraction.
78     */
79     Fraction pop();
80     /**
81     * To return the maximum available capacity of row.
82     * @param void
83     * @return The capacity.
84     */
85     unsigned getCapacity() const;
86     /**
87     * To return the length of row which is currently used by elements.
88     * @param void
89     * @return The length.
90     */
91     unsigned getLength() const;
92     /**
93     * To populate the row from given Fractions.
94     * The previous elements of the row are overriden.
95     * @param int the number of fractions to be inserted.
96     * @param values[] the list of fractions to insert from.
97     * @return void
98     */
99     void setFractions(const int &length, Fraction fractions[]);
100    /**
101     * To fill the specified elements of row with the given Fraction.
102     * The previous elements of the row are overriden.
103     * @param int the number of elements to override.
104     * @param Fraction the fraction to fill.
105     * @return void
106     */
107    void fillRow(const int &length, const Fraction &fraction);
108    /**
109     * To remove all elements of row.
110     * The resulting row is now empty, but it still retains its capacity.
111     * @param void
112     * @return void
113     */
114    void clear();
115
116    //? helpers
117    /**
118     * To return the number of starting 0s.
119     * @param void
120     * @return The number of elements after starting zeros.
121     */
122    unsigned startingZeroCount() const;
123
124    //? operators:
125    /**
126     * To fill all elements with a new fraction.
127     * @param Fraction the source fraction.
128     * @return A reference to the current row for cascading function calls .
129     */
130    Row &operator=(const Fraction &frac);
131    /**
132     * To assign a new row to the current row.
133     * @param Row the source row.
134     * @return A reference to the current row for cascading function calls .
135     */
136    Row &operator=(const Row &other);
137    /**
138     * To find the sum of two rows.
139     * @param Row the second row.
140     * @return The sum of two rows.
141     */
142    Row operator+(const Row &other) const;
143    /**
144     * To add a row into the current row (element-wise).
145     * @param Row the second row.
146     * @return A reference to the current row for cascading function calls .
147     */
148    Row &operator+=(const Row &other);
149    /**
150     * To find the difference of two rows.
151     * @param Row the second row.
152     * @return The difference of two rows.
153     */
154    Row operator-(const Row &other) const;
155    /**
156     * To subtract a row from the current row (element-wise).
157     * @param Row the second row.
158     * @return A reference to the current row for cascading function calls .
159     */
160    Row &operator-=(const Row &other);
161    /**
162     * To find the product of a row and a fraction.
163     * @param Fraction the source fraction.
164     * @return The product row whose each element is a [element*fraction].
165     */
166    Row operator*(const Fraction &frac) const;
167    /**
168     * To multiply every element of row by a Fraction.
169     * @param Fraction the source fraction.
170     * @return A reference to the current row for cascading function calls .
171     */
172    Row &operator*=(const Fraction &frac);
173    /**
174     * To find the division of a row and a fraction.
175     * @param Fraction the source fraction.
176     * @return The resulting row whose each element is a [element/fraction].
177     */
178    Row operator/(const Fraction &frac) const;
179    /**
180     * To divide every element of row by a Fraction.
181     * @param Fraction the source fraction.
182     * @return A reference to the current row for cascading function calls .
183     */
184    Row &operator/=(const Fraction &frac);
185    /**
186     * To compare two rows.
187     * @param Row the second row.
188     * @return True if both rows are equal; otherwise, false.
189     */
190    bool operator==(const Row &other) const;
191    /**
192     * To compare two rows.
193     * @param Row the second row.
194     * @return False if both rows are equal; otherwise, true.
195     */
196    bool operator!=(const Row &other) const;
197    /**
198     * To acces an element of row.
199     * @param int the index of element.
200     * @return the Fraction element at the index, if any.
201     */
202    Fraction operator[](const int &index) const;
203};
```

Matrix.h

```
1 #pragma once
2 #include "Row.h"
3 using namespace std;
4
5 // A structure to hold attributes of an element of a matrix
6 struct Element
7 {
8     // the fraction value of element
9     Fraction frac = 0;
10    // the row number of element
11    unsigned row = 0;
12    // the column number of element
13    unsigned col = 0;
14 };
15
16 /**
17 * A class to implement the working of a general matrix.
18 * It supports all workings of a matrix.
19 */
20 class Matrix
21 {
22     //? frind functions:
23     /**
24     * To display the Matrix in a formatted fashion.
25     * @param ostream The output stream to write to.
26     * @param Row The Matrix to display.
27     * @return The same output stream for cascading the function call.
28     */
29     friend ostream &operator<<(ostream &os, const Matrix &matrix);
30
31 public:
32     // the body to hold rows of matrix
33     Row *body = nullptr;
34     // the number of of matrix
35     unsigned rows = 0;
36     // the number of columns of matrix
37     unsigned cols = 0;
38     // a flag to control the ouput of each step
39     bool steps = false;
40
41     //? private functions:
42     /**
43     * To allocate memory for rows of the matrix
44     * @param int the number of rows.
45     * @return void
46     */
47     void allocateRows(const int &rows);
48     /**
49     * To allocate memory for columns of the matrix
50     * @param int the number of columns.
51     * @param Fraction[][][] the elements of matrix.
52     * @return void
53     */
54     void allocateColumns(const int &cols, Fraction *fractions[]);
55     /**
56     * To deallocate memory for matrix.
57     * @param void.
58     * @return void.
59     */
60     void deallocateMatrix();
61
62 public:
63     //? constructors and destructor:
64     // to initialize the matrix of given dimensions with the given Fractions
65     Matrix(const int &rows, const int &cols, Fraction **fractions);
66     // to destroy the matrix when exiting
67     ~Matrix();
68
69     //? access modifiers:
70     /**
71     * To fill the specified number of rows with given fraction.
72     * @param int number of rows.
73     * @param Fraction the source fraction.
74     * @return void
75     */
76     void fillRow(const int &row, const Fraction &frac);
77     /**
78     * To fill all rows with given fraction.
79     * @param Fraction the source fraction.
80     * @return void
81     */
82     void fill(const Fraction &frac);
83
84     //? EROs (Elementary row operations):
85     /**
86     * To add the source row into destination row after multiplying it a by a factor, if any.
87     * @param int destination row.
88     * @param int source row.
89     * @param Fraction the factor (by default, it is 1).
90     * @return void
91     */
92     void add(const int &rowDest, const int &rowSrc, const Fraction &multiplier = 1);
93     /**
94     * To multiply the specified row by given factor.
95     * @param int row number.
96     * @param Fraction the factor.
97     * @return void
98     */
99     void multiply(const int &row, const Fraction &factor);
100    /**
101     * To divide the specified row by given fraction.
102     * @param int row number.
103     * @param Fraction the fraction.
104     * @return void
105     */
106    void divide(const int &row, const Fraction &factor);
107    /**
108     * To interchange two rows.
109     * @param int row1 number.
110     * @param int row2 number.
111     * @return void
112     */
113    void change(const int &row1, const int &row2);
114
115    //? helper functions:
116    /**
117     * To sort matrix rows according to their number of starting 0s.
118     * @param void
119     * @return void
120     */
121    void sortRows();
122    /**
123     * To transform current matrix into echelon form.
124     * @param bool an indicator to control transformation to reduced echelon form
125     * @return void
126     */
127    void toEchelon(const bool &reduced = false);
128    /**
129     * To show steps whenever the matrix is transformed to a new state
130     * @param bool a flag to control ouput
131     * @return void
132     */
133    void showSteps(const bool &show)
134    {
135        this->steps = show;
136    }
137};
```

Fraction.cpp

```
1 #include "../includes/Fraction.h"
2 //? constructor and destructor:
3
4 Fraction::Fraction()
5 {
6     setValue(0, 1);
7 }
8
9
10 Fraction::Fraction(const int &value)
11 {
12     setValue(value, 1);
13 }
14
15 Fraction::Fraction(const int &numerator, const int &denominator)
16 {
17     setValue(numerator, denominator);
18 }
19
20 Fraction::Fraction(const Fraction &other)
21 {
22     setValue(other.numerator, other.denominator);
23 }
24
25 //? private functions:
26
27 int Fraction::gcd(int a, int b) const
28 {
29     int r;
30     while (b > 0)
31     {
32         r = a % b;
33         a = b;
34         b = r;
35     }
36     return a;
37 }
38
39 //? access modifiers:
40
41 int Fraction::getNumerator() const
42 {
43     return numerator;
44 }
45
46 int Fraction::getDenominator() const
47 {
48     return denominator;
49 }
50
51 void Fraction::setValue(int numerator, int denominator)
52 {
53     // handling an undefined state
54     if (denominator == 0)
55     {
56         throw logic_error("Denominator cannot be zero.");
57     }
58     else if (denominator < 0)
59     {
60         // transfer - sign to numerator
61         numerator = -numerator;
62         denominator = -denominator;
63     }
64     // simplifying the fraction
65     int GCD = gcd(abs(numerator), abs(denominator));
66     this->n numerator = numerator / GCD;
67     this->denominator = denominator / GCD;
68 }
69
70 //? helper functions:
71
72 double Fraction::toDouble() const
73 {
74     return static_cast<double>(this->n numerator) / this->denominator;
75 }
76
77 string Fraction::toString() const
78 {
79     string s;
80     s += to_string(this->n numerator);
81     if (this->denominator != 1)
82     {
83         s += "/";
84         s += to_string(this->denominator);
85     }
86     return s;
87 }
88
89 //? operator overloading :
90
91 Fraction &Fraction::operator=(const Fraction &other)
92 {
93     setValue(other.numerator, other.denominator);
94     return *this;
95 }
96
97 Fraction Fraction::operator+(const Fraction &other) const
98 {
99     const int num = this->n numerator * other.denominator + this->denominator * other.numerator;
100    const int denom = this->denominator * other.denominator;
101    return Fraction(num, denom);
102 }
103
104 Fraction &Fraction::operator+=(const Fraction &other)
105 {
106     *this = *this + other;
107     return *this;
108 }
109
110 Fraction Fraction::operator-(const Fraction &other) const
111 {
112     return *this + Fraction(-other.numerator, other.denominator);
113 }
114
115 Fraction &Fraction::operator-=(const Fraction &other)
116 {
117     *this = *this - other;
118     return *this;
119 }
120
121 Fraction Fraction::operator*(const Fraction &other) const
122 {
123     return Fraction(numerator * other.numerator, denominator * other.denominator);
124 }
125
126 Fraction &Fraction::operator*=(const Fraction &other)
127 {
128     // avoiding " *this = *this * other " to save some function calls
129     setValue(numerator * other.numerator, denominator * other.denominator);
130     return *this;
131 }
132
133 Fraction Fraction::operator/(const Fraction &other) const
134 {
135     return Fraction(numerator * other.denominator, denominator * other.numerator);
136 }
137
138 Fraction &Fraction::operator/=(const Fraction &other)
139 {
140     // avoiding " *this = *this / other " to save some functions' calls
141     setValue(numerator * other.denominator, denominator * other.numerator);
142     return *this;
143 }
144
145 bool Fraction::operator==(const Fraction &other) const
146 {
147     return this->toDouble() == other.toDouble();
148 }
149
150 bool Fraction::operator!=(const Fraction &other) const
151 {
152     return this->toDouble() != other.toDouble();
153 }
154
155 //? friend functions:
156
157 ostringstream &operator<<(ostringstream &os, const Fraction &other)
158 {
159     os << other.toString();
160     return os;
161 }
162 istream &operator>>(istream &is, Fraction &other)
163 {
164     is >> other.numerator;
165     is >> other.denominator;
166     return is;
167 }
```

Row.cpp

```
1 #include "../includes/Row.h"
2
3 //? constructors and destructor:
4
5 Row::Row() = default;
6
7 Row::Row(const int &capacity)
8 {
9     allocate(capacity);
10 }
11
12 Row::Row(const int &capacity, Fraction fractions[])
13 {
14     allocate(capacity);
15     setFractions(capacity, fractions);
16 }
17
18 Row::Row(const Row &other)
19 {
20     allocate(other.capacity);
21     setFractions(other.capacity, other.elements);
22 }
23
24 Row::~Row()
25 {
26     deallocateRow();
27 }
28
29 //? private functions:
30
31 void Row::allocate(const int &capacity)
32 {
33     if (capacity <= 0)
34         throw out_of_range("The capacity must be a greater than 0");
35
36     this->length = 0;
37     this->capacity = capacity;
38     this->elements = new Fraction[capacity];
39 }
40
41 void Row::deallocateRow()
42 {
43     delete[] this->elements;
44     this->elements = nullptr;
45     this->length = 0;
46     this->capacity = 0;
47 }
48
49 //? access modifiers:
50
51 int Row::push(const Fraction &fraction)
52 {
53     // if row is already full
54     if (this->length == this->capacity)
55         throw overflow_error("The row is already full");
56
57     const unsigned newIndex = this->length++;
58     elements[newIndex] = fraction;
59     return newIndex;
60 }
61
62 Fraction Row::pop()
63 {
64     // if row is already empty
65     if (this->length == 0)
66         throw underflow_error("The row is already empty");
67
68     return elements[--length];
69 }
70
71 unsigned Row::getCapacity() const
72 {
73     return capacity;
74 }
75
76 unsigned Row::getLength() const
77 {
78     return length;
79 }
80
81 void Row::setFractions(const int &length, Fraction fractions[])
82 {
83     if (length <= 0)
84         throw underflow_error("The specified length should be greater than 0");
85     if (length > this->capacity)
86         throw overflow_error("The specified length is more than the available capacity");
87
88     this->length = 0;
89     for (unsigned i = 0; i < length; i++)
90         push(fractions[i]);
91 }
92
93 void Row::fillRow(const int &length, const Fraction &fraction)
94 {
95     if (length <= 0)
96         throw underflow_error("The specified length should be greater than 0");
97     if (length > this->capacity)
98         throw overflow_error("The specified length is more than the available capacity");
99
100    this->length = 0;
101    for (unsigned i = 0; i < length; i++)
102        push(fraction);
103 }
104
105 void Row::clear()
106 {
107     this->length = 0;
108 }
109
110 //? helper functions:
111
112 unsigned Row::startingZeroCount() const
113 {
114     unsigned count;
115     for (count = 0; count < this->length; count++)
116         if (elements[count] != 0)
117             break;
118     return count;
119 }
120
121 //? operations:
122
123 Row &Row::operator=(const Fraction &frac)
124 {
125     fillRow(this->length, Fraction(frac));
126     return *this;
127 }
128
129 Row &Row::operator=(const Row &other)
130 {
131     if (other.capacity != this->capacity)
132     {
133         deallocateRow();
134         allocate(other.capacity);
135     }
136     setFractions(other.capacity, other.elements);
137     return *this;
138 }
139
140 Row Row::operator+(const Row &other) const
141 {
142     if (this->capacity != other.capacity)
143         throw runtime_error("In order to add two rows, there capacity must be same");
144
145     Row temp(this->capacity);
146     for (unsigned col = 0; col < this->capacity; col++)
147         temp.push(elements[col] + other.elements[col]);
148     return temp;
149 }
150
151 Row &Row::operator+=(const Row &other)
152 {
153     if (this->capacity != other.capacity)
154         throw runtime_error("In order to add two rows, there capacity must be same");
155
156     for (unsigned i = 0; i < this->length; i++)
157         elements[i] += other.elements[i];
158     return *this;
159 }
160
161 Row Row::operator-(const Row &other) const
162 {
163     if (this->capacity != other.capacity)
164         throw runtime_error("In order to subtract two rows, there capacity must be same");
165
166     Row temp(this->capacity);
167     for (unsigned col = 0; col < this->capacity; col++)
168         temp.push(elements[col] - other.elements[col]);
169     return temp;
170 }
171
172 Row &Row::operator-=(const Row &other)
173 {
174     if (this->capacity != other.capacity)
175         throw runtime_error("In order to subtract two rows, there capacity must be same");
176
177     for (unsigned i = 0; i < this->length; i++)
178         elements[i] -= other.elements[i];
179     return *this;
180 }
181
182 Row Row::operator*(const Fraction &frac) const
183 {
184     Row temp(this->capacity);
185     for (unsigned col = 0; col < this->capacity; col++)
186         temp.push(elements[col] * frac);
187     return temp;
188 }
189
190 Row &Row::operator*=(const Fraction &frac)
191 {
192     for (unsigned i = 0; i < this->length; i++)
193         elements[i] *= frac;
194     return *this;
195 }
196
197 Row Row::operator/(const Fraction &frac) const
198 {
199     Row temp(this->capacity);
200     for (unsigned col = 0; col < this->capacity; col++)
201         temp.push(elements[col] / frac);
202     return temp;
203 }
204
205 Row &Row::operator/=(const Fraction &frac)
206 {
207     for (unsigned i = 0; i < this->length; i++)
208         elements[i] /= frac;
209     return *this;
210 }
211
212 bool Row::operator==(const Row &other) const
213 {
214     if (this->capacity != other.capacity)
215         throw runtime_error("In order to compare two rows, there capacity must be same");
216
217     bool areEqual = true;
218     for (unsigned col = 0; col < this->capacity; col++)
219     {
220         if (elements[col] != other.elements[col])
221         {
222             areEqual = false;
223             break;
224         }
225     }
226     return areEqual;
227 }
228
229 bool Row::operator!=(const Row &other) const
230 {
231     return !(*this == other);
232 }
233
234 Fraction Row::operator[](const int &index) const
235 {
236     if (index < 0 || index >= this->length)
237         throw out_of_range("The index is out of range of the available length");
238     return elements[index];
239 }
240
241 //? friend functions:
242
243 ostream &operator<<(ostream &os, const Row &row)
244 {
245     os << "[";
246     for (unsigned col = 0; col < row.length; col++)
247     {
248         os << setw(10);
249         os << row.elements[col];
250         os << setw(4);
251     }
252     os << "]";
253     return os;
254 }
255
256 istream &operator>>(istream &is, Row &row)
257 {
258     for (unsigned col = 0; col < row.getCapacity(); col++)
259         is >> row.elements[col];
260     return is;
261 }
262 }
```

Matrix.cpp

```
1 #include "../includes/Matrix.h"
2
3 //? constructors and destructor:
4
5 Matrix::Matrix(const int &rows, const int &cols, Fraction **fractions)
6 {
7     allocateRows(rows);
8     allocateColumns(cols, fractions);
9 }
10
11 Matrix::~Matrix()
12 {
13     deallocateMatrix();
14 }
15
16 //? private functions:
17
18 void Matrix::allocateRows(const int &rows)
19 {
20     if (rows <= 0)
21         throw range_error("The number of rows must be greater than 0");
22
23     this->rows = rows;
24     this->body = new Row[rows];
25 }
26
27 void Matrix::allocateColumns(const int &cols, Fraction *fractions[])
28 {
29     if (cols <= 0)
30         throw range_error("The number of cols must be greater than 0");
31
32     this->cols = cols;
33     for (unsigned i = 0; i < this->rows; i++)
34         body[i] = Row(cols, fractions[i]);
35 }
36
37 void Matrix::deallocateMatrix()
38 {
39     delete[] body;
40     this->rows = 0;
41     this->cols = 0;
42 }
43
44 //? access modifiers:
45
46 void Matrix::fillRow(const int &row, const Fraction &frac)
47 {
48     if (row <= 0 || row > this->rows)
49         throw out_of_range("The row to be filled is out of range ");
50
51     body[row - 1].clear();
52     body[row - 1].fillRow(this->cols, frac);
53 }
54 void Matrix::fill(const Fraction &frac)
55 {
56     for (unsigned i = 0; i < this->rows; i++)
57         fillRow(i, frac);
58 }
59
60 //? EROs (Elementary row operations):
61
62 void Matrix::add(const int &rowDest, const int &rowSrc, const Fraction &multiplier)
63 {
64     if (rowDest <= 0 || rowDest > this->rows)
65         throw out_of_range("The destination row is out of range");
66     if (rowSrc <= 0 || rowSrc > this->rows)
67         throw out_of_range("The source row is out of range");
68
69     body[rowDest - 1] += body[rowSrc - 1] * multiplier;
70 }
71
72 void Matrix::multiply(const int &row, const Fraction &factor)
73 {
74     if (row <= 0 || row > this->rows)
75         throw out_of_range("The row is out of range");
76
77     body[row - 1] *= factor;
78 }
79
80 void Matrix::divide(const int &row, const Fraction &factor)
81 {
82     if (row <= 0 || row > this->rows)
83         throw out_of_range("The row is out of range");
84
85     body[row - 1] /= factor;
86 }
87
88 void Matrix::change(const int &row1, const int &row2)
89 {
90     if (row1 <= 0 || row1 > this->rows)
91         throw out_of_range("The row is out of range");
92     if (row2 <= 0 || row2 > this->rows)
93         throw out_of_range("The row is out of range");
94
95     Row row1Copy(body[row1 - 1]);
96     body[row1 - 1] = body[row2 - 1];
97     body[row2 - 1] = row1Copy;
98 }
99
100 //? helper functions:
101
102 void Matrix::sortRows()
103 {
104     for (unsigned pass = 1; pass < rows; pass++)
105     {
106         for (unsigned i = 0; i < rows - 1; i++)
107         {
108             unsigned currRow = body[i].startingZeroCount();
109             unsigned nextRow = body[i + 1].startingZeroCount();
110
111             if (currRow > nextRow)
112             {
113                 change(i + 1, i + 2);
114                 if (this->steps)
115                 {
116                     cout << "Change R" << i + 1 << " with R" << i + 2 << "\n";
117                     cout << *this << "\n";
118                 }
119             }
120             // if equal num of levels, change to put 1 on the top
121             else if (currRow == nextRow)
122             {
123                 // skip if there is no nonzero values in the row
124                 if (currRow >= this->cols || nextRow >= this->cols)
125                     continue;
126
127                 // checking the first non zero values
128                 if (body[i][currRow] != 1 && body[i][nextRow] == 1)
129                 {
130                     change(i + 1, i + 2);
131                     if (this->steps)
132                     {
133                         cout << "Change R" << i + 1 << " with R" << i + 2 << "\n";
134                         cout << *this << "\n";
135                     }
136                 }
137             }
138         }
139     }
140 }
141
142 void Matrix::toEchelon(const bool &reduced)
143 {
144     // array to hold target elements of each column
145     // target element is the element after which all elements are 0
146     Element targetElements[cols];
147
148     // maximum row of the target elements
149     unsigned maxRow = 0;
150     // maximum col of the target elements
151     unsigned maxCol = 0;
152
153     // sorting the rows
154     sortRows();
155
156     // iterating through the matrix column-wise
157     for (unsigned col = 0; col < cols; col++)
158     {
159         Element &target = targetElements[col];
160         for (unsigned row = 0; row < rows; row++)
161         {
162             const Fraction element = body[row][col];
163             // skip if element is 0
164             if (element == 0)
165                 continue;
166             // if it is the first column
167             if (col == 0)
168             {
169                 target.row = row + 1;
170                 target.col = col + 1;
171                 target.frac = element;
172                 break;
173             }
174
175             // if the element is to the far right of the previous target elements
176             const Element &prevTarget = targetElements[col - 1];
177             if (col > (prevTarget.col - 1) && row > (prevTarget.row - 1))
178             {
179                 target.row = row + 1;
180                 target.col = col + 1;
181                 target.frac = element;
182                 break;
183             }
184             // if there is no target element in this column
185             target.frac = 0;
186             target.row = rows;
187             target.col = cols;
188         }
189
190         // transforming the matrix
191         if (target.frac != 0)
192         {
193             // make target value 1
194             if (target.frac != 1)
195             {
196                 divide(target.row, target.frac);
197                 if (this->steps)
198                 {
199                     cout << "Dividing R" << target.row << " by " << target.frac << endl;
200                     cout << *this << "\n";
201                 }
202             }
203
204             // determine whether the result should be a reduced echelon or not
205             const unsigned startingRow = reduced ? 0 : target.row;
206             // make all elements of the column 0
207             for (unsigned row = startingRow; row < rows; row++)
208             {
209                 const Fraction oldVal = body[row][col];
210                 const Fraction newVal = oldVal * -1;
211
212                 // skip the element if it is the target element
213                 if (row == target.row - 1)
214                     continue;
215                 // skip the element if it is already 0
216                 if (oldVal == 0)
217                     continue;
218
219                 add(row + 1, target.row, newVal);
220
221                 if (this->steps)
222                 {
223                     cout << "Adding " << newVal << "*R" << target.row << " in R" << row + 1 << endl;
224                     cout << *this << "\n";
225                 }
226             }
227         }
228     }
229 }
230
231 //? friend functions:
232
233 ostream &operator<<(ostream &os, const Matrix &matrix)
234 {
235     for (unsigned row = 0; row < matrix.rows; row++)
236         os << matrix.body[row] << "\n";
237     return os;
238 }
```

main.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <exception>
4 #include "../includes/Matrix.h"
5 #include "./defs.cpp"
6 using namespace std;
7
8 int main()
9 {
10     // opening the input file
11     ifstream inputFile = ifstream("input.txt");
12     if (!inputFile.is_open())
13         throw runtime_error("Exception: Input file not found. Try again.");
14
15     // reading the dimensions of matrix
16     unsigned rows = 0;
17     unsigned cols = 0;
18     inputFile >> rows >> cols;
19     // allocate memory for the matrix
20     Fraction **matrix = new Fraction *[rows];
21     for (unsigned i = 0; i < rows; i++)
22         matrix[i] = new Fraction[cols];
23
24     // reading the elements
25     for (unsigned row = 0; row < rows; row++)
26     {
27         for (unsigned col = 0; col < cols; col++)
28         {
29             double x;
30             inputFile >> x;
31             matrix[row][col] = Fraction(x);
32
33             if (inputFile.eof())
34             {
35                 row = rows; // to break outer loop
36                 break;
37             }
38         }
39     }
40
41     // declaring the Matrix
42     Matrix m(rows, cols, matrix);
43
44     // add this line if you want to see each step
45     // m.showSteps(true);
46
47     cout << "* Original Matrix:\n"
48         << m << "\n";
49
50     m.toEchelon();
51     cout << "* Echelon Form:\n"
52         << m << "\n";
53
54     // pass true to transform to reduce form
55     m.toEchelon(true);
56     cout << "* Reduced Echelon Form:\n"
57         << m << "\n";
58
59     // deallocating memory
60     for (unsigned i = 0; i < rows; i++)
61         delete[] matrix[i];
62     delete[] matrix;
63     return 0;
64 }
```