Contents

1 Basic Test Results 2

2 Makefile 3

3 Matrix.hpp 4

4 MatrixExceptions.h 9

5 Rational.cpp 11

6 RationalExceptions.h 14

1 Basic Test Results

```
Makefile
Matrix.hpp
MatrixExceptions.h
Rational.cpp
RationalExceptions.h
g++ -Wall -std=c++11 -c Rational.cpp
g++ -Wall -std=c++11 -c GenericMatrixDriver.cpp
g++ -Wall -std=c++11 Rational.o GenericMatrixDriver.o -o GenericMatrixDriver

Running...
Opening tar file
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Importing files
OK
Compilation check...
Testing GenMat1.in-out
Running test...
OK
Compilation went without errors, BUT you must check to see if you got warnings!!!

========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 Makefile

```
CC = g++ -Wall -std=c++11
ALL = Matrix.hpp Rational.cpp RationalExcaptions.h MatrixExcaptions.h
O_FILE = Rational.o GenericMatrixDriver.o

defaulf: GenericMatrixDriver

GenericMatrixDriver: $(O_FILE)
    $(CC) $(O_FILE) -o GenericMatrixDriver

GenericMatrixDriver.o: Matrix.hpp Rational.h GenericMatrixDriver.cpp
    $(CC) -c GenericMatrixDriver.cpp

Rational.o: Rational.h Rational.cpp
    $(CC) -c Rational.cpp

tar:
    tar cvf ex3.tar $(ALL) Makefile

clean:
    rm -f *.o GenericMatrixDriver

.PHONY: tar clean
```

3 Matrix.hpp

```cpp
/**
* Matrix.hpp
*
* ------------------------------------------------------------------------------------------------
* General:        This class represents a generic matrix and can operate thing such as M+N, M*N
*                 and more.
*                 The matrix can be of any number type that support +, *, = , <<.
*
* MyMatrix            - default constructor
*                       create the zero element of the number
*
*                     - class constructor
*                       create a new colSizeXrowSize matrix from the values in the vector
*
*                     - copy constructor
*
*                     - move constructor
*
* ~MyMatrix             - class destructor
*
* operator=             - assign the other matrix into this
*
* operator<<          - ostream for the givan matrix
*
* operator+             - add this matrix with the other one and return the new matrix
*
* operator*             - matrix*matrix - multiply this matrix with the other one and return the new matrix
*                     - matrix*scalar - multiply this matrix with the scalar and return the new matrix
*                     - scalar*matrix - multiply this matrix with the scalar and return the new matrix
*
* transpose             - return a new matrix in size of colXrow from this rowXcol matrix
*                       such that new[i][j] = this[j][i]
*
* trace                 - return true iff the matrix is square and assign the trace to the givan reference.
*                       return false if the matrix is not a square and assign 0 to the givan reference.
*                       trace - the sum of the diagonal values
*/
#ifndef _MATRIX_HPP_
#define _MATRIX_HPP_

#include <vector>
#include <iostream>
#include "Rational.h"
#include "MatrixExceptions.h"

/**
* General:        This class represents a generic matrix and can operate thing such as M+N, M*N
*                 and more.
*                 The matrix can be of any number type that support +, *, = , <<.
*/
template <class T>
class Matrix
{
public:
    /**
    * default constructor
    * create 1X1 matrix with the default of the number
    */
    Matrix<T>()
```

```
60      {
61          _rows = 1;
62          _cols = 1;
63          T cell = T();
64          std::vector<T> row;
65          row.push_back(cell);
66          _matrix.push_back(row);
67      }
68
69      /**
70      * class constructor
71      * create a new colSizeXrowSize matrix from the values in the vector
72      */
73      Matrix<T>(const int& rows, const int& cols, const std::vector<T> cells) : _rows(rows), _cols(cols)
74      {
75          if ((int)cells.size() < rows * cols)
76          {
77              throw TooSmallVectorExcaption();
78          }
79          else if ((int)cells.size() > rows * cols)
80          {
81              throw TooBigVectorExcaption();
82          }
83          T cell;
84          for (int r = 0; r < _rows; r++)
85          {
86              std::vector<T> row;
87              for (int c = 0; c < _cols; c++)
88              {
89                  cell = cells[r*_cols + c];
90                  row.push_back(cell);
91              }
92              _matrix.push_back(row);
93          }
94      }
95
96      /**
97      * copy constructor
98      */
99      Matrix(const Matrix<T>& other) : _rows(other._rows), _cols(other._cols), _matrix(other._matrix)
100     {
101
102     }
103
104     /**
105     * move constructor
106     *
107     */
108     Matrix(Matrix<T> && other) :_rows(other._rows), _cols(other._cols),
109                                 _matrix(move(other._matrix))
110     {
111
112     }                                          bad_move_semantics
113
114     /**
115     * class destructor
116     *
117     */
118     ~Matrix()
119     {
120         _cols = 0;
121         _rows = 0;
122         _matrix.clear();
123     }
124
125     /**
126     * assign the other matrix into this using copy-and-swap.
127     *
```

```
128         */
129         Matrix& operator=(Matrix<T> other)
130         {
131             std::swap(_rows, other._rows);
132             std::swap(_cols, other._cols);
133             std::swap(_matrix, other._matrix);
134             return *this;
135         }
136 
137         /**
138         * add this matrix with the other one and return the new matrix
139         *
140         */
141         Matrix operator+(const Matrix<T>& other) const
142         {
143             if (_cols != other._cols || _rows != other._rows)
144             {
145                 throw WrongSizePlusExcaption();
146             }
147             else
148             {
149                 std::vector<T> temp;
150                 for (int r = 0; r < _rows; r++)
151                 {
152                     for (int c = 0; c < _cols; c++)
153                     {
154                         temp.push_back(_matrix[r][c] + other._matrix[r][c]);
155                     }
156                 }
157                 return Matrix<T>(_rows, _cols, temp);
158             }
159         }
160 
161         /**
162         * multiply this matrix with the other one and return the new matrix
163         *
164         */
165         Matrix operator*(const Matrix<T>& other) const
166         {
167             if (_cols != other._rows)
168             {
169                 throw WrongSizeMulExcaption();
170             }
171             else
172             {
173                 std::vector<T> temp;
174                 T cell;
175                 for (int r = 0; r < _rows; r++)
176                 {
177                     for (int c = 0; c < other._cols; c++)
178                     {
179                         cell = 0;
180                         for (int i = 0; i < _cols; i++)
181                         {
182                             cell = cell + _matrix[r][i] * other._matrix[i][c];
183                         }
184                         temp.push_back(cell);
185                     }
186                 }
187                 return Matrix<T>(_rows, other._cols, temp);
188             }
189         }
190 
191         /**
192         * matrix*scalar - multiply this matrix with the scalar and return the new matrix
193         *
194         */
195         Matrix operator*(const T& scalar) const
```

```
196     {
197
198         std::vector<T> temp;
199         T cell;
200         for (int r = 0; r < _rows; r++)
201         {
202             for (int c = 0; c < _cols; c++)
203             {
204                 cell = _matrix[r][c] * scalar;
205                 temp.push_back(cell);
206             }
207         }
208         return Matrix<T>(_rows, _cols, temp);
209     }
210
211     /**
212     * scalar*matrix - multiply this matrix with the scalar and return the new matrix
213     *
214     */
215     friend Matrix operator*(const T& scalar, const Matrix<T>& other)
216     {
217         return Matrix<T>(other * scalar);
218     }
219
220     /**
221     * return a new matrix in size of colXrow from this rowXcol matrix
222     * such that new[i][j] = this[j][i]
223     */
224     Matrix transpose() const
225     {
226         std::vector<T> temp;
227         for (int c = 0; c < _cols; c++)
228         {
229             for (int r = 0; r < _rows; r++)
230             {
231                 temp.push_back(_matrix[r][c]);
232             }
233         }
234         return Matrix<T>(_cols, _rows, temp);
235     }
236
237     /**
238     * return true iff the matrix is square and assign the trace to the givan reference.
239     * return false if the matrix is not a square and assign 0 to the givan reference.
240     * trace - the sum of the diagonal values
241     *
242     */
243     bool hasTrace(T& trace) const
244     {
245         trace = 0;
246         if (_rows != _cols)
247         {
248             return false;
249         }
250         else
251         {
252             for (int c = 0; c < _cols; c++)
253             {
254                 trace = trace + _matrix[c][c];
255             }
256             return true;
257         }
258     }
259
260     /**
261     * ostream for the givan matrix
262     *
263     */
```

```
264     friend std::ostream& operator<<(std::ostream &os, const Matrix<T> &matrix)
265     {
266         for (int r = 0; r < matrix._rows; r++)
267         {
268             os << matrix._matrix[r][0];
269             for (int c = 1; c < matrix._cols; c++)
270             {
271                 os << " " << matrix._matrix[r][c];
272             }
273             os << "\n";
274         }
275         return os;
276     }
277
278 private:
279     int _rows;
280     int _cols;
281     std::vector<std::vector<T>> _matrix;
282
283 };
284
285
286 /**
287 * specialization for hasTrace generic method
288 * calculate the trace in case of Rational number in order to save time in fixing the rational
289 * number only once(at the end)
290 */
291 template<>
292 bool Matrix<Rational>::hasTrace(Rational& trace) const
293 {
294     std::cout << "Performing specialized function of trace for Rational values" << std::endl;
295     trace = Rational(0);
296     if (_rows != _cols)
297     {
298         return false;
299     }
300     else
301     {
302         long int numerator = _matrix[0][0].getNumerator();
303         long int denominator = _matrix[0][0].getDenominator();
304         for (int c = 1; c < _cols; c++)
305         {
306             numerator = numerator * _matrix[c][c].getDenominator() +
307                 _matrix[c][c].getNumerator() * denominator;
308             denominator = denominator * _matrix[c][c].getDenominator();
309         }
310         trace = Rational(numerator, denominator);
311         return true;
312     }
313 }
314 #endif //_MATRIX_HPP_
```

4 MatrixExceptions.h

```
1  /**
2  * MatrixException.h
3  *
4  * excaption that may happend in Matrix class
5  * - exception in case of different in matrixs size in operator+
6  * - exception in case of different in matrixs size in operator*
7  * - exception in case of too much element in vector
8  * - exception in case of too less element in vector
9  */
10 #include <iostream>
11 #include <exception>
12
13 #ifndef MATRIX_EXCEPTIONS_H
14 #define MATRIX_EXCEPTIONS_H
15
16 /**
17 * exception in case of different in matrixs size in operator+
18 */
19 class WrongSizePlusExcaption : public std::exception
20 {
21 public:
22     /**
23     * message to display
24     */
25     virtual const char* what() const throw()
26     {
27         return "matrixs are in different size";
28     }
29 };
30 /**
31 * exception in case of different in matrixs size in operator*
32 */
33 class WrongSizeMulExcaption : public std::exception
34 {
35 public:
36     /**
37     * message to display
38     */
39     virtual const char* what() const throw()
40     {
41         return "Left column and right rows are in different size";
42     }
43 };
44 /**
45 * exception in case of too much element in vector
46 */
47 class TooBigVectorExcaption : public std::exception
48 {
49 public:
50     /**
51     * message to display
52     */
53     virtual const char* what() const throw()
54     {
55         return "vector have an extra values";
56     }
57 };
58 /**
59 * exception in case of too less element in vector
```

nice_exception_structure

```
60  */
61  class TooSmallVectorExcaption : public std::exception
62  {
63  public:
64      /**
65      * message to display
66      */
67      virtual const char* what() const throw()
68      {
69          return "vector missing values";
70      }
71  };
72  #endif //MATRIX_EXCEPTIONS_H
```

5 Rational.cpp

```cpp
/**
* Rational - class to represent a ratioanl number - a ratio between two
* integers.
*/
#include <string>
#include <sstream>
#include "Rational.h"
#include "RationalExceptions.h"

#define DEFAULT_NUMINATOR 0
#define DEFAULT_DENOMINATOR 1
#define FRACTION '/'

/**
* Constructors geting int or default which is set to 0
*/
Rational::Rational(const long int &value)
{
    _numerator = value;
    _denominator = 1;
}
/**
* Constructors geting numerator and denominator
*/
Rational::Rational(const long int &numerator, const long int &denominator)
{
    _numerator = numerator;
    {
        if (denominator == 0)
        {
            throw DivisionByZeroExceptions();
        }
        _denominator = denominator;
    }
    _fixZero();
    _makeRepresentationCoprime();
    _fixNegativity();
}
/**
* Constructors by string "numerator/denominator"
*/
Rational::Rational(const std::string &str)
{
    char split_char = FRACTION;
    std::string temp;
    std::string tempDenomerator;
    std::istringstream number(str);

    std::getline(number, temp, split_char);
    _numerator = std::stoi(temp);

    std::getline(number, temp, split_char);
    _denominator = std::stoi(temp);

    if (_denominator == 0)
    {
        throw DivisionByZeroExceptions();
    }

```

```
60      _fixZero();
61      _makeRepresentationCoprime();
62      _fixNegativity();
63  }
64
65  /**
66  * Constructors copy constructor
67  */
68  Rational::Rational(const Rational &other)
69  {
70      _numerator = other._numerator;
71      _denominator = other._denominator;
72  }
73  /**
74  * Constructors move constructor
75  */
76  Rational::Rational(Rational && other)
77  {
78      _numerator = other._numerator;
79      _denominator = other._denominator;
80  }
81
82  /**
83  * Assignment operator, use copy-and-swap idiom
84  */
85  Rational& Rational::operator=(Rational other)
86  {
87      std::swap(_numerator, other._numerator);
88      std::swap(_denominator, other._denominator);
89      return *this;
90  }
91
92  /**
93  * Returns the numerator
94  */
95  const long int Rational::getNumerator() const
96  {
97      return _numerator;
98  }
99  /**
100 * Returns the denominator
101 */
102 const long int Rational::getDenominator() const
103 {
104     return _denominator;
105 }
106
107 /**
108 * Summing two Rationals
109 * Addition should be calculated in the way we calculate addition of 2
110 * ratios (using their common denominator).
111 */
112 const Rational Rational::operator+(const Rational &other) const
113 {
114
115     Rational result(_numerator * other._denominator + other._numerator * _denominator,
116         _denominator * other._denominator);
117
118     result._fixZero();
119     result._makeRepresentationCoprime();
120     result._fixNegativity();
121
122     return result;
123
124 }
125 /**
126 * Multiplying operator for Rational
127 * Multiplication should be calculated in the way we calculate
```

```
128  * multiplication of 2 ratios (separately for the numerator and for the
129  * denominator).
130  */
131  const Rational Rational::operator*(const Rational &other) const
132  {
133      Rational result(_numerator * other._numerator, other._denominator * _denominator);
134
135      result._fixZero();
136      result._makeRepresentationCoprime();
137      result._fixNegativity();
138
139      return result;
140  }
141
142  /**
143  * operator<< for stream insertion
144  * The format is numerator/denominator w/o spaces or addional characters
145  */
146  std::ostream& operator<<(std::ostream &os, const Rational &number)
147  {
148      os << number._numerator << FRACTION << number._denominator;
149      return os;
150  }
151
152  void Rational::_fixZero()
153  {
154      if (_numerator == 0)
155      {
156          _numerator = DEFAULT_NUMINATOR;
157          _denominator = DEFAULT_DENOMINATOR;
158      }
159  }
160  void Rational::_fixNegativity()
161  {
162      if (_denominator < 0)
163      {
164          _denominator = -_denominator;
165          _numerator = -_numerator;
166      }
167  }
168  void Rational::_makeRepresentationCoprime()
169  {
170      if (_numerator != 1 && _denominator != 1)
171      {
172          long int gcd = _greatestCommonDivisor(_numerator, _denominator);
173          _numerator /= gcd;
174          _denominator /= gcd;
175      }
176  }
177
178  const long int Rational::_greatestCommonDivisor(const long int &a, const long int &b)
179  {
180      if (b == 0)
181      {
182          return a;
183      }
184      else
185      {
186          return _greatestCommonDivisor(b, a % b);
187      }
188  }
```

Better not to use recrstion

6 RationalExceptions.h

```
/**
* RationalException.h
*
* excaption that may happend in Rational class
* - exception in case of division by zero
*/
#include <iostream>
#include <exception>

#ifndef RATIONAL_EXCEPTIONS_H
#define RATIONAL_EXCEPTIONS_H

/**
* exception in case of division by zero
*/
class DivisionByZeroExceptions : public std::exception
{
public:
    /**
    * message to display
    */
    virtual const char* what() const throw()
    {
        return "division by zero";
    }
};

#endif // RATIONAL_EXCEPTIONS_H
```