TP3 : Gestion des exceptions

Paul Chaignon, Xavier Fraboulet

INSA de Rennes
4INFO, groupe 2.2

15 octobre 2013

1 Fractions

Listing 1 – Fraction.h

```
#include <stdexcept>
#include <iostream>

class Fraction {
private:
  int num;
  int den;
  bool overflowMultiplication(int, int) const;
  bool overflowAddition(int, int) const;
  bool overflowSoustraction(int, int) const;

public:
  Fraction(int);
  Fraction(int, int);
  Fraction operator+(const Fraction&) const;
  Fraction operator-(const Fraction&) const;
  Fraction operator/(const Fraction&) const;
  Fraction operator*(const Fraction&) const;
  double eval();
};
```

Listing 2 – Fraction.cpp

```
/**
 * \file Fraction.cpp
 * \brief Methodes de la classe Fraction
 * \author Paul Chaignon
```

```
5  * \author Xavier Fraboulet
6  * \version 1.0
7  * \date 09/10/13
8  */
9 #include "Fraction.h"
10
11 /**
12  * \fn Fraction
13  * \brief Constructeur avec denominateur a 1.
14  * \param[in] i Le numerateur (la valeur de la fraction).
15  */
16 Fraction::Fraction(int i) {
17   num = i;
18   den = 1;
19 }
20
21 /**
22  * \fn Fraction
23  * \brief Constructeur
24  * \param[in] num Le numerateur.
25  * \param[in] den Le denominateur, doit etre different de 0.
26  * \throw logic_error Si le denominateur est a 0.
27  */
28 Fraction::Fraction(int num, int den) {
29   if(den == 0) {
30     throw std::logic_error("Creation d'une fraction avec zero au denominateur");
31   }
32   this->num = num;
33   this->den = den;
34 }
35
36 /**
37  * \fn overflowMultiplication
38  * \brief Verifie qu'aucun overflow ou underflow n'a lieu lors de la
       multiplication de 2 entiers.
39  * \param[in] a Le premier entier.
40  * \param[in] b Le deuxieme entier.
41  * \return Vrai si un overflow ou underflow a lieu.
42  */
43 bool Fraction::overflowMultiplication(int a, int b) const {
44   if(abs(a) > std::numeric_limits<int>::max()/abs(b)) {
45     return true;
46   }
47   if(a > std::numeric_limits<int>::min()/b) {
48     return true;
49   }
50   return false;
51 }
52
```

```
53 /**
54  * \fn overflowAddition
55  * \brief Verifie qu'aucun overflow ou underflow n'a lieu lors de l'addition de 2
        entiers.
56  * \param[in] a Le premier entier.
57  * \param[in] b Le deuxieme entier.
58  * \return Vrai si un overflow ou underflow a lieu.
59  */
60 bool Fraction::overflowAddition(int a, int b) const {
61   if(a>0 && b>0 && a>std::numeric_limits<int>::max()-b) {
62     return true;
63   }
64   if(a<0 && b<0 && a<std::numeric_limits<int>::min()-b) {
65     return true;
66   }
67   return false;
68 }
69
70 /**
71  * \fn overflowSoustraction
72  * \brief Verifie qu'aucun overflow ou underflow n'a lieu lors de la soustraction
        de 2 entiers.
73  * \param[in] a Le premier entier.
74  * \param[in] b Le deuxieme entier.
75  * \return Vrai si un overflow ou underflow a lieu.
76  */
77 bool Fraction::overflowSoustraction(int a, int b) const {
78   if(a>0 && b<0 && a>std::numeric_limits<int>::max()+b) {
79     return true;
80   }
81   if(a<0 && b>0 && a<std::numeric_limits<int>::min()+b) {
82     return true;
83   }
84   return false;
85 }
86
87 /**
88  * \fn operator+
89  * \brief Redefinie l'operation d'addition pour deux fractions.
90  * \param[in] f La fraction a ajouter a la fraction courante.
91  * \return La fraction resultat.
92  * \throw logic_error Si un overflow ou underflow a lieu.
93  */
94 Fraction Fraction::operator+(const Fraction& f) const {
95   if(this->overflowMultiplication(f.num, this->den)) {
96     throw std::logic_error("Overflow sur le numerateur lors de l'addition.");
97   }
98   int mult1 = f.num * this->den;
99
```

```
100   if(this->overflowMultiplication(f.den, this->num)) {
101     throw std::logic_error("Overflow sur le numerateur lors de l'addition.");
102   }
103   int mult2 = f.den * this->num;
104 
105   if(this->overflowAddition(f.num, this->num)) {
106     throw std::logic_error("Overflow sur le numerateur lors de l'addition.");
107   }
108   int numerateur = mult1 + mult2;
109 
110   if(this->overflowMultiplication(f.num, this->num)) {
111     throw std::logic_error("Overflow sur le denominateur lors de l'addition.");
112   }
113   int deno = this->den * f.den;
114 
115   return Fraction(numerateur, deno);
116 }
117 
118 /**
119  * \fn operator-
120  * \brief Redefinie l'operation de soustraction pour deux fractions.
121  * \param[in] f La fraction a soustraire a la fraction courante.
122  * \return La fraction resultat.
123  * \throw logic_error Si un overflow ou underflow a lieu.
124  */
125 Fraction Fraction::operator-(const Fraction& f) const {
126   if(this->overflowMultiplication(f.num, this->den)) {
127     throw std::logic_error("Overflow sur le numerateur lors de la soustraction.");
128   }
129   int mult1 = f.num * this->den;
130 
131   if(this->overflowMultiplication(f.den, this->num)) {
132     throw std::logic_error("Overflow sur le numerateur lors de la soustraction.");
133   }
134   int mult2 = f.den * this->num;
135 
136   if(this->overflowSoustraction(f.num, this->num)) {
137     throw std::logic_error("Overflow sur le numerateur lors de la soustraction.");
138   }
139   int numerateur = mult1 - mult2;
140 
141   if(this->overflowMultiplication(f.num, this->num)) {
142     throw std::logic_error("Overflow sur le denominateur lors de la
          soustraction.");
143   }
144   int deno = this->den * f.den;
145 
146   return Fraction(numerateur, deno);
147 }
```

```
148
149 /**
150  * \fn operator/
151  * \brief Redefinie l'operation de division pour deux fractions.
152  * \param[in] f La fraction qui doit diviser la fraction courante.
153  * \return La fraction resultat.
154  * \throw logic_error Si un overflow ou underflow a lieu.
155  */
156 Fraction Fraction::operator/(const Fraction& f) const {
157   if(f.num == 0) {
158     throw std::logic_error("Division par zero.");
159   }
160   return this->operator*(Fraction(f.den, f.num));
161 }
162
163 /**
164  * \fn operator*
165  * \brief Redefinie l'operation de multiplication pour deux fractions.
166  * \param[in] f La fraction a multiplier a la fraction courante.
167  * \return La fraction resultat.
168  * \throw logic_error Si un overflow ou underflow a lieu.
169  */
170 Fraction Fraction::operator*(const Fraction& f) const {
171   if(this->overflowMultiplication(f.num, this->num)) {
172     throw std::logic_error("Overflow sur le numerateur lors de la
        multiplication.");
173   }
174   int numerateur = f.num * this->num;
175
176   if(this->overflowMultiplication(f.den, this->den)) {
177     throw std::logic_error("Overflow sur le numerateur lors de la
        multiplication.");
178   }
179   int deno = this->den * f.den;
180
181   return Fraction(numerateur, deno);
182 }
183
184 /**
185  * \fn eval
186  * \brief Evalue une fraction.
187  * \return La valeur decimale resultat de l'evaluation de la fraction.
188  */
189 double Fraction::eval() {
190   return num / den;
191 }
```

Listing 3 – main.cpp

```
1 #include "Fraction.h"
```

```
2 #include <cassert>
3 #include "stdafx.h"
4
5 /**
6  * Tests
7  */
8 void main() {
9   try {
10    Fraction(5, 0);
11    assert(false);
12  } catch(std::exception& e) {}
13
14  Fraction f1 = Fraction(10);
15  Fraction f2 = Fraction(0, 2);
16  try {
17    Fraction mult = f1 / f2;
18    assert(false);
19  } catch(std::exception& e) {}
20
21  try {
22    Fraction add = f1 + f2;
23    assert(false);
24  } catch(std::exception& e) {}
25 }
```

2 Chaines de caractères

Listing 4 – sequences.h

```
1
2 #ifndef SEQ_H
3 #define SEQ_H
4
5 #include <set>
6 #include <string>
7 #include <iostream>
8
9 /*! \brief Alphabet class. Used to validate a character against an alphabet. */
10 class alpha {
11 public:
12   /*! \brief Constructor. Builds an alphabet from an input string.
13   * \param s String used as alphabet */
14   alpha(const std::string & s) {
15     for(std::string::const_iterator c = s.begin(); c!=s.end(); c++) {
```

```
16        _cs.insert(*c);
17      }
18    }
19
20    /*! \brief Checks if the given character is in the alphabet.
21    * \param c Character to validate against the alphabet.
22    * \return True if the character is in the alphabet. False otherwise. */
23    bool is_in_alpha(char c) const {
24      return _cs.find(c) != _cs.end();
25    }
26
27  private:
28    /*! The alphabet */
29    std::set<char> _cs;
30
31  };
32
33
34  /*! \brief Generic sequence with alphabet. */
35  class seqmac {
36  public:
37    /*! \brief Constructor. Builds the sequence from an input string and an
          alphabet.
38    * \param seq Sequence of characters
39    * \param name Name of the sequence
40    * \param alphabet alphabet to be used to verify the input sequence */
41    seqmac (const std::string & seq, const std::string & name, const std::string &
          alphabet);
42
43    /*! \brief Output the sequence to an output stream.
44    * \param os Output stream
45    * \param seq Sequence
46    * \return The output stream */
47    friend std::ostream & operator<< (std::ostream & os, const seqmac & seq);
48
49  protected:
50    /*! Sequence */
51    std::string _seq;
52    /*! Name */
53    std::string _name;
54
55  private:
56    /*! Alphabet */
57    const alpha _alph;
58
59    /*! \brief Formatted output of the sequence.
60    * \param os Output stream */
61    void writeseq(std::ostream & os) const
62    {
```

```
63      os << "SEQUENCE" << std::endl << "--------------\n";
64      os << "Nom : " << _name << std::endl;
65      os << "Seq : " << _seq << std::endl;
66      os << "aa  : " << _seq.size() << std::endl;
67    }
68 };
69
70
71 /*! \brief Specialized sequence for proteins. */
72 class seqprot: public seqmac {
73 public:
74   /*! \brief Constructor. Builds the proteine sequence from an input string with
          alphabet check. The alphabet is hardcoded for protein characters.
75   * \param seq Sequence of characters
76   * \param name Name of the sequence */
77   seqprot(const std::string & seq="", const std::string & name="") : seqmac(seq,
          name, "ACDEFGHIKLMNPQRSTV") {
78
79   }
80 };
81
82
83 /*! \brief Specialized sequence for ADN. */
84 class seqadn: public seqmac {
85 public:
86   /*! \brief Constructor. Builds the ADN sequence from an input string with
          alphabet check. The alphabet is hardcoded for ADN characters.
87   * \param seq Sequence of characters
88   * \param name Name of the sequence */
89   seqadn(const std::string & seq="", const std::string & name="");
90 };
91
92
93 /*! \brief Specialized sequence for ARN. */
94 class seqarn: public seqmac {
95 public:
96   /*! \brief Constructor. Builds the ARN sequence from an input string with
          alphabet check. The alphabet is hardcoded for ARN characters.
97   * \param seq Sequence of characters
98   * \param name Name of the sequence */
99   seqarn(const std::string & seq="", const std::string & name="");
100 };
101
102
103 #endif
```

Listing 5 – sequences.cpp

```
1 #include "sequences.h"
2
```

```
3  using namespace std;
4
5  ostream & operator<<(ostream & os, const seqmac & s) {
6    s.writeseq(os);
7    return os;
8  }
9
10 seqmac::seqmac(const string & seq, const string & name, const string & alphabet):
       _alph(alphabet), _name(name) {
11   string s = "";
12   for(string::const_iterator c = seq.begin(); c!=seq.end(); c++) {
13     if(_alph.is_in_alpha(*c)) {
14       s += *c;
15     } else {
16       throw std::invalid_argument("La lettre n'appartient pas a l'alphabet
             autorise");
17     }
18   }
19   _seq = s;
20 }
21
22 seqadn::seqadn(const string & seq, const string & name): seqmac(seq, name,
       "CGAT") {
23   if(_seq.size() % 3 != 0) {
24     throw std::invalid_argument("La sequence doit avoir une taille multiple de
         3");
25   }
26
27   bool start = false;
28   bool stop = false;
29   for(int i = 0; i < _seq.size(); i+=3) {
30     string sub = _seq.substr(i, 3);
31     if(!start) {
32       start = (sub == "ATG");
33     }
34
35     if(sub=="TAA" || sub=="TAG" || sub=="TGA") {
36       if(!start) {
37         throw std::invalid_argument("Le codon START doit etre avant le codon
               STOP");
38       } else {
39         stop = true;
40       }
41     }
42   }
43
44   if (!start) {
45     throw std::invalid_argument("Le codon START est manquant");
46   }
```

```
47   if (!stop) {
48     throw std::invalid_argument("Le codon STOP est manquant");
49   }
50 }
51
52 seqarn::seqarn(const string & seq, const string & name): seqmac(seq, name,
       "ACGU") {
53   if(_seq.size() % 3 != 0) {
54     throw std::invalid_argument("La sequence doit avoir une taille multiple de
         3");
55   }
56
57   bool start = false;
58   bool stop = false;
59   for(int i=0; i<_seq.size(); i+=3) {
60     string sub = _seq.substr(i, 3);
61     if(!start) {
62       start = sub == "AUG";
63     }
64
65     if(sub=="UAA" || sub=="UAG" || sub=="UGA") {
66       if(!start) {
67         throw std::invalid_argument("Le codon START doit etre avant le codon
             STOP");
68       } else {
69         stop = true;
70       }
71     }
72   }
73
74   if(!start) {
75     throw std::invalid_argument("Le codon START est manquant");
76   }
77   if(!stop) {
78     throw std::invalid_argument("Le codon STOP est manquant");
79   }
80 }
```