CPPLI : TD 8 : C++ : Classe

Nicolas Vansteenkiste Romain Absil Jonas Beleho *
(Esi – He2b)

Année académique 2019 – 2020

Ce TD [1] aborde la création et l'utilisation de classes [2] en C++.
Réalisez ce TD en exploitant le standard C++17 [3].

1. Énumération fortement typée `Sign`

Le fichier `sign_incomplete.h` [4] contient la définition de l'`enum class Sign` ainsi que celles de diverses fonctions qui l'utilisent, mais *sans* les implémentations de ces dernières.

Toute la documentation de l'énumération fortement typée [5] (*strongly typed enumeration*) `Sign` est disponible au format html, après décompression du fichier `html.7z` [6].

Le fichier `sign_incomplete.h` est reproduit en annexe A, mais référez-vous plutôt à sa documentation html.

---

*Et aussi, lors des années passées : Monica Bastreghi, Stéphan Monbaliu, Anne Rousseau et Moussa Wahid.

1. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/td08_cpp_withAppendix.pdf (consulté le 27 novembre 2019).
2. https://en.cppreference.com/w/cpp/language/class (consulté le 24 novembre 2019).
3. https://isocpp.org/search/google?q=c%2B%2B17 (consulté le 27 novembre 2019).
4. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/sign_incomplete.h (consulté le 27 novembre 2019).
5. https://docs.microsoft.com/fr-fr/cpp/cpp/enumerations-cpp?view=vs-2019 (consulté le 24 novembre 2019).
6. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/html.7z (consulté le 27 novembre 2019).

Ex. 8.1 Lisez la documentation de l'`enum class Sign` et de ses fonctions.

Ex. 8.2 Prenez le fichier `sign_incomplete.h` et :

1. renommez-le en `sign.h` ;
2. adaptez là où nécessaire son contenu à son nouveau nom ;
3. déplacez l'énumération `Sign` et ses fonctions dans l'espace de noms [7] (ou de nommage) `gxxxxx` où `xxxxx` est votre numéro d'étudiant ;
4. complétez et modifiez les fonctions marquées d'un commentaire `// TODO` de sorte qu'elles respectent les spécifications renseignées dans la documentation ;
5. testez exhaustivement !

2. Classe `Fraction`

Le fichier `fraction_incomplete.h` [8] contient la définition de la `class Fraction` ainsi que celles de diverses fonctions qui l'utilisent. Les implémentations de celles-ci ainsi que des méthodes de `Fraction` manquent dans les fichiers `fraction_incomplete.h` et `fraction_incomplete.cpp` [9].

La documentation de cette classe et de ces fonctions est disponible au format html après décompression du fichier `html.7z`.

Les fichiers `fraction_incomplete.h` et `fraction_incomplete.cpp` sont reproduits en annexe B. Il est cependant vivement conseillé de plutôt se référer à la documentation html !

Ex. 8.3 Lisez la documentation de la `class Fraction` et de ses fonctions.

Ex. 8.4 Prenez les fichiers `fraction_incomplete.h` et `fraction_incomplete.cpp` et :

1. renommez-le en `fraction.h` et `fraction.cpp`, respectivement ;
2. adaptez leurs contenus à leurs nouveaux noms, ainsi qu'au nouveau nom donné au fichier `sign_incomplete.h` à l'exercice 2 ;
3. déplacez la classe `Fraction` ainsi que ses fonctions dans l'espace de noms `gxxxxx` où `xxxxx` est votre numéro d'étudiant ;
4. complétez et modifiez les fichiers `fraction.h` et `fraction.cpp` aux endroits marqués d'un commentaire `// TODO` de sorte que leurs contenus respectent les spécifications renseignées dans la documentation ;
5. testez exhaustivement !

7. https://en.cppreference.com/w/cpp/language/namespace (consulté le 24 novembre 2019).
8. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/fraction_incomplete.h (consulté le 27 novembre 2019).
9. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/fraction_incomplete.cpp (consulté le 27 novembre 2019).

2.1. Variante `constexpr`

Le fichier `fraction_constexpr_incomplete.h` [10] contient la définition d'une variation de la `class Fraction`. Dans cette variante, toutes les méthodes et toutes les fonctions sont marquées `constexpr` [11]. Les implémentations des méthodes et fonctions sont manquantes dans le fichier `fraction_constexpr_incomplete.h`.

La documentation de cette classe et de ces fonctions est disponible au format html après décompression du fichier `html_constexpr.7z` [12].

Le fichiers `fraction_constexpr_incomplete.h` est reproduit en annexe C. Il est cependant vivement conseillé de plutôt se référer à la documentation html !

Ex. 8.5 Prenez le fichier `fraction_constexpr_incomplete.h` et :

1. renommez-le en `fraction_constexpr.h` ;
2. adaptez son contenu à son nouveau nom, ainsi qu'au nouveau nom donné au fichier `sign_incomplete.h` à l'exercice 2 ;
3. déplacez la classe `Fraction` ainsi que ses fonctions dans l'espace de noms `gxxxxx` où `xxxxx` est votre numéro d'étudiant ;
4. complétez et modifiez le fichier `fraction_constexpr.h` aux endroits marqués d'un commentaire `// TODO` de sorte que son contenu respecte les spécifications renseignées dans la documentation ;
5. testez exhaustivement !

3. Mise en œuvre

Les fichiers `data_fraction.h` [13] et `data_fraction.cpp` [14] contiennent les définitions et implémentations de deux fonctions, `data_signed()` et `data_unsigned()`, pour la génération de fractions. Les prototypes de ces fonctions sont précédés de quelques commentaires qu'il vous est conseillé de lire.

Ces fichiers sont reproduits en annexe D. Par ailleurs, `data_fraction.cpp` inclut le fichier `random.hpp` [15] reproduit en annexe E.

10. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/fraction_constexpr_incomplete.h (consulté le 27 novembre 2019).
11. https://en.cppreference.com/w/cpp/language/constexpr (consulté le 24 novembre 2019).
12. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/html_constexpr.7z (consulté le 27 novembre 2019).
13. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/data_fraction.h (consulté le 27 novembre 2019).
14. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/data_fraction.cpp (consulté le 27 novembre 2019).
15. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td08_cpp/ressource/random/random.hpp (consulté le 27 novembre 2019).

3.1. Variante `signed`

Ex. 8.6 Utilisez la fonction `data_signed()` pour tester le constructeur de `Fraction` à deux arguments `int`. Pour ce faire, invoquez cette fonction et utilisez toutes les `std::pair<int, int>` qu'elle retourne pour garnir un `std::vector`[16] de `Fraction`. Cependant, tenez à jour un compteur des `std::pair`[17] générant une erreur lors de l'instanciation d'une fraction.

Affichez le compteur d'erreurs, la taille du `std::vector<Fraction>` et le contenu de ce dernier.

Voici un affichage possible correspondant à cet exercice :

```
error_count: 7
fractions.size(): 43

fractions content:
7/6 -4 0 -7/3 3/7 -5/7 -3/7 1/3 -1/3 1/6 -3 -1/3 3/2 4/7 -4/3 1/2
-7/2 -4/7 3/4 -1/3 1/3 3 4/3 2/5 0 -6/7 0 2/3 -7/6 0 -1/2 0 -1 1
-5/6 -7/3 3 4/3 7/6 4/3 1/5 -1/5 3
```

3.2. Tri via pointeurs

Dans l'exercice qui suit, il s'agit de trier dans l'ordre croissant les fractions du `std::vector<Fraction>` obtenu à l'Ex. 8.6. Si on tente de le trier directement avec l'algorithme `std::sort()`[18], un gros problème se pose ! La classe `Fraction` est immuable car tous ses attributs sont `const`. Il est dès lors impossible de réassigner les contenus des cellules du `std::vector` de `Fraction` et donc de les modifier ou encore de trier le `std::vector`.

Il existe heureusement des parades. On peut construire, élément par élément, un nouveau `std::vector` dont le contenu est identique à celui à trier si ce n'est qu'il est précisément trié. C'est fastidieux.

Une parade alternative consiste à produire un `std::vector` de *pointeurs* de `Fraction`. Le premier élément du `std::vector` de pointeurs pointe sur la première fraction du `std::vector` de `Fraction`, le deuxième pointeur sur la deuxième `Fraction`, etc. jusqu'au dernier pointeur pointant sur la dernière `Fraction` du `std::vector` de `Fraction`. Ensuite, à la place de trier le `std::vector` de `Fraction` — opération impossible —, on trie le `std::vector` de pointeurs. En parcourant alors le `std::vector` de pointeurs trié et en déréférençant ceux-ci, le résultat est similaire au tri du `std::vector` de `Fraction` ! C'est cette approche que nous adoptons dans l'exercice suivant.

Ex. 8.7 À la suite de votre code réponse à l'Ex. 8.6, produisez un `std::vector` de pointeurs de `const Fraction` dont chaque élément pointe sur l'élément de même

16. https://en.cppreference.com/w/cpp/container/vector (consulté le 27 novembre 2019).
17. https://en.cppreference.com/w/cpp/utility/pair (consulté le 27 novembre 2019).
18. https://en.cppreference.com/w/cpp/algorithm/sort (consulté le 24 novembre 2019).

index du `std::vector` de `Fraction` construit initialement. Les pointeurs sont de type `const Fraction *` pour rendre impossible toute modification par inadvertance des `Fraction` du premier `std::vector`. L'algorithme `std::transform()` [19] peut être utile.

Affichez le déréférencement du contenu du `std::vector` de pointeurs avant le tri, triez les pointeurs dans l'ordre croissant des fractions pointées puis affichez le déréférencement du contenu du `std::vector` de pointeurs après le tri.

Avec les mêmes données que celles de l'Ex. 8.6, voici un affichage possible :

```
before sorting (pointer):
7/6 -4 0 -7/3 3/7 -5/7 -3/7 1/3 -1/3 1/6 -3 -1/3 3/2 4/7 -4/3 1/2
-7/2 -4/7 3/4 -1/3 1/3 3 4/3 2/5 0 -6/7 0 2/3 -7/6 0 -1/2 0 -1 1
-5/6 -7/3 3 4/3 7/6 4/3 1/5 -1/5 3

after sorting (pointer):
-4 -7/2 -3 -7/3 -7/3 -4/3 -7/6 -1 -6/7 -5/6 -5/7 -4/7 -1/2 -3/7
-1/3 -1/3 -1/3 -1/5 0 0 0 0 0 1/6 1/5 1/3 1/3 2/5 3/7 1/2 4/7 2/3
3/4 1 7/6 7/6 4/3 4/3 4/3 3/2 3 3 3
```

3.3. Variante `unsigned`

Ex. 8.8 Utilisez la fonction `data_unsigned()` des fichiers `data_fraction.h` et `data_fraction.cpp` pour tester le constructeur de `Fraction` à trois arguments : un `Sign` et deux `unsigned`. Pour ce faire, invoquez cette fonction et utilisez tous les `std::tuple<int, unsigned, unsigned>` qu'elle retourne pour garnir un `std::vector` de `Fraction` [20]. Cependant, tenez à jour un compteur des `std::tuple` [21] générant une erreur lors de l'instanciation d'une fraction.

Affichez le compteur d'erreurs, la taille du `std::vector<Fraction>` et le contenu de ce dernier.

Voici un affichage possible correspondant à cet exercice :

```
error_count: 33
fractions.size(): 67

before sorting:
-1/6 0 1/2 1 0 -7/10 -1/3 7 1 3/5 -1 1/2 -6 -5/4 0 5/3 2/9 8/5
3/2 1 10/9 -1 -2/3 1/2 7/8 1 -7/5 7/2 1 1/2 -5/9 -1 0 3/4 7/2
7/9 0 -2/5 2 3/8 4/3 0 2/9 -2/3 -9 1 -1/3 8/7 7/6 -3 -7 5/6 -1/5
4/3 3 -2 7/5 7/10 -7/3 -3/7 2/9 5/8 -1/7 2/3 10/7 -8 -9/8
```

19. https://en.cppreference.com/w/cpp/algorithm/transform (consulté le 24 novembre 2019).
20. Vous pouvez éventuellement purger et recycler le `std::vector<Fraction>` de l'Ex. 8.6.
21. https://en.cppreference.com/w/cpp/utility/tuple (consulté le 27 novembre 2019).

3.4. Tri via `reference_wrapper`

Dans l'exercice qui suit, il s'agit de trier dans l'ordre décroissant les fractions du `std::vector<Fraction>` obtenu à l'Ex. 8.8.

Plutôt que de trier ce `std::vector` à travers un `std::vector<const Fraction *>` comme dans la section 3.2, nous allons ici mettre en œuvre une nouvelle parade. Cette alternative consiste à produire un `std::vector` de `std::reference_wrapper` [22] de `Fraction`. Le premier élément du `std::vector` de `std::reference_wrapper` réfère la première fraction du `std::vector` de `Fraction`, le deuxième `std::reference_wrapper` la deuxième `Fraction`, etc. jusqu'au dernier `std::reference_wrapper` référant la dernière `Fraction` du `std::vector` de `Fraction`. Ensuite, à la place de trier le `std::vector` de `Fraction` — opération impossible —, on trie le `std::vector` de `std::reference_wrapper`. Ceci est possible car contrairement aux *lvalue references* [23], les `std::reference_wrapper` se détachent de l'objet référencé par leur opérateur d'assignation [24].

Ex. 8.9 À la suite de votre code réponse à l'Ex. 8.8, produisez un `std::vector` de `std::reference_wrapper<const Fraction>` dont chaque élément réfère l'élément de même index du `std::vector` de `Fraction` construit initialement. Ce sont des `const Fraction` qui sont enveloppées pour rendre impossible toute modification par inadvertance des `Fraction` du premier `std::vector`. L'algorithme `std::copy()` [25] peut être utile.

Affichez le déréférencement du contenu du `std::vector` de `std::reference_wrapper` avant le tri, triez les `std::reference_wrapper` dans l'ordre décroissant des fractions référencées puis affichez après le tri le déréférencement du contenu du `std::vector` de `std::reference_wrapper`.

Avec les mêmes données que celles de l'Ex. 8.8, voici un affichage possible :

```
before sorting (reference_wrapper):
-1/6 0 1/2 1 0 -7/10 -1/3 7 1 3/5 -1 1/2 -6 -5/4 0 5/3 2/9 8/5
3/2 1 10/9 -1 -2/3 1/2 7/8 1 -7/5 7/2 1 1/2 -5/9 -1 0 3/4 7/2
7/9 0 -2/5 2 3/8 4/3 0 2/9 -2/3 -9 1 -1/3 8/7 7/6 -3 -7 5/6 -1/5
4/3 3 -2 7/5 7/10 -7/3 -3/7 2/9 5/8 -1/7 2/3 10/7 -8 -9/8

after sorting (reference_wrapper):
7 7/2 7/2 3 2 5/3 8/5 3/2 10/7 7/5 4/3 4/3 7/6 8/7 10/9 1 1 1 1
1 1 7/8 5/6 7/9 3/4 7/10 2/3 5/8 3/5 1/2 1/2 1/2 1/2 3/8 2/9 2/9
2/9 0 0 0 0 0 0 -1/7 -1/6 -1/5 -1/3 -1/3 -2/5 -3/7 -5/9 -2/3 -2/3
```

22. https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper (consulté le 24 novembre 2019).
23. https://en.cppreference.com/w/cpp/language/reference#Lvalue_references (consulté le 24 novembre 2019).
24. https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper/operator%3D (consulté le 24 novembre 2019).
25. https://en.cppreference.com/w/cpp/algorithm/copy (consulté le 24 novembre 2019).

```
-7/10 -1 -1 -1 -9/8 -5/4 -7/5 -2 -7/3 -3 -6 -7 -8 -9
```

A. Énumération fortement typée `Sign`

```cpp
/*!
 * \file sign_incomplete.h
 *
 * \brief Définition de l'énumération fortement typée nvs::Sign
 *        et de fonctions associées.
 */
#ifndef SIGN_INCOMPLETE_H
#define SIGN_INCOMPLETE_H

#include <string>
#include <ostream>

/*!
 * \brief Espace de nom de Nicolas Vansteenkiste.
 */
namespace nvs
{

/*!
 * \brief Énumération fortement typée représentant un [signe]
 *        (https://fr.wikipedia.org/wiki/Signe_(arithm%C3%A9tique))
 *        au sens mathématique.
 */
enum class Sign
{
    /*!
     * \brief Constante d'énumération destinée à représenter le
     *        signe « plus » (+), c'est-à-dire celui d'un
     *        [nombre (strictement) positif]
     *        (https://fr.wikipedia.org/wiki/Nombre_positif).
     *
     * Lorsqu'on transtype Sign::PLUS en entier, on obtient la
     * valeur +1.
     */
    PLUS = +1,

    /*!
     * \brief Constante d'énumération destinée à représenter le
```

```
39          *         signe « moins » (-), c'est-à-dire celui d'un
40          *         [nombre (strictement) négatif]
41          *         (https://fr.wikipedia.org/wiki/Nombre_n%C3%A9gatif).
42          *
43          * Lorsqu'on transtype Sign::MINUS en entier, on obtient la
44          * valeur -1.
45          */
46         MINUS = -1,
47
48         /*!
49          * \brief Constante d'énumération destinée à représenter le
50          *         signe du nombre [zéro]
51          *         (https://fr.wikipedia.org/wiki/Z%C3%A9ro).
52          *
53          * Lorsqu'on transtype Sign::ZERO en entier, on obtient la
54          * valeur 0.
55          */
56         ZERO = 0
57 };
58
59 // prototypes (pas vraiment nécesssaire si fonction inline)
60
61 /*!
62  * \brief Fonction retournant le signe d'un entier donné.
63  *
64  * \param value l'entier dont on désire le signe.
65  *
66  * \return le signe de `value`.
67  */
68 constexpr Sign sign(int value);
69
70 /*!
71  * \brief Fonction retournant le signe opposé d'un signe donné.
72  *
73  * On a :
74  *
75  * |    sign     | opposite(sign) |
76  * |:-----------:|:--------------:|
77  * | Form::PLUS  |  Form::MINUS   |
78  * | Form::MINUS |  Form::PLUS    |
79  * | Form::ZERO  |  Form::ZERO    |
80  *
81  * \param sign signe dont on désire obtenir l'opposé.
82  *
```

```
83  * \return le signe opposé de `sign`.
84  *
85  * \see operator-(Sign).
86  */
87 constexpr Sign opposite(Sign sign);
88
89 /*!
90  * \brief Opérateur retournant le signe opposé d'un signe donné.
91  *
92  * On a :
93  *
94  * |    sign     |     - sign    |
95  * |:-----------:|:-------------:|
96  * | Form::PLUS  |  Form::MINUS  |
97  * | Form::MINUS |  Form::PLUS   |
98  * | Form::ZERO  |  Form::ZERO   |
99  *
100 * \param sign signe dont on désire obtenir l'opposé.
101 *
102 * \return le signe opposé de `sign`.
103 *
104 * \see opposite(Sign).
105 */
106 constexpr Sign operator-(Sign sign);
107
108 /*!
109  * \brief Fonction retournant le signe produit de deux nvs::Sign.
110  *
111  * La [règle des signes]
112  * (https://fr.wikipedia.org/wiki/Signe_(arithm%C3%A9tique)#R%C3%A8gle_des_signes)
113  * est étendue pour prendre en compte le signe de zéro :
114  *
115  * |     lhs     |     rhs     | product(lhs, rhs) |
116  * |:-----------:|:-----------:|:-----------------:|
117  * | Form::PLUS  | Form::PLUS  |     Form::PLUS    |
118  * | Form::PLUS  | Form::MINUS |     Form::MINUS   |
119  * | Form::PLUS  | Form::ZERO  |     Form::ZERO    |
120  * | Form::MINUS | Form::PLUS  |     Form::MINUS   |
121  * | Form::MINUS | Form::MINUS |     Form::PLUS    |
122  * | Form::MINUS | Form::ZERO  |     Form::ZERO    |
123  * | Form::ZERO  | Form::PLUS  |     Form::ZERO    |
124  * | Form::ZERO  | Form::MINUS |     Form::ZERO    |
125  * | Form::ZERO  | Form::ZERO  |     Form::ZERO    |
126  *
```

```
127  * \param lhs un premier Sign.
128  * \param rhs un second Sign.
129  *
130  * \return le signe produit de `lhs` et `rhs` comme dans le tableau
131  *         ci-dessus.
132  *
133  * \see operator*(Sign, Sign).
134  */
135 constexpr Sign product(Sign lhs, Sign rhs);
136
137 /*!
138  * \brief Opérateur retournant le signe produit de deux nvs::Sign.
139  *
140  * La [règle des signes]
141  * (https://fr.wikipedia.org/wiki/Signe_(arithm%C3%A9tique)#R%C3%A8gle_des_signes)
142  * est étendue pour prendre en compte le signe de zéro :
143  *
144  * |     lhs     |     rhs     |    lhs * rhs   |
145  * |:-----------:|:-----------:|:--------------:|
146  * | Form::PLUS  | Form::PLUS  |   Form::PLUS   |
147  * | Form::PLUS  | Form::MINUS |   Form::MINUS  |
148  * | Form::PLUS  | Form::ZERO  |   Form::ZERO   |
149  * | Form::MINUS | Form::PLUS  |   Form::MINUS  |
150  * | Form::MINUS | Form::MINUS |   Form::PLUS   |
151  * | Form::MINUS | Form::ZERO  |   Form::ZERO   |
152  * | Form::ZERO  | Form::PLUS  |   Form::ZERO   |
153  * | Form::ZERO  | Form::MINUS |   Form::ZERO   |
154  * | Form::ZERO  | Form::ZERO  |   Form::ZERO   |
155  *
156  * \param lhs un premier Sign.
157  * \param rhs un second Sign.
158  *
159  * \return le signe produit de `lhs` et `rhs` comme dans le tableau
160  *         ci-dessus.
161  *
162  * \see product(Sign, Sign).
163  */
164 constexpr Sign operator*(Sign lhs, Sign rhs);
165
166 /*!
167  * \brief Fonction de conversion d'un nvs::Sign en std::string.
168  *
169  * \param sign le signe à convertir.
170  *
```

```
171  * \return une std::string représentant `sign`.
172  */
173 inline std::string to_string(Sign sign);
174
175 /*!
176  * \brief Opérateur d'injection d'un nvs::Sign dans un flux en
177  *        sortie.
178  *
179  * \param out le flux dans lequel l'injection est réalisée.
180  * \param sign le signe à injecter.
181  *
182  * \return le flux après injection.
183  */
184 inline std::ostream & operator<<(std::ostream & out, Sign sign);
185
186 // implémentations
187
188 constexpr Sign sign(int value)
189 {
190     // TODO
191     return { };
192 }
193
194 constexpr Sign opposite(Sign sign)
195 {
196     // TODO
197     return { };
198 }
199
200 constexpr Sign operator-(Sign sign)
201 {
202     // TODO
203     return { };
204 }
205
206 constexpr Sign product(Sign lhs, Sign rhs)
207 {
208     // TODO
209     return { };
210 }
211
212 constexpr Sign operator*(Sign lhs, Sign rhs)
213 {
214     // TODO
```

```
215     return { };
216 }
217
218 inline std::string to_string(Sign sign)
219 {
220     // TODO
221     return { };
222 }
223
224 inline std::ostream & operator<<(std::ostream & out, Sign sign)
225 {
226     // TODO
227     return out;
228 }
229
230 }
231
232 #endif // SIGN_INCOMPLETE_H
```

B. Classe `Fraction`

B.1. Fichier d'en-têtes

```
1  /*!
2   * \file fraction_incomplete.h
3   *
4   * \brief Définition de la classe nvs::Fraction ainsi que d'un
5   *        type (nvs::Fraction::Form) et de fonctions associées.
6   */
7  #ifndef FRACTION_INCOMPLETE_H
8  #define FRACTION_INCOMPLETE_H
9
10 #include <string>
11 #include <ostream>
12 #include <tuple>
13
14 #include "sign_incomplete.h"
15
16 /*!
17  * \brief Espace de nom de Nicolas Vansteenkiste.
18  */
19 namespace nvs
20 {
```

```
21
22 /*!
23  * \brief Classe représentant une [fraction]
24  * (https://fr.wikipedia.org/wiki/Fraction_(math%C3%A9matiques)).
25  *
26  * Comme caractéristique principale, indiquons la sauvegarde du
27  * signe sous la forme d'un nvs::Sign tandis que les numérateur
28  * et dénominateur sont stockés sous une forme non signée.
29  *
30  * Par ailleurs, toute fraction est toujours stockée sous forme
31  * simplifiée, c'est-à-dire comme une [fraction irréductible]
32  * (https://fr.wikipedia.org/wiki/Fraction_irr%C3%A9ductible).
33  *
34  * Outre les méthodes de la classe Fraction, il existe de
35  * multiples fonctions pour les utiliser. Elles sont définies
36  * dans l'espace de nom \ref nvs dans le fichier fraction.h.
37  *
38  */
39 class Fraction
40 {
41     /*!
42      * \brief [Signe]
43      *        (https://fr.wikipedia.org/wiki/Signe_(arithm%C3%A9tique))
44      *        de la fraction.
45      */
46     const Sign sign_;
47
48     /*!
49      * \brief [Valeur absolue]
50      *        (https://fr.wikipedia.org/wiki/Valeur_absolue) du
51      *        [numérateur]
52      *        (https://fr.wikipedia.org/wiki/Num%C3%A9rateur)
53      *        obtenu après [réduction de la fraction]
54      * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
55      *
56      * Dans la suite, on appelle ce numérateur _numérateur réduit_
57      * ou simplement _numérateur_.
58      */
59     const unsigned numerator_;
60
61     /*!
62      * \brief [Valeur absolue]
63      *        (https://fr.wikipedia.org/wiki/Valeur_absolue) du
64      *        [dénominateur]
```

```
65        *         (https://fr.wikipedia.org/wiki/D%C3%A9nominateur)
66        *         obtenu après [réduction de la fraction]
67        * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
68        *
69        * Dans la suite, on appelle ce dénominateur
70        * _dénominateur réduit_ ou simplement _dénominateur_.
71        */
72       const unsigned denominator_;
73
74     public:
75
76       /*!
77        * \brief Constructeur avec déduction du signe.
78        *
79        * Les arguments du constructeur sont ici dénommés numérateur et
80        * dénominateur _bruts_.
81        *
82        * Les attributs \ref numerator_ et \ref denominator_
83        * sont obtenus après [réduction]
84        * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
85        * de la fraction brute. L'attribut \ref sign_ est également
86        * calculé.
87        *
88        * Si le dénominateur est nul, une exception
89        * std::invalid_argument est levée.
90        *
91        * Par convention, la fraction correspondant à la valeur zéro,
92        * c'est-à-dire de numérateur nul et dénominateur non nul, est
93        * représentée avec :
94        *   + \ref sign_ égal à Sign::ZERO ;
95        *   + \ref numerator_ égal à 0 ;
96        *   + \ref denominator_ égal à 1.
97        *
98        * \param numerator numérateur brut.
99        * \param denominator dénominateur brut.
100       *
101       * \throw std::invalid_argument si `denominator` est nul.
102       */
103      Fraction(int numerator = 0, int denominator = 1);
104
105      /*!
106       * \brief Constructeur avec signe explicite.
107       *
108       * Les arguments de construction sont le signe et les valeurs
```

```
109       * absolues des numérateur et dénominateur _bruts_.
110       *
111       * Les attributs \ref numerator_ et \ref denominator_
112       * sont obtenus après [réduction]
113       * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
114       * de la fraction brute.
115       *
116       * Si le dénominateur est nul, une exception
117       * std::invalid_argument est levée.
118       *
119       * Une exception std::invalid_argument est également levée si
120       * le signe fourni est Sign::ZERO alors que la fraction,
121       * plus précisément le numérateur, n'est pas nulle. Par
122       * contre, si le numérateur fourni est nul, l'attribut
123       * \ref sign_ est mis à Sign::ZERO, quelle que soit la valeur
124       * du paramètre `sign`.
125       *
126       * Par convention, la fraction correspondant à la valeur zéro,
127       * c'est-à-dire de numérateur nul et dénominateur non nul, est
128       * représentée avec :
129       *   + \ref sign_ égal à Sign::ZERO ;
130       *   + \ref numerator_ égal à 0 ;
131       *   + \ref denominator_ égal à 1.
132       *
133       * \param sign signe de la fraction à construire.
134       * \param numerator valeur absolue du numérateur brut.
135       * \param denominator valeur absolue du dénominateur brut.
136       *
137       * \throw std::invalid_argument si
138       *      + `denominator` est nul ;
139       *      + `sign` est Sign::ZERO alors que `numerator` n'est pas
140       *        nul.
141       */
142      Fraction(Sign sign, unsigned numerator = 0,
143               unsigned denominator = 1);
144
145      /*!
146       * \brief Accesseur en lecture du signe.
147       *
148       * \return le signe de la fraction.
149       */
150      inline Sign sign() const;
151
152      /*!
```

```
153       * \brief Accesseur en lecture du numérateur.
154       *
155       * Le numérateur retourné est toujours le numérateur réduit.
156       *
157       * \return le numérateur réduit de la fraction.
158       */
159      inline unsigned numerator() const;
160
161      /*!
162       * \brief Accesseur en lecture du dénominateur.
163       *
164       * Le dénominateur retourné est toujours le dénominateur réduit.
165       *
166       * \return le dénominateur réduit de la fraction.
167       */
168      inline unsigned denominator() const;
169
170      /*!
171       * \brief Accesseur en lecture sous la forme _unité_ +
172       *        _partie fractionnaire_.
173       *
174       * Le premier champ du std::tuple retourné est le signe de
175       * la fraction.
176       *
177       * Par exemple :
178       *    + pour la fraction 5 / 6, ce premier champ vaut
179       *      Sign::PLUS car 5 / 6 > 0 ;
180       *    + pour la fraction 15 / 6, ce premier champ vaut
181       *      Sign::PLUS car 15 / 6 > 0 ;
182       *    + pour la fraction 0 / 6, ce premier champ vaut
183       *      Sign::ZERO car 0 / 6 = 0 ;
184       *    + pour la fraction - 18 / 6, ce premier champ vaut
185       *      Sign::MINUS car - 18 / 6 < 0 ;
186       *    + pour la fraction - 23 / 6, ce premier champ vaut
187       *      Sign::MINUS car - 23 / 6 < 0.
188       *
189       * Le deuxième champ du std::tuple retourné est le nombre
190       * d'unités de la valeur absolue de la fraction. Plus
191       * précisément, c'est le plus grand naturel
192       * inférieur ou égal à la valeur absolue de la fraction.
193       *
194       * Par exemple :
195       *    + pour la fraction 5 / 6, ce deuxième champ vaut 0
196       *      car 0 <= 5 / 6 < 1 ;
```

```
197     *    + pour la fraction 15 / 6, ce deuxième champ vaut 2
198     *      car 2 <= 15 / 6 < 3 ;
199     *    + pour la fraction 0 / 6, ce deuxième champ vaut 0
200     *      car 0 <= 0 / 6 < 1 ;
201     *    + pour la fraction - 18 / 6, ce deuxième champ vaut 3
202     *      car 3 <= 18 / 6 < 4 ;
203     *    + pour la fraction - 23 / 6, ce deuxième champ vaut 3
204     *      car 3 <= 23 / 6 < 4.
205     *
206     * Le troisième champ du std::tuple retourné est le numérateur
207     * de la fraction réduite obtenue lorsqu'on soustrait le nombre
208     * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
209     * de la fraction de départ.
210     *
211     * Par exemple :
212     *    + pour la fraction 5 / 6, ce troisième champ vaut 5 car
213     *      5 / 6 - 0 = 5 / 6 ;
214     *    + pour la fraction 15 / 6, ce troisième champ vaut 1 car
215     *      15 / 6 - 2 = 1 / 2 ;
216     *    + pour la fraction 0 / 6, ce troisième champ vaut 0 car
217     *      0 / 6 - 0 = 0 ;
218     *    + pour la fraction - 18 / 6, ce troisième champ vaut 0 car
219     *      18 / 6 - 3 = 0 ;
220     *    + pour la fraction - 23 / 6, ce troisième champ vaut 5 car
221     *      23 / 6 - 3 = 5 / 6.
222     *
223     * Le quatrième champ du std::tuple retourné est le dénominateur
224     * de la fraction réduite obtenue lorsqu'on soustrait le nombre
225     * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
226     * de la fraction de départ.
227     *
228     * Par exemple :
229     *    + pour la fraction 5 / 6, ce quatrième champ vaut 6 car
230     *      5 / 6 - 0 = 5 / 6 ;
231     *    + pour la fraction 15 / 6, ce quatrième champ vaut 2 car
232     *      15 / 6 - 2 = 1 / 2 ;
233     *    + pour la fraction 0 / 6, ce quatrième champ vaut 1 car
234     *      0 / 6 - 0 = 0 / 1 ;
235     *    + pour la fraction - 18 / 6, ce quatrième champ vaut 1 car
236     *      18 / 6 - 3 = 0 / 1 ;
237     *    + pour la fraction - 23 / 6, ce quatrième champ vaut 6 car
238     *      23 / 6 - 3 = 5 / 6.
239     *
240     * En résumé et par exemple :
```

```
241      *    + pour la fraction 5 / 6, le std::tuple retourné vaut
242      *      <Sign::PLUS, 0, 5, 6> ;
243      *    + pour la fraction 15 / 6, le std::tuple retourné vaut
244      *      <Sign::PLUS, 2, 1, 2> ;
245      *    + pour la fraction 0 / 6, le std::tuple retourné vaut
246      *      <Sign::ZERO, 0, 0, 1> ;
247      *    + pour la fraction - 18 / 6, le std::tuple retourné vaut
248      *      <Sign::MINUS, 3, 0, 1> ;
249      *    + pour la fraction - 23 / 6, le std::tuple retourné vaut
250      *      <Sign::MINUS, 3, 5, 6>.
251      *
252      * \return un std::tuple représentant la fraction sous la
253      *         forme <signe, unités, numérateur, dénominateur>
254      *         tel que décrit ci-dessus.
255      *
256      * \see unit_form(const Fraction &).
257      */
258     inline std::tuple<Sign, unsigned, unsigned, unsigned>
259     unit_form() const;
260
261     /*!
262      * \brief Conversion d'une Fraction en std::string.
263      *
264      * Si le dénominateur vaut 1, il n'est pas utilisé. Par exemple,
265      * la fraction « - 2 / 1 » est représentée par la std::string
266      * « - 2 ».
267      *
268      * \return une std::string représentant la fraction.
269      *
270      * \see to_string(const Fraction &).
271      */
272     std::string to_string() const;
273
274     /*!
275      * \brief Conversion d'une Fraction en double.
276      */
277     inline explicit operator double() const;
278 };
279
280 // prototypes
281
282 /*!
283  * \brief Calcul de la [valeur absolue]
284  *        (https://fr.wikipedia.org/wiki/Valeur_absolue)
```

```
285  *        d'une fraction.
286  *
287  * \param fraction fraction dont on désire connaitre la valeur
288  *        absolue.
289  *
290  * \return la valeur absolue de `fraction`.
291  */
292 inline Fraction abs(const Fraction & fraction);
293
294 /*!
295  * \brief Expression d'une fraction sous la forme _unité_ +
296  *        _partie fractionnaire_.
297  *
298  * Le premier champ du std::tuple retourné est le signe de
299  * la fraction.
300  *
301  * Par exemple :
302  *    + pour la fraction 5 / 6, ce premier champ vaut
303  *      Sign::PLUS car 5 / 6 > 0 ;
304  *    + pour la fraction 15 / 6, ce premier champ vaut
305  *      Sign::PLUS car 15 / 6 > 0 ;
306  *    + pour la fraction 0 / 6, ce premier champ vaut
307  *      Sign::ZERO car 0 / 6 = 0 ;
308  *    + pour la fraction - 18 / 6, ce premier champ vaut
309  *      Sign::MINUS car - 18 / 6 < 0 ;
310  *    + pour la fraction - 23 / 6, ce premier champ vaut
311  *      Sign::MINUS car - 23 / 6 < 0.
312  *
313  * Le deuxième champ du std::tuple retourné est le nombre
314  * d'unités de la valeur absolue de la fraction. Plus
315  * précisément, c'est le plus grand naturel
316  * inférieur ou égal à la valeur absolue de la fraction.
317  *
318  * Par exemple :
319  *    + pour la fraction 5 / 6, ce deuxième champ vaut 0
320  *      car 0 <= 5 / 6 > 1 ;
321  *    + pour la fraction 15 / 6, ce deuxième champ vaut 2
322  *      car 2 <= 15 / 6 < 3 ;
323  *    + pour la fraction 0 / 6, ce deuxième champ vaut 0
324  *      car 0 <= 0 / 6 < 1 ;
325  *    + pour la fraction - 18 / 6, ce deuxième champ vaut 3
326  *      car 3 <= 18 / 6 < 4 ;
327  *    + pour la fraction - 23 / 6, ce deuxième champ vaut 3
328  *      car 3 <= 23 / 6 < 4.
```

```
329  *
330  * Le troisième champ du std::tuple retourné est le numérateur
331  * de la fraction réduite obtenue lorsqu'on soustrait le nombre
332  * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
333  * de la fraction de départ.
334  *
335  * Par exemple :
336  *    + pour la fraction 5 / 6, ce troisième champ vaut 5 car
337  *      5 / 6 - 0 = 5 / 6 ;
338  *    + pour la fraction 15 / 6, ce troisième champ vaut 1 car
339  *      15 / 6 - 2 = 1 / 2 ;
340  *    + pour la fraction 0 / 6, ce troisième champ vaut 0 car
341  *      0 / 6 - 0 = 0 ;
342  *    + pour la fraction - 18 / 6, ce troisième champ vaut 0 car
343  *      18 / 6 - 3 = 0 ;
344  *    + pour la fraction - 23 / 6, ce troisième champ vaut 5 car
345  *      23 / 6 - 3 = 5 / 6.
346  *
347  * Le quatrième champ du std::tuple retourné est le dénominateur
348  * de la fraction réduite obtenue lorsqu'on soustrait le nombre
349  * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
350  * de la fraction de départ.
351  *
352  * Par exemple :
353  *    + pour la fraction 5 / 6, ce quatrième champ vaut 6 car
354  *      5 / 6 - 0 = 5 / 6 ;
355  *    + pour la fraction 15 / 6, ce quatrième champ vaut 2 car
356  *      15 / 6 - 2 = 1 / 2 ;
357  *    + pour la fraction 0 / 6, ce quatrième champ vaut 1 car
358  *      0 / 6 - 0 = 0 / 1 ;
359  *    + pour la fraction - 18 / 6, ce quatrième champ vaut 1 car
360  *      18 / 6 - 3 = 0 / 1 ;
361  *    + pour la fraction - 23 / 6, ce quatrième champ vaut 6 car
362  *      23 / 6 - 3 = 5 / 6.
363  *
364  * En résumé et par exemple :
365  *    + pour la fraction 5 / 6, le std::tuple retourné vaut
366  *      <Sign::PLUS, 0, 5, 6> ;
367  *    + pour la fraction 15 / 6, le std::tuple retourné vaut
368  *      <Sign::PLUS, 2, 1, 2> ;
369  *    + pour la fraction 0 / 6, le std::tuple retourné vaut
370  *      <Sign::ZERO, 0, 0, 1> ;
371  *    + pour la fraction - 18 / 6, le std::tuple retourné vaut
372  *      <Sign::MINUS, 3, 0, 1> ;
```

```
373  *    + pour la fraction - 23 / 6, le std::tuple retourné vaut
374  *      <Sign::MINUS, 3, 5, 6>.
375  *
376  * \param fraction fraction dont on désire la forme _unité_ +
377  *                 _partie fractionnaire_.
378  *
379  * \return un std::tuple représentant la fraction sous la
380  *         forme <signe, unités, numérateur, dénominateur>
381  *         tel que décrit ci-dessus.
382  *
383  * \see Fraction::unit_form().
384  */
385 inline std::tuple<Sign, unsigned, unsigned, unsigned>
386 unit_form(const Fraction & fraction);
387
388 /*!
389  * \brief Conversion d'une Fraction en std::string.
390  *
391  * Si le dénominateur vaut 1, il n'est pas utilisé. Par exemple,
392  * la fraction « - 2 / 1 » est représentée par la std::string
393  * « - 2 ».
394  *
395  * \param fraction fraction dont on désire la conversion en
396  *                 std::string.
397  *
398  * \return une std::string représentant `fraction`.
399  *
400  * \see Fraction::to_string().
401  */
402 inline std::string to_string(const Fraction & fraction);
403
404 /*!
405  * \brief Opérateur d'injection d'une Fraction dans un flux en
406  *        sortie.
407  *
408  * \param out flux dans lequel l'injection est réalisée.
409  * \param fraction fraction à injecter.
410  *
411  * \return le flux après injection.
412  */
413 inline std::ostream & operator<<(std::ostream & out,
414                                  const Fraction & fraction);
415
416 /*!
```

```
417  * \brief Opérateur de comparaison pour tester l'égalité de deux
418  *        fractions.
419  *
420  * \param lhs une fraction.
421  * \param rhs une fraction
422  *
423  * \return `true` si `lhs` et `rhs` sont égales, `false` sinon.
424  */
425 inline bool operator==(const Fraction & lhs, const Fraction & rhs);
426 
427 /*!
428  * \brief Opérateur de comparaison pour tester la différence de deux
429  *        fractions.
430  *
431  * \param lhs une fraction.
432  * \param rhs une fraction
433  *
434  * \return `true` si `lhs` et `rhs` sont différentes, `false` sinon.
435  */
436 inline bool operator!=(const Fraction & lhs, const Fraction & rhs);
437 
438 /*!
439  * \brief Opérateur de comparaison pour tester la relation d'ordre
440  *        strictement inférieur de deux fractions.
441  *
442  * \param lhs une fraction.
443  * \param rhs une fraction
444  *
445  * \return `true` si `lhs` est strictement inférieure à `rhs`,
446  *         `false` sinon.
447  */
448 bool operator<(const Fraction & lhs, const Fraction & rhs);
449 
450 /*!
451  * \brief Opérateur de comparaison pour tester la relation d'ordre
452  *        inférieur ou égal de deux fractions.
453  *
454  * \param lhs une fraction.
455  * \param rhs une fraction
456  *
457  * \return `true` si `lhs` est inférieure ou égale à `rhs`,
458  *         `false` sinon.
459  */
460 inline bool operator<=(const Fraction & lhs, const Fraction & rhs);
```

```
461
462 /*!
463  * \brief Opérateur de comparaison pour tester la relation d'ordre
464  *        strictement supérieur de deux fractions.
465  *
466  * \param lhs une fraction.
467  * \param rhs une fraction
468  *
469  * \return `true` si `lhs` est strictement supérieur à `rhs`,
470  *         `false` sinon.
471  */
472 inline bool operator>(const Fraction & lhs, const Fraction & rhs);
473
474 /*!
475  * \brief Opérateur de comparaison pour tester la relation d'ordre
476  *        supérieur ou égal de deux fractions.
477  *
478  * \param lhs une fraction.
479  * \param rhs une fraction
480  *
481  * \return `true` si `lhs` est supérieur ou égal à `rhs`,
482  *         `false` sinon.
483  */
484 inline bool operator>=(const Fraction & lhs, const Fraction & rhs);
485
486 /*!
487  * \brief Opérateur unaire identique.
488  *
489  * \param fraction une fraction.
490  *
491  * \return `fraction`.
492  */
493 inline Fraction operator+(const Fraction & fraction);
494
495 /*!
496  * \brief Opérateur unaire opposé.
497  *
498  * \param fraction une fraction.
499  *
500  * \return l'opposé de `fraction`.
501  */
502 inline Fraction operator-(const Fraction & fraction);
503
504 /*!
```

```
505  * \brief Opérateur arithmétique d'[addition]
506  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Addition)
507  *        de deux fractions.
508  *
509  * \param lhs une fraction.
510  * \param rhs une fraction.
511  *
512  * \return la fraction égale à la somme de `lhs` et `rhs`.
513  */
514 Fraction operator+(const Fraction & lhs, const Fraction & rhs);
515 
516 /*!
517  * \brief Opérateur arithmétique de [soustraction]
518  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Subtraction)
519  *        de deux fractions.
520  *
521  * \param lhs une fraction.
522  * \param rhs une fraction.
523  *
524  * \return la fraction égale à la soustraction de `rhs` à `lhs`.
525  */
526 inline Fraction operator-(const Fraction & lhs, const Fraction & rhs);
527 
528 /*!
529  * \brief Opérateur arithmétique de [multiplication]
530  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Multiplication)
531  *        de deux fractions.
532  *
533  * \param lhs une fraction.
534  * \param rhs une fraction.
535  *
536  * \return la fraction égale au produit de `lhs` par `rhs`.
537  */
538 Fraction operator*(const Fraction & lhs, const Fraction & rhs);
539 
540 /*!
541  * \brief Opérateur arithmétique de [division]
542  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Division)
543  *        de deux fractions.
544  *
545  * \param lhs une fraction.
546  * \param rhs une fraction.
547  *
548  * \return la fraction égale à la division de `lhs` par `rhs`.
```

```
549  *
550  * \throw std::invalid_argument si `rhs` est nul.
551  */
552 Fraction operator/(const Fraction & lhs, const Fraction & rhs);
553 
554 // implémentation de méthodes inline
555 
556 inline Sign Fraction::sign() const
557 {
558     // TODO
559     return { };
560 }
561 
562 inline unsigned Fraction::numerator() const
563 {
564     // TODO
565     return { };
566 }
567 
568 inline unsigned Fraction::denominator() const
569 {
570     // TODO
571     return { };
572 }
573 
574 inline std::tuple<Sign, unsigned, unsigned, unsigned>
575 Fraction::unit_form() const
576 {
577     // TODO
578     return { };
579 }
580 
581 inline Fraction::operator double() const
582 {
583     // TODO
584     return { };
585 }
586 
587 // implémentation de fonctions inline
588 
589 inline Fraction abs(const Fraction & fraction)
590 {
591     // TODO
592     return { };
```

```
593 }
594
595 inline std::tuple<Sign, unsigned, unsigned, unsigned>
596 unit_form(const Fraction & fraction)
597 {
598     // TODO
599     return { };
600 }
601
602 inline std::string to_string(const Fraction & fraction)
603 {
604     // TODO
605     return { };
606 }
607
608 inline std::ostream & operator<<(std::ostream & out,
609                                  const Fraction & fraction)
610 {
611     // TODO
612     return out;
613 }
614
615 inline bool operator==(const Fraction & lhs, const Fraction & rhs)
616 {
617     // TODO
618     return { };
619 }
620
621 inline bool operator!=(const Fraction & lhs, const Fraction & rhs)
622 {
623     // TODO
624     return { };
625 }
626
627 inline bool operator<=(const Fraction & lhs, const Fraction & rhs)
628 {
629     // TODO
630     return { };
631 }
632
633 inline bool operator>(const Fraction & lhs, const Fraction & rhs)
634 {
635     // TODO
636     return { };
```

```
637 }
638
639 inline bool operator>=(const Fraction & lhs, const Fraction & rhs)
640 {
641     // TODO
642     return { };
643 }
644
645 inline Fraction operator+(const Fraction & fraction)
646 {
647     // TODO
648     return { };
649 }
650
651 inline Fraction operator-(const Fraction & fraction)
652 {
653     // TODO
654     return { };
655 }
656
657 inline Fraction operator-(const Fraction & lhs, const Fraction & rhs)
658 {
659     // TODO
660     return { };
661 }
662
663 } // namespace nvs
664
665 #endif // FRACTION_INCOMPLETE_H
```

B.2. Fichier source

```
1  // TODO : include manquants
2
3  #include "fraction_incomplete.h"
4
5  namespace nvs
6  {
7
8  // méthodes
9
10 Fraction::Fraction(int numerator, int denominator) :
11     // TODO
```

```
12      sign_ { },
13      numerator_ { },
14      denominator_ { }
15  { }
16
17  Fraction::Fraction(Sign sign, unsigned numerator,
18                     unsigned denominator) :
19      // TODO
20      sign_ { },
21      numerator_ { },
22      denominator_ { }
23  { }
24
25  std::string Fraction::to_string() const
26  {
27      // TODO
28      return { };
29  }
30
31  // fonctions
32
33  bool operator<(const Fraction & lhs,
34                 const Fraction & rhs)
35  {
36      // TODO
37      return { };
38  }
39
40  Fraction operator+(const Fraction & lhs,
41                     const Fraction & rhs)
42  {
43      // TODO
44      return { };
45  }
46
47  Fraction operator*(const Fraction & lhs,
48                     const Fraction & rhs)
49  {
50      // TODO
51      return { };
52  }
53
54  Fraction operator/(const Fraction & lhs, const Fraction & rhs)
55  {
```

```
56      // TODO
57      return { };
58  }
59
60  } // namespave nvs
```

C. Classe `Fraction constexpr`

```
1   /*!
2    * \file fraction_constexpr_incomplete.h
3    *
4    * \brief Définition de la classe nvs::Fraction ainsi que d'un
5    *        type (nvs::Fraction::Form) et de fonctions associées.
6    */
7   #ifndef FRACTION_CONSTEXPR_INCOMPLETE_H
8   #define FRACTION_CONSTEXPR_INCOMPLETE_H
9
10  #include <string>
11  #include <ostream>
12  #include <tuple>
13
14  // TODO include manquants
15
16  #include "sign_incomplete.h"
17
18  /*!
19   * \brief Espace de nom de Nicolas Vansteenkiste.
20   */
21  namespace nvs
22  {
23
24  /*!
25   * \brief Classe représentant une [fraction]
26   * (https://fr.wikipedia.org/wiki/Fraction_(math%C3%A9matiques)).
27   *
28   * Comme caractéristique principale, indiquons la sauvegarde du
29   * signe sous la forme d'un nvs::Sign tandis que les numérateur
30   * et dénominateur sont stockés sous une forme non signée.
31   *
32   * Par ailleurs, toute fraction est toujours stockée sous forme
33   * simplifiée, c'est-à-dire comme une [fraction irréductible]
34   * (https://fr.wikipedia.org/wiki/Fraction_irr%C3%A9ductible).
```

```
35  *
36  * Outre les méthodes de la classe Fraction, il existe de
37  * multiples fonctions pour les utiliser. Elles sont définies
38  * dans l'espace de nom \ref nvs dans le fichier fraction_constexpr.h.
39  *
40  */
41 class Fraction
42 {
43     /*!
44      * \brief [Signe]
45      *        (https://fr.wikipedia.org/wiki/Signe_(arithm%C3%A9tique))
46      *        de la fraction.
47      */
48     const Sign sign_;
49
50     /*!
51      * \brief [Valeur absolue]
52      *        (https://fr.wikipedia.org/wiki/Valeur_absolue) du
53      *        [numérateur]
54      *        (https://fr.wikipedia.org/wiki/Num%C3%A9rateur)
55      *        obtenu après [réduction de la fraction]
56      * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
57      *
58      * Dans la suite, on appelle ce numérateur _numérateur réduit_
59      * ou simplement _numérateur_.
60      */
61     const unsigned numerator_;
62
63     /*!
64      * \brief [Valeur absolue]
65      *        (https://fr.wikipedia.org/wiki/Valeur_absolue) du
66      *        [dénominateur]
67      *        (https://fr.wikipedia.org/wiki/D%C3%A9nominateur)
68      *        obtenu après [réduction de la fraction]
69      * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
70      *
71      * Dans la suite, on appelle ce dénominateur
72      * _dénominateur réduit_ ou simplement _dénominateur_.
73      */
74     const unsigned denominator_;
75
76   public:
77
78     /*!
```

```
79        * \brief Constructeur avec déduction du signe.
80        *
81        * Les arguments du constructeur sont ici dénommés numérateur et
82        * dénominateur _bruts_.
83        *
84        * Les attributs \ref numerator_ et \ref denominator_
85        * sont obtenus après [réduction]
86        * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
87        * de la fraction brute. L'attribut \ref sign_ est également
88        * calculé.
89        *
90        * Si le dénominateur est nul, une exception
91        * std::invalid_argument est levée.
92        *
93        * Par convention, la fraction correspondant à la valeur zéro,
94        * c'est-à-dire de numérateur nul et dénominateur non nul, est
95        * représentée avec :
96        *   + \ref sign_ égal à Sign::ZERO ;
97        *   + \ref numerator_ égal à 0 ;
98        *   + \ref denominator_ égal à 1.
99        *
100       * \param numerator numérateur brut.
101       * \param denominator dénominateur brut.
102       *
103       * \throw std::invalid_argument si `denominator` est nul.
104       */
105      inline constexpr Fraction(int numerator = 0, int denominator = 1);
106
107      /*!
108       * \brief Constructeur avec signe explicite.
109       *
110       * Les arguments de construction sont le signe et les valeurs
111       * absolues des numérateur et dénominateur _bruts_.
112       *
113       * Les attributs \ref numerator_ et \ref denominator_
114       * sont obtenus après [réduction]
115       * (https://fr.wikipedia.org/wiki/Plus_grand_commun_diviseur_de_nombres_entiers
116       * de la fraction brute.
117       *
118       * Si le dénominateur est nul, une exception
119       * std::invalid_argument est levée.
120       *
121       * Une exception std::invalid_argument est également levée si
122       * le signe fourni est Sign::ZERO alors que la fraction,
```

```
123       * plus précisément le numérateur, n'est pas nulle. Par
124       * contre, si le numérateur fourni est nul, l'attribut
125       * \ref sign_ est mis à Sign::ZERO, quelle que soit la valeur
126       * du paramètre `sign`.
127       *
128       * Par convention, la fraction correspondant à la valeur zéro,
129       * c'est-à-dire de numérateur nul et dénominateur non nul, est
130       * représentée avec :
131       *   + \ref sign_ égal à Sign::ZERO ;
132       *   + \ref numerator_ égal à 0 ;
133       *   + \ref denominator_ égal à 1.
134       *
135       * \param sign signe de la fraction à construire.
136       * \param numerator valeur absolue du numérateur brut.
137       * \param denominator valeur absolue du dénominateur brut.
138       *
139       * \throw std::invalid_argument si
140       *      + `denominator` est nul ;
141       *      + `sign` est Sign::ZERO alors que `numerator` n'est pas
142       *        nul.
143       */
144      inline constexpr Fraction(Sign sign, unsigned numerator = 0,
145                                unsigned denominator = 1);
146
147      /*!
148       * \brief Accesseur en lecture du signe.
149       *
150       * \return le signe de la fraction.
151       */
152      inline constexpr Sign sign() const;
153
154      /*!
155       * \brief Accesseur en lecture du numérateur.
156       *
157       * Le numérateur retourné est toujours le numérateur réduit.
158       *
159       * \return le numérateur réduit de la fraction.
160       */
161      inline constexpr unsigned numerator() const;
162
163      /*!
164       * \brief Accesseur en lecture du dénominateur.
165       *
166       * Le dénominateur retourné est toujours le dénominateur réduit.
```

```
167      *
168      * \return le dénominateur réduit de la fraction.
169      */
170     inline constexpr unsigned denominator() const;
171
172     /*!
173      * \brief Accesseur en lecture sous la forme _unité_ +
174      *        _partie fractionnaire_.
175      *
176      * Le premier champ du std::tuple retourné est le signe de
177      * la fraction.
178      *
179      * Par exemple :
180      *    + pour la fraction 5 / 6, ce premier champ vaut
181      *      Sign::PLUS car 5 / 6 > 0 ;
182      *    + pour la fraction 15 / 6, ce premier champ vaut
183      *      Sign::PLUS car 15 / 6 > 0 ;
184      *    + pour la fraction 0 / 6, ce premier champ vaut
185      *      Sign::ZERO car 0 / 6 = 0 ;
186      *    + pour la fraction - 18 / 6, ce premier champ vaut
187      *      Sign::MINUS car - 18 / 6 < 0 ;
188      *    + pour la fraction - 23 / 6, ce premier champ vaut
189      *      Sign::MINUS car - 23 / 6 < 0.
190      *
191      * Le deuxième champ du std::tuple retourné est le nombre
192      * d'unités de la valeur absolue de la fraction. Plus
193      * précisément, c'est le plus grand naturel
194      * inférieur ou égal à la valeur absolue de la fraction.
195      *
196      * Par exemple :
197      *    + pour la fraction 5 / 6, ce deuxième champ vaut 0
198      *      car 0 <= 5 / 6 < 1 ;
199      *    + pour la fraction 15 / 6, ce deuxième champ vaut 2
200      *      car 2 <= 15 / 6 < 3 ;
201      *    + pour la fraction 0 / 6, ce deuxième champ vaut 0
202      *      car 0 <= 0 / 6 < 1 ;
203      *    + pour la fraction - 18 / 6, ce deuxième champ vaut 3
204      *      car 3 <= 18 / 6 < 4 ;
205      *    + pour la fraction - 23 / 6, ce deuxième champ vaut 3
206      *      car 3 <= 23 / 6 < 4.
207      *
208      * Le troisième champ du std::tuple retourné est le numérateur
209      * de la fraction réduite obtenue lorsqu'on soustrait le nombre
210      * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
```

```
211     * de la fraction de départ.
212     *
213     * Par exemple :
214     *    + pour la fraction 5 / 6, ce troisième champ vaut 5 car
215     *      5 / 6 - 0 = 5 / 6 ;
216     *    + pour la fraction 15 / 6, ce troisième champ vaut 1 car
217     *      15 / 6 - 2 = 1 / 2 ;
218     *    + pour la fraction 0 / 6, ce troisième champ vaut 0 car
219     *      0 / 6 - 0 = 0 ;
220     *    + pour la fraction - 18 / 6, ce troisième champ vaut 0 car
221     *      18 / 6 - 3 = 0 ;
222     *    + pour la fraction - 23 / 6, ce troisième champ vaut 5 car
223     *      23 / 6 - 3 = 5 / 6.
224     *
225     * Le quatrième champ du std::tuple retourné est le dénominateur
226     * de la fraction réduite obtenue lorsqu'on soustrait le nombre
227     * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
228     * de la fraction de départ.
229     *
230     * Par exemple :
231     *    + pour la fraction 5 / 6, ce quatrième champ vaut 6 car
232     *      5 / 6 - 0 = 5 / 6 ;
233     *    + pour la fraction 15 / 6, ce quatrième champ vaut 2 car
234     *      15 / 6 - 2 = 1 / 2 ;
235     *    + pour la fraction 0 / 6, ce quatrième champ vaut 1 car
236     *      0 / 6 - 0 = 0 / 1 ;
237     *    + pour la fraction - 18 / 6, ce quatrième champ vaut 1 car
238     *      18 / 6 - 3 = 0 / 1 ;
239     *    + pour la fraction - 23 / 6, ce quatrième champ vaut 6 car
240     *      23 / 6 - 3 = 5 / 6.
241     *
242     * En résumé et par exemple :
243     *    + pour la fraction 5 / 6, le std::tuple retourné vaut
244     *      <Sign::PLUS, 0, 5, 6> ;
245     *    + pour la fraction 15 / 6, le std::tuple retourné vaut
246     *      <Sign::PLUS, 2, 1, 2> ;
247     *    + pour la fraction 0 / 6, le std::tuple retourné vaut
248     *      <Sign::ZERO, 0, 0, 1> ;
249     *    + pour la fraction - 18 / 6, le std::tuple retourné vaut
250     *      <Sign::MINUS, 3, 0, 1> ;
251     *    + pour la fraction - 23 / 6, le std::tuple retourné vaut
252     *      <Sign::MINUS, 3, 5, 6>.
253     *
254     * \return un std::tuple représentant la fraction sous la
```

```
255         *          forme <signe, unités, numérateur, dénominateur>
256         *          tel que décrit ci-dessus.
257         *
258         * \see unit_form(const Fraction &).
259         */
260        inline constexpr std::tuple<Sign, unsigned, unsigned, unsigned>
261        unit_form() const;
262
263        /*!
264         * \brief Conversion d'une Fraction en std::string.
265         *
266         * Si le dénominateur vaut 1, il n'est pas utilisé. Par exemple,
267         * la fraction « - 2 / 1 » est représentée par la std::string
268         * « - 2 ».
269         *
270         * \return une std::string représentant la fraction.
271         *
272         * \see to_string(const Fraction &).
273         */
274        inline std::string to_string() const;
275
276        /*!
277         * \brief Conversion d'une Fraction en double.
278         */
279        inline constexpr explicit operator double() const;
280    };
281
282    // prototypes
283
284    /*!
285     * \brief Calcul de la [valeur absolue]
286     *        (https://fr.wikipedia.org/wiki/Valeur_absolue)
287     *        d'une fraction.
288     *
289     * \param fraction fraction dont on désire connaitre la valeur
290     *        absolue.
291     *
292     * \return la valeur absolue de `fraction`.
293     */
294    constexpr Fraction abs(const Fraction & fraction);
295
296    /*!
297     * \brief Expression d'une fraction sous la forme _unité_ +
298     *        _partie fractionnaire_.
```

```
299  *
300  * Le premier champ du std::tuple retourné est le signe de
301  * la fraction.
302  *
303  * Par exemple :
304  *    + pour la fraction 5 / 6, ce premier champ vaut
305  *      Sign::PLUS car 5 / 6 > 0 ;
306  *    + pour la fraction 15 / 6, ce premier champ vaut
307  *      Sign::PLUS car 15 / 6 > 0 ;
308  *    + pour la fraction 0 / 6, ce premier champ vaut
309  *      Sign::ZERO car 0 / 6 = 0 ;
310  *    + pour la fraction - 18 / 6, ce premier champ vaut
311  *      Sign::MINUS car - 18 / 6 < 0 ;
312  *    + pour la fraction - 23 / 6, ce premier champ vaut
313  *      Sign::MINUS car - 23 / 6 < 0.
314  *
315  * Le deuxième champ du std::tuple retourné est le nombre
316  * d'unités de la valeur absolue de la fraction. Plus
317  * précisément, c'est le plus grand naturel
318  * inférieur ou égal à la valeur absolue de la fraction.
319  *
320  * Par exemple :
321  *    + pour la fraction 5 / 6, ce deuxième champ vaut 0
322  *      car 0 <= 5 / 6 > 1 ;
323  *    + pour la fraction 15 / 6, ce deuxième champ vaut 2
324  *      car 2 <= 15 / 6 < 3 ;
325  *    + pour la fraction 0 / 6, ce deuxième champ vaut 0
326  *      car 0 <= 0 / 6 < 1 ;
327  *    + pour la fraction - 18 / 6, ce deuxième champ vaut 3
328  *      car 3 <= 18 / 6 < 4 ;
329  *    + pour la fraction - 23 / 6, ce deuxième champ vaut 3
330  *      car 3 <= 23 / 6 < 4.
331  *
332  * Le troisième champ du std::tuple retourné est le numérateur
333  * de la fraction réduite obtenue lorsqu'on soustrait le nombre
334  * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
335  * de la fraction de départ.
336  *
337  * Par exemple :
338  *    + pour la fraction 5 / 6, ce troisième champ vaut 5 car
339  *      5 / 6 - 0 = 5 / 6 ;
340  *    + pour la fraction 15 / 6, ce troisième champ vaut 1 car
341  *      15 / 6 - 2 = 1 / 2 ;
342  *    + pour la fraction 0 / 6, ce troisième champ vaut 0 car
```

```
343 *      0 / 6 - 0 = 0 ;
344 *    + pour la fraction - 18 / 6, ce troisième champ vaut 0 car
345 *      18 / 6 - 3 = 0 ;
346 *    + pour la fraction - 23 / 6, ce troisième champ vaut 5 car
347 *      23 / 6 - 3 = 5 / 6.
348 *
349 * Le quatrième champ du std::tuple retourné est le dénominateur
350 * de la fraction réduite obtenue lorsqu'on soustrait le nombre
351 * d'unités, c'est-à-dire le deuxième champ, de la valeur absolue
352 * de la fraction de départ.
353 *
354 * Par exemple :
355 *    + pour la fraction 5 / 6, ce quatrième champ vaut 6 car
356 *      5 / 6 - 0 = 5 / 6 ;
357 *    + pour la fraction 15 / 6, ce quatrième champ vaut 2 car
358 *      15 / 6 - 2 = 1 / 2 ;
359 *    + pour la fraction 0 / 6, ce quatrième champ vaut 1 car
360 *      0 / 6 - 0 = 0 / 1 ;
361 *    + pour la fraction - 18 / 6, ce quatrième champ vaut 1 car
362 *      18 / 6 - 3 = 0 / 1 ;
363 *    + pour la fraction - 23 / 6, ce quatrième champ vaut 6 car
364 *      23 / 6 - 3 = 5 / 6.
365 *
366 * En résumé et par exemple :
367 *    + pour la fraction 5 / 6, le std::tuple retourné vaut
368 *      <Sign::PLUS, 0, 5, 6> ;
369 *    + pour la fraction 15 / 6, le std::tuple retourné vaut
370 *      <Sign::PLUS, 2, 1, 2> ;
371 *    + pour la fraction 0 / 6, le std::tuple retourné vaut
372 *      <Sign::ZERO, 0, 0, 1> ;
373 *    + pour la fraction - 18 / 6, le std::tuple retourné vaut
374 *      <Sign::MINUS, 3, 0, 1> ;
375 *    + pour la fraction - 23 / 6, le std::tuple retourné vaut
376 *      <Sign::MINUS, 3, 5, 6>.
377 *
378 * \param fraction fraction dont on désire la forme _unité_ +
379 *                 _partie fractionnaire_.
380 *
381 * \return un std::tuple représentant la fraction sous la
382 *         forme <signe, unités, numérateur, dénominateur>
383 *         tel que décrit ci-dessus.
384 *
385 * \see Fraction::unit_form().
386 */
```

```
387 constexpr std::tuple<Sign, unsigned, unsigned, unsigned>
388 unit_form(const Fraction & fraction);
389
390 /*!
391  * \brief Conversion d'une Fraction en std::string.
392  *
393  * Si le dénominateur vaut 1, il n'est pas utilisé. Par exemple,
394  * la fraction « - 2 / 1 » est représentée par la std::string
395  * « - 2 ».
396  *
397  * \param fraction fraction dont on désire la conversion en
398  *                 std::string.
399  *
400  * \return une std::string représentant `fraction`.
401  *
402  * \see Fraction::to_string().
403  */
404 inline std::string to_string(const Fraction & fraction);
405
406 /*!
407  * \brief Opérateur d'injection d'une Fraction dans un flux en
408  *        sortie.
409  *
410  * \param out flux dans lequel l'injection est réalisée.
411  * \param fraction fraction à injecter.
412  *
413  * \return le flux après injection.
414  */
415 inline std::ostream & operator<<(std::ostream & out,
416                                  const Fraction & fraction);
417
418 /*!
419  * \brief Opérateur de comparaison pour tester l'égalité de deux
420  *        fractions.
421  *
422  * \param lhs une fraction.
423  * \param rhs une fraction
424  *
425  * \return `true` si `lhs` et `rhs` sont égales, `false` sinon.
426  */
427 inline constexpr bool operator==(const Fraction & lhs,
428                                  const Fraction & rhs);
429
430 /*!
```

```
431  * \brief Opérateur de comparaison pour tester la différence de deux
432  *        fractions.
433  *
434  * \param lhs une fraction.
435  * \param rhs une fraction
436  *
437  * \return `true` si `lhs` et `rhs` sont différentes, `false` sinon.
438  */
439 inline constexpr bool operator!=(const Fraction & lhs,
440                                  const Fraction & rhs);
441
442 /*!
443  * \brief Opérateur de comparaison pour tester la relation d'ordre
444  *        strictement inférieur de deux fractions.
445  *
446  * \param lhs une fraction.
447  * \param rhs une fraction
448  *
449  * \return `true` si `lhs` est strictement inférieure à `rhs`,
450  *         `false` sinon.
451  */
452 inline constexpr bool operator<(const Fraction & lhs,
453                                 const Fraction & rhs);
454
455 /*!
456  * \brief Opérateur de comparaison pour tester la relation d'ordre
457  *        inférieur ou égal de deux fractions.
458  *
459  * \param lhs une fraction.
460  * \param rhs une fraction
461  *
462  * \return `true` si `lhs` est inférieure ou égale à `rhs`,
463  *         `false` sinon.
464  */
465 inline constexpr bool operator<=(const Fraction & lhs,
466                                  const Fraction & rhs);
467
468 /*!
469  * \brief Opérateur de comparaison pour tester la relation d'ordre
470  *        strictement supérieur de deux fractions.
471  *
472  * \param lhs une fraction.
473  * \param rhs une fraction
474  *
```

```
475  * \return `true` si `lhs` est strictement supérieur à `rhs`,
476  *         `false` sinon.
477  */
478 inline constexpr bool operator>(const Fraction & lhs,
479                                 const Fraction & rhs);
480 
481 /*!
482  * \brief Opérateur de comparaison pour tester la relation d'ordre
483  *        supérieur ou égal de deux fractions.
484  *
485  * \param lhs une fraction.
486  * \param rhs une fraction
487  *
488  * \return `true` si `lhs` est supérieur ou égal à `rhs`,
489  *         `false` sinon.
490  */
491 inline constexpr bool operator>=(const Fraction & lhs,
492                                  const Fraction & rhs);
493 
494 /*!
495  * \brief Opérateur unaire identique.
496  *
497  * \param fraction une fraction.
498  *
499  * \return `fraction`.
500  */
501 inline constexpr Fraction operator+(const Fraction & fraction);
502 
503 /*!
504  * \brief Opérateur unaire opposé.
505  *
506  * \param fraction une fraction.
507  *
508  * \return l'opposé de `fraction`.
509  */
510 inline constexpr Fraction operator-(const Fraction & fraction);
511 
512 /*!
513  * \brief Opérateur arithmétique d'[addition]
514  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Addition)
515  *        de deux fractions.
516  *
517  * \param lhs une fraction.
518  * \param rhs une fraction.
```

```
519  *
520  * \return la fraction égale à la somme de `lhs` et `rhs`.
521  */
522 inline constexpr Fraction operator+(const Fraction & lhs,
523                                     const Fraction & rhs);
524
525 /*!
526  * \brief Opérateur arithmétique de [soustraction]
527  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Subtraction)
528  *        de deux fractions.
529  *
530  * \param lhs une fraction.
531  * \param rhs une fraction.
532  *
533  * \return la fraction égale à la soustraction de `rhs` à `lhs`.
534  */
535 inline constexpr Fraction operator-(const Fraction & lhs,
536                                     const Fraction & rhs);
537
538 /*!
539  * \brief Opérateur arithmétique de [multiplication]
540  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Multiplication)
541  *        de deux fractions.
542  *
543  * \param lhs une fraction.
544  * \param rhs une fraction.
545  *
546  * \return la fraction égale au produit de `lhs` par `rhs`.
547  */
548 inline constexpr Fraction operator*(const Fraction & lhs,
549                                     const Fraction & rhs);
550
551 /*!
552  * \brief Opérateur arithmétique de [division]
553  *        (https://en.wikipedia.org/wiki/Fraction_(mathematics)#Division)
554  *        de deux fractions.
555  *
556  * \param lhs une fraction.
557  * \param rhs une fraction.
558  *
559  * \return la fraction égale à la division de `lhs` par `rhs`.
560  *
561  * \throw std::invalid_argument si `rhs` est nul.
562  */
```

```cpp
inline constexpr Fraction operator/(const Fraction & lhs,
                                    const Fraction & rhs);

// implémentation de fonctions inline

constexpr Fraction abs(const Fraction & fraction)
{
    // TODO
    return { };
}

constexpr std::tuple<Sign, unsigned, unsigned, unsigned>
unit_form(const Fraction & fraction)
{
    // TODO
    return { };
}

std::string to_string(const Fraction & fraction)
{
    // TODO
    return { };
}

std::ostream & operator<<(std::ostream & out,
                          const Fraction & fraction)
{
    // TODO
    return out;
}

constexpr bool operator==(const Fraction & lhs,
                          const Fraction & rhs)
{
    // TODO
    return { };
}

constexpr bool operator!=(const Fraction & lhs,
                          const Fraction & rhs)
{
    // TODO
    return { };
}
```

```cpp

constexpr bool operator<(const Fraction & lhs,
                         const Fraction & rhs)
{
    // TODO
    return { };
}

constexpr bool operator<=(const Fraction & lhs,
                          const Fraction & rhs)
{
    // TODO
    return { };
}

constexpr bool operator>(const Fraction & lhs,
                         const Fraction & rhs)
{
    // TODO
    return { };
}

constexpr bool operator>=(const Fraction & lhs,
                          const Fraction & rhs)
{
    // TODO
    return { };
}

inline constexpr Fraction operator+(const Fraction & fraction)
{
    // TODO
    return { };
}

inline constexpr Fraction operator-(const Fraction & fraction)
{
    // TODO
    return { };
}

constexpr Fraction operator+(const Fraction & lhs,
                             const Fraction & rhs)
{
```

```
651     // TODO
652     return { };
653 }
654 
655 constexpr Fraction operator-(const Fraction & lhs,
656                              const Fraction & rhs)
657 {
658     // TODO
659     return { };
660 }
661 
662 constexpr Fraction operator*(const Fraction & lhs,
663                              const Fraction & rhs)
664 {
665     // TODO
666     return { };
667 }
668 
669 constexpr Fraction operator/(const Fraction & lhs,
670                              const Fraction & rhs)
671 {
672     // TODO
673     return { };
674 }
675 
676 // implémentation de méthodes inline
677 
678 constexpr Fraction::Fraction(int numerator, int denominator) :
679     // TODO
680     sign_ { },
681     numerator_ { },
682     denominator_ { }
683 { }
684 
685 constexpr Fraction::Fraction(Sign sign, unsigned numerator,
686                              unsigned denominator) :
687     // TODO
688     sign_ { },
689     numerator_ { },
690     denominator_ { }
691 { }
692 
693 constexpr Sign Fraction::sign() const
694 {
```

```
695     // TODO
696     return { };
697 }
698 
699 constexpr unsigned Fraction::numerator() const
700 {
701     // TODO
702     return { };
703 }
704 
705 constexpr unsigned Fraction::denominator() const
706 {
707     // TODO
708     return { };
709 }
710 
711 constexpr std::tuple<Sign, unsigned, unsigned, unsigned>
712 Fraction::unit_form() const
713 {
714     // TODO
715     return { };
716 }
717 
718 std::string Fraction::to_string() const
719 {
720     // TODO
721     return { };
722 }
723 
724 constexpr Fraction::operator double() const
725 {
726     // TODO
727     return { };
728 }
729 
730 } // namespace nvs
731 
732 #endif // FRACTION_CONSTEXPR_INCOMPLETE_H
```

D. Fonctions de génération de données

D.1. Fichier d'en-têtes

```
1  /*!
2   * \file data_fraction.h
3   *
4   * \brief Fonctions pour la génération de données de création de
5   *        fractions.
6   */
7  #ifndef DATA_FRACTION_H
8  #define DATA_FRACTION_H
9
10 #include <vector>
11 #include <utility>
12 #include <tuple>
13
14 namespace nvs
15 {
16
17 /*!
18  * \brief Énumération fortement typée pour choisir le type de
19  *        comportement aléatoire.
20  */
21 enum class Random
22 {
23     /*!
24      * \brief Constante d'énumération destinée à représenter un
25      *        comportement aléatoire reproductible d'une exécution
26      *        l'autre.
27      */
28     REPRODUCTIBLE,
29
30     /*!
31      * \brief Constante d'énumération destinée à représenter un
32      *        comportement aléatoire _non_ reproductible
33      *        d'une exécution l'autre.
34      */
35     UNIQUE
36 };
37
38 /*!
39  * \brief Fonction pour la génération de fraction avec deux entiers
40  *        signés en argument.
```

```
41  *
42  * Le premier attribut de chaque std::pair sert de numérateur,
43  * le second de dénominateur.
44  *
45  * Il peut y avoir des valeurs de dénominateur nulles.
46  *
47  * \param size le nombre de std::pair à produire.
48  * \param type le type de comportement aléatoire attendu.
49  *
50  * \return std::vector de `size` std::pair.
51  */
52 std::vector<std::pair<int, int>>
53 data_signed(unsigned size = 1'000'000, Random type = Random::UNIQUE);
54
55 /*!
56  * \brief Fonction pour la génération de fraction avec un signe et
57  *        deux entiers non signés en argument.
58  *
59  * Le premier champ de chaque std::tuple sert de signe,
60  * le deuxième de numérateur,
61  * le troisième de dénominateur.
62  *
63  * L'`int` prend ses valeurs parmi { -1, 0, 1 }, conformément aux
64  * valeurs de l'enum class nvs::Sign.
65  *
66  * Il peut y avoir des valeurs de signe nulles alors que le premier
67  * `unsigned` n'est pas nul, ou des valeurs de dénominateur nulles.
68  *
69  * \param size le nombre de std::tuple à produire.
70  * \param type le type de comportement aléatoire attendu.
71  *
72  * \return std::vector de `size` std::tuple.
73  */
74 std::vector<std::tuple<int, unsigned, unsigned>>
75 data_unsigned(unsigned size = 1'000'000, Random type = Random::UNIQUE);
76
77 }
78
79 #endif // DATA_FRACTION_H
```

D.2. Fichier source

```cpp
#include <vector>
#include <utility>
#include <tuple>
#include <cmath>

#include "data_fraction.h"
#include "random/random.hpp"

namespace nvs
{

std::vector<std::pair<int, int>> data_signed(unsigned size,
                                Random type)
{
    using std::vector;
    using std::pair;
    using std::sqrt;

    vector<pair<int, int>> result(size);

    if (type == Random::UNIQUE)
    {
        nvs::randomize();
    }

    int max { static_cast<int>(sqrt(size)) };
    int min { -max };
    for (unsigned idx { 0 }; idx < size; ++idx)
    {
        result[idx] = { nvs::random_value(min, max),
                        nvs::random_value(min, max)
                      };
    }

    return result;
}

std::vector<std::tuple<int, unsigned, unsigned>> data_unsigned(
            unsigned size, Random type)
{
    using std::vector;
    using std::tuple;
```

```
43      using std::sqrt;
44
45      vector<tuple<int, unsigned, unsigned>> result(size);
46
47      if (type == Random::UNIQUE)
48      {
49          nvs::randomize();
50      }
51
52      unsigned max { static_cast<unsigned>(sqrt(size)) };
53      for (unsigned idx { 0 }; idx < size; ++idx)
54      {
55          result[idx] = { nvs::random_value(-1, 1),
56                          nvs::random_value(0u, max),
57                          nvs::random_value(0u, max)
58                        };
59      }
60
61      return result;
62  }
63
64  }
```

E. Random

```
1   /*!
2    * \file random.hpp
3    * \brief Définitions de fonctions conviviales pour générer des
4    *        séquences pseudo-aléatoires.
5    */
6   #ifndef RANDOM_HPP
7   #define RANDOM_HPP
8
9   #include <random>
10  #include <utility>
11  #include <limits>
12  #ifdef _WIN32
13  #include <ctime>
14  #endif
15
16  /*!
17   * \brief Espace de nom de Nicolas Vansteenkiste.
18   *
```

```
 */
namespace nvs
{

// fonctions

/*!
 * \brief Un générateur de nombres uniformément aléatoires.
 *
 * Cette fonction produit et partage un unique
 * générateur de nombres uniformément aléatoires
 * (_Uniform Random Number Generator_).
 * Elle est issue de Random Number Generation in C++11
 * ([WG21 N3551]
 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
 * par Walter E. Brown.
 *
 * _Remarque_ : Sous Windows, c'est un std::mt19937 qui est
 * retourné, sous les autres systèmes d'exploitation c'est
 * un std::default_random_engine. La raison en est qu'avec
 * gcc sous Windows, la première valeur retournée par
 * un std::default_random_engine change peu en fonction de la
 * graine plantée avec nvs::randomize. Pour s'en convaincre,
 * exécuter nvs::random_value(1, 100000) par des instances
 * successives d'un même programme...
 *
 * \return un générateur de nombres uniformément aléatoires.
 */
inline auto & urng()
{
#ifdef _WIN32
    static std::mt19937 u {};
    // https://stackoverflow.com/a/32731387
    // dans le lien précédent : Linux   <-> gcc
    //                          et Windows <-> msvc
#else
    static std::default_random_engine u {};
#endif
    return u;
}

/*!
 * \brief Un peu de bruit.
 *
```

```
63  * Cette fonction met le générateur de nombres uniformément
64  * aléatoires partagé par nvs::urng() dans un état aléatoire.
65  * Elle est issue de Random Number Generation in C++11
66  * ([WG21 N3551]
67  * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
68  * par Walter E. Brown.
69  */
70 inline void randomize()
71 {
72 #ifdef _WIN32
73     urng().seed(std::time(nullptr));
74     // https://stackoverflow.com/a/18908041
75 #else
76     static std::random_device rd {};
77     urng().seed(rd());
78 #endif
79 }
80
81 /*!
82  * \brief Générateur de flottants aléatoires.
83  *
84  * Les flottants produits se distribuent uniformément entre
85  * `min` et `max`, la valeur minimale comprise, la maximale non.
86  *
87  * Si `max` est strictement inférieur à `min`, les contenus de ces
88  * variables sont permutés.
89  *
90  * Cette fonction est largement inspirée par Random Number
91  * Generation in C++11 ([WG21 N3551]
92  * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
93  * par Walter E. Brown.
94  *
95  * _Remarque_ : Par rapport au modèle de fonction
96  * nvs::random_value<T> produisant des entiers aléatoires, les
97  * arguments `min` et `max` sont inversés de sorte à avoir la
98  * valeur nulle (0) comme borne (minimale ou maximale) si la
99  * fonction est appelée avec un seul argument. Notez que cela n'a
100 * pas de réelle incidence sur la signification des paramètres
101 * puisque leurs contenus sont permutés si nécessaire.
102 *
103 * \param max borne supérieure (ou inférieure) de l'intervalle
104 *            dans lequel les flottants sont générés.
105 * \param min borne inférieure (ou supérieure) de l'intervalle
106 *            dans lequel les flottants sont générés.
```

```
107  *
108  * \return un flottant dans l'intervalle semi-ouvert à droite
109  *         [`min`, `max`[ (ou [`max`, `min`[ si `max` < `min`).
110  */
111 inline double random_value(double max = 1., double min = 0.)
112 {
113     static std::uniform_real_distribution<double> d {};
114 
115     if (max < min) std::swap(min, max);
116 
117     return d(urng(), decltype(d)::param_type {min, max});
118 }
119 
120 // fonctions template
121 
122 /*!
123  * \brief Générateur d'entiers aléatoires.
124  *
125  * Les entiers produits se distribuent uniformément entre
126  * `min` et `max`, ces valeurs incluses.
127  *
128  * The effect is undefined if T is not one of : short, int, long,
129  * long long, unsigned short, unsigned int, unsigned long, or
130  * unsigned long long.
131  *
132  * Si `max` est strictement inférieur à `min`, les contenus de ces
133  * variables sont permutés.
134  *
135  * Cette fonction est largement inspirée par Random Number
136  * Generation in C++11 ([WG21 N3551]
137  * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
138  * par Walter E. Brown.
139  *
140  * \param min valeur minimale (ou maximale) pouvant être retournée.
141  * \param max valeur maximale (ou minimale) pouvant être retournée.
142  *
143  * \return un entier entre `min` et `max`.
144  */
145 template<typename T = int>
146 inline T random_value(T min = std::numeric_limits<T>::min(),
147                       T max = std::numeric_limits<T>::max())
148 {
149     static std::uniform_int_distribution<T> d {};
150 
```

```
151     if (max < min) std::swap(min, max);
152
153     return d(urng(), typename decltype(d)::param_type {min, max});
154 }
155
156 } // namespace nvs
157
158 #endif // RANDOM_HPP
```