Séance 3
Mise au point d'une classe Rationnel

1 Objectifs du TP

L'objectif est de disposer d'un type de données pour représenter des nombres *rationnels*, c'est-à-dire des fractions, à l'aide d'une classe.

Plus tard, dans ce TP, nous verrons comment ajouter quelques surcharges des opérateurs arithmétiques qui permettront d'écrire les expressions sous une forme habituelle :

```
Rationnel demi{1,2};
Rationnel tiers{1,3};
Rationnel resultat;
resultat = (demi *  demi) + tiers;}
cout << "le resultat est " << resultat << endl;
```

2 Les normes d'écriture du code

Important : à partir de ce TP, vous respecterez *impérativement* les normes de programmation suivantes :

1. les noms des **classes** commencent par une majuscule et sont écrits en notation *CamelCase* : quand le nom comporte plusieurs mots, la première lettre de chaque mot est en majuscule, et les autres en minuscules. Exemples

```
class Personnage;
class PersonnageNonJoueur;
```

2. Pour chaque classe, il y a 2 fichiers dont le préfixe est exactement le nom de la classe : le fichier d'entête avec le suffixe `.h` (ou `.hpp`) qui décrit la classe et les prototypes des fonctions libres associées, et un fichier `.cpp`, `.cc` ou `.cxx` contenant le code d'implémentation de la classe.

```
// fichier Point.h}
#ifndef POINT_H
#define POINT_H
class Point {
private:
  ...
public:
 Point(int x, int y);
};
#endif
```

```
// fichier Point.cc
#include "Point.h"
```

```
3
4  Point::Point(int x, int y) {
5    ...
6  }
```

3. Dans le fichier d'entête, la variable du préprocesseur qui empêche les inclusions multiples (et mutuelles) est nommée à partir du nom de fichier en majuscules, avec un soulignement à la place du point. On peut y ajouter "INCLUDED", qui précise son rôle. Exemple ci-dessus.
4. les noms des **variables membres** commencent par le préfixe `m_` ou `my_`, et le reste en minuscule, en CamelCase si il y a plusieurs mots :

```
1  int m_window;
2  int m_meilleurScore;
```

5. Les noms des **fonctions membres** (ou méthodes) sont en CamelCase, et commencent par une majuscule.

```
1  void afficherOptions() {
2    ...
3  }
```

6. Les **paramètres et les variable locales** d'une fonction sont écrits en **snake_case** : en minuscule en séparant les mots par des "blancs soulignés". Exemple

```
1  void ajouterBonus(int nombre_de_points) {
2    int total_points = m_points +
         nombre_de_points;}
3      ...
4  }
```

7. Les noms des **constantes** sont en **SNAKE_CASE** majuscule. Exemple

```
1  const int VALEUR_BONUS = 100;
```

3 Etape : mise en place

Tapez un programme de test `test_rationnels.cc` contenant les lignes

```
1  #include "Rationnel.h"
2
3  int main() {
4    Rationnel r1{3,4};
5      cout << "resultat = ";
6    r1.afficher();
7      cout << endl;
8    return 0;
9  }
```

Ecrivez une interface pour les **Rationnels**, dans le fichier `Rationnel.h`. L'accès aux attributs sera privé

```
// Rationnel.h

class Rationnel {
private:
  m_num;
  int m_deno;
public:
  ...
};
```

Ecrivez le code de `Rationnel.cc` qui permettra au programme de test de s'exécuter.

L'exécution du programme affichera :

```
résultat = 3/4
```

Note : dans le constructeur on supposera que le dénominateur n'est jamais nul. Si c'est le cas, le constructeur produira un message d'erreur et arrêtera le programme par `exit(EXIT_FAILURE);`.

4 Etape : normalisation des signes

On souhaite que les rationnels soient construits en les **normalisant**.

Ajoutez dans le programme de test les lignes

```
   Rationnel r2 { 2, -3};
   Rationnel r3 {-4, -5};
   cout << "devrait afficher  -2/3 : ";
   r2.afficher();
   cout << "devrait afficher   4/5 : ";
   r3.afficher();
```

Il va falloir ajouter du code pour changer éventuellement des signes.

Modifiez le constructeur pour qu'il appelle une fonction membre privée `normaliser()` qui s'en occupe.

5 Etape : normalisation

Pour compléter la normalisation, il faut stocker la fraction sous forme réduite, le numérateur et le dénominateur étant premiers entre eux. Ainsi le code

```
Rationnel r4{12,24};
r4.afficher();
```

devra afficher `-1/2`. On utilisera la fonction suivante :

```
/**
* pgcd de deux nombres positifs (non nuls)
```

```
*/

int pgcd(int a, int b){
  do {
    int reste = a % b; // b != 0
    a = b;
    b = reste;
  } while(b!=0);
  return a ;
}
```

6 Etape : accesseurs

Définir des opérations `getNum()` et `getDeno()` qui retournent le numérateur et le dénominateur d'un rationnel. Test à ajouter :

```
cout << "getNum() devrait afficher -1 :";
cout << r4.getNum() << endl ;
  cout << "getDeno() devrait afficher 2 :";
cout << r4.getDeno() << endl ;
```

7 Etape : mutateurs

Définir des opérations `setNum(int)` et `setDeno(int)` qui changent le numérateur et le dénominateur. Attention, à ne pas oublier de normaliser ensuite :

```
Rationnel r5{22,7};
r5.setNum(21);
cout << "setNum : devrait afficher 3/1 : ";
r5.afficher();
r5.setDeno(-6)
cout << "setNum : devrait afficher -1/2 : ";
r5.afficher();
```

8 Etape : constructeur par défaut.

Jusqu'ici vous avez défini et utilisé un constructeur à deux paramètres. Mais si vous avez besoin de définir un tableau de `Rationnels`

```
   Rationnel t[5];
```

il faut définir un constructeur par défaut (sans paramètres) pour la classe `Rationnel`. Ce constructeur donnera une valeur initiale "raisonnable" (par exemple $0/1$).

9 Opérateur de comparaison

Ecrire l'opérateur de comparaison == qui indique si un Rationnel est égal à un autre

```
if ( r2 == r4 ) {
  cout << "oui";
}
```

10 Opérateur d'addition

Ecrire l'opérateur d'addition d'un Rationnel et d'un autre

```
Rationnel x{-3,4};
Rationnel y{-4,3};
Rationnel z = x + y;
```

11 Opérateur de soustraction

Ecrire l'opérateur de soustraction d'un Rationnel et d'un autre

```
Rationnel x{-3,4};
Rationnel y{-4,3};
Rationnel z = x - y;
```

12 Opérateur de multiplication

Ecrire l'opérateur de multiplication d'un Rationnel et d'un autre

```
Rationnel x{-3,4};
Rationnel y{-4,3};
Rationnel z = x * y;
```

13 Opérateur de division

Ecrire l'opérateur de division d'un Rationnel par un autre

```
Rationnel x{-3,4};
Rationnel y{-4,3};
Rationnel z = x / y;
```

14 Tests unitaires Boost

Mettez en forme les tests que vous avez utilisés sous forme de tests unitaires Boost.

15 `toString()` : conversion en chaîne

La fonction `afficher()` écrite au début est trop limitée, elle ne permet que d'écrire sur `cout`. Il est plus intéressant d'avoir une fonction membre - traditionnellement appelée `toString()` qui retourne la chaine de caractère qui représente un rationnel.

Nous pourrons l'utiliser pour d'autres choses, écrire dans un fichier, etc. ou tout simplement dans un "pipeline" d'écritures. Exemple :

```
Rationnel r1 {3,4};
Rationnel r2 {4,5};
cout << r1.toString() << " + " << r2.toString()
<< " = " << r1.plus(r2).toString()
<< endl;
```

C'est facile à faire, il suffit d'utiliser un objet "stringstream", c'est un tampon sur lequel on peut écrire avec "<<" (comme sur `cout`) et on peut récupérer la chaine de texte qu'il contient par `str()` :

```
string toString() const {
  ostringstream tampon;
  tampon << ......;
  return tampon.str();
}
```

Depuis C++11, une autre méthode consiste à utiliser la fonction `to_string`.