TPs 1 et 2
Mise au point d'une classe Rationnel

Nous nous proposons d'écrire une application permettant de manipuler des nombres rationnels. Nous allons pour cela représenter le type Rationnel à l'aide d'une classe. Avant de commencer, voici en vrac quelques bonnes habitudes pour faciliter la lecture (et la relecture) de votre code (ces pratiques sont classiques) :

Conventions de nommage

- Traditionnellement, les noms des données membres d'une classe (ou *attributs* d'une classe) sont en minuscules et commencent par `my_` ou `mon_` ou `m_` ou juste `_`. Si cela n'a rien d'obligatoire, cela permet de distinguer très facilement les variables et les paramètres des données propres à la classe. (Voir l'exemple de la classe `Rationnel` ci-dessous.)
- Les noms de classes ou structures commencent par des majuscules (si ce sont des mots composés, les mots sont accolés et le début de chaque mot est mis en majuscule) : `class Point; class Vecteur; class TriangleIsocele; ...`
- les noms des fonctions membres (ou *méthodes*, ou *opérations*) commencent par une minuscule ; si leur nom est formé de plusieurs mots, mettre une majuscule au début des mots intérieurs : `void initialise(); void translateHorizontalement(float dx); float vitesseAngulaire(); ...`
- les noms des paramètres d'une fonction et des variables locales sont en minuscules ; s'ils sont composés de plusieurs mots, les séparer par `_` : `float delta_x, delta_y; float distance(const Point & autre_point); ...`
- les noms des constantes, même propres à la classe, sont en majuscules ; si ils sont formés de plusieurs mots, mettre des `_` entre : `const float PI = 3.14; const int MAX_NB_ELEMENTS = 100;`

Exercice 1 : La classe Rationnel

1. Proposez une interface (fichier `Rationnel.h`) ; l'accès aux attributs sera privé :

```
class Rationnel {
  private :
    int my_num ;
    int my_deno ;
  public :
  ...
} ;
```

   Dans cette première version, vous offrirez les fonctions membres suivantes :
   - Un constructeur prenant en paramètres deux entiers.
   - affiche : affiche sur la sortie standard le Rationnel sous la forme : `<numérateur> / <dénominateur>`

2. Ecrivez les fonctions membres (fichier `Rationnel.cc`).

3. Ecrivez un programme permettant de tester le fonctionnement de la classe (fichiers TesteRationnel.cc et Makefile).

Exercice 2 : Les constructeurs

Le constructeur que vous avez défini prend en paramètres deux entiers ; mais dans certaines situations, on peut avoir besoin de construire des objets Rationnel sans pouvoir fournir les paramètres ; une situation typique est celle d'un tableau de Rationnel. Il faut donc définir un constructeur par défaut. De plus, pour utiliser des Rationnel définis par défaut, on est généralement conduit à modifier leurs valeurs. Mais le numérateur et le dénominateur sont des attributs privés ! Une solution est que la classe fournisse des accesseurs en écriture.

1. Ecrivez les fonctions membres suivantes :
   — Un constructeur par défaut (quelles valeurs choisir ?).
   — setNum : affecte une valeur au numérateur.
   — setDeno : affecte une valeur au dénominateur.
2. Modifiez votre programme pour vérifier que tout cela fonctionne bien.

Exercice 3 : Les opérations sur les rationnels

1. Ajoutez deux fonctions membres qui permettent de :
   — transformer un rationnel en son inverse,
   — tester l'égalité d'un rationnel avec un autre rationnel.

   Testez à l'aide de votre programme d'essai ces nouvelles fonctionnalités.
2. Vous allez maintenant ajouter les 4 opérations arithmétiques de base : addition, soustraction, multiplication et division que vous prendrez soin de tester. Par exemple, le prototype de la fonction de soustraction sera :

   ```
   void Rationnel::soustraction(const Rationnel & autre, Rationnel & difference) const
   ```

Exercice 4 : Pour aller un peu plus loin

Ecrivez et testez les fonctions membres suivantes :

— reduit : transforme le Rationnel en sa forme réduite (simplifiée). Vous pourrez écrire une fonction utilitaire pgcd ; une solution élégante serait de créer deux fichiers util.h et util.cc dans lequels vous déclarerez puis définirez pgcd.
— toString : retourne la chaîne de caractères correspondant au Rationnel. Cette méthode facilite l'affichage. On peut maintenant écrire :

```
q1.addition(q2,q3);
cout << q1.toString() << ‘‘+’’ << q2.toString() <<‘‘=’’ << q3.toString() << endl;
```

Pour écrire cette méthode, vous définirez une fonction utilitaire intToString en vous inspirant du code suivant :

```
ostringstream oss;
oss << x;
string s = oss.str();
```

Il est possible d'associer un flux à une chaîne de caractères grâce à un `ostringstream` de la biblothèque `sstream`. Nous utiliserons alors ces flux de la même manière que

nous utilisons les flux d'entrée/sortie `cin` et `cout`. A tout moment, nous pourrons récupérer la chaîne associée au flux grâce à la méthode `str()`.

Selon le temps qui vous reste, vous pouvez faire en sorte que la chaîne retournée par toString corresponde aux usages : forme réduite, éventuel signe "-" au numérateur, enfin 3 et non 3/1, et 0 au lieu de 0/12.

Exercice 5 : Une classe Complexe

On rappelle qu'un nombre complexe peut s'écrire $a+ib$ où a et b sont des nombres réels, tandis que i est un nombre dit "imaginaire" et tel que $i^2 = -1$.

a est appelé la partie réelle, b la partie imaginaire du nombre complexe $a + ib$.

Soient deux complexes $a + ib$ et $a' + ib'$, leur somme vaut $(a + a') + i(b + b')$

et leur produit $(aa' - bb') + i(ab' + ba')$

En vous inspirant de la classe Rationnel, écrivez une classe Complexe : `Complexe.h, Complexe.cc, TesteComplexe.cc, Makefile`.

Si vous avez terminé, passez au TP 2 bis.