Exercice 20 - La classe **Fraction**

Créer un projet vide et ajouter trois fichiers `fraction.h`, `fraction.cpp` et `main.cpp`. Définir la fonction principale `main()` dans le fichier `main.cpp`. S'assurer que le projet compile correctement. Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés.

Question 1

Définir en C++ une classe `Fraction` qui comportera deux attributs `numerateur` et `denominateur` de type **`int`**. Englober cette classe dans un espace de noms `MATH`. Dans la fonction `main()`, définir un objet `Fraction` avec la valeur `3/4`.

Question 2

Faire en sorte que les attributs de la classe ne soient plus accessibles par un utilisateur de la classe (si ce n'était pas déjà le cas). Déclarer et définir des accesseurs en lecture des attributs de la classe (`getNumerateur()` et `getDenominateur()`) et un accesseur en édition des attributs la classe (`setFraction(int n, int d)`). Faire attention à la validité des valeurs stockées dans les attributs. Les accesseurs en lecture seront (**`inline`**) alors que l'accesseur en écriture (édition) sera définis dans le fichier `fraction.cpp`.

Question 3

Déclarer et définir un ou plusieurs constructeurs pour cette classe. Faire attention à la validité des valeurs stockées dans les attributs. Faire cette question deux fois : une fois en utilisant des affectations pour donner une valeur initiale aux attributs, et une fois en utilisant des initialisations.

Question 4

Ajouter à la classe `Fraction` **la méthode privée** `simplification()` suivante (en tant que méthode non-**`inline`**) qui permet de simplifier une fraction :

```
// à ajouter en tant méthode privée de la classe Fraction
void MATH::Fraction::simplification(){
 // si le numerateur est 0, le denominateur prend la valeur 1
 if (numerateur==0) { denominateur=1; return; }
 /* un denominateur ne devrait pas être 0;
    si c'est le cas, on sort de la méthode */
 if (denominateur==0) return;
 /* utilisation de l'algorithme d'Euclide pour trouver le Plus Grand Commun
    Denominateur (PGCD) entre le numerateur et le denominateur */
 int a=numerateur, b=denominateur;
 // on ne travaille qu'avec des valeurs positives...
 if (a<0) a=-a; if (b<0) b=-b;
 while(a!=b){ if (a>b) a=a-b; else b=b-a; }
 // on divise le numerateur et le denominateur par le PGCD=a
 numerateur/=a; denominateur/=a;
 // si le denominateur est négatif, on fait passer le signe - au denominateur
 if (denominateur<0) { denominateur=-denominateur; numerateur=-numerateur; }
}
```

fraction.cpp

Utiliser cette méthode pour améliorer le(s) constructeur(s) et/ou l'accesseur en écriture quand cela vous paraît utile.

Question 5

Ecrire la méthode `somme` qui permet de faire une addition de 2 fractions. Notons que $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$. Réfléchir aux types de retour possibles pour cette fonction.

Question 6

Ajouter un destructeur à la classe `Fraction`. Ce destructeur contiendra une instruction permettant d'afficher sur le flux `cout` un message de destruction avec l'adresse de l'objet concerné. De même, ajouter un message de construction dans le ou les constructeurs de la classe `Fraction`. Recopier le code suivant dans le fichier `main.cpp`, exécuter le programme et analyser les constructions et destructions des objets.

```
#include <iostream>
#include "fraction.h"
```

```
using namespace std;
using namespace MATH;

Fraction* myFunction(){
 Fraction fx(7,8);
 Fraction* pfy=new Fraction(2,3);
 return pfy;
}

int main(){
  Fraction f1(3,4);
  Fraction f2(1,6);
  Fraction* pf3=new Fraction(1,2);
  cout<<"ouverture d'un bloc\n";
  Fraction* pf6;
  {
   Fraction f4(3,8);
     Fraction f5(4,6);
     pf6=new Fraction(1,3);
  }
  cout<<"fin d'un bloc\n";
  cout<<"debut d'une fonction\n";
  Fraction* pf7=myFunction();
  cout<<"fin d'une fonction\n";
  cout<<"desallocations controlee par l'utilisateur :\n";
  delete pf6;
  delete pf7;
  return 0;
}
```

fraction.cpp

Question 7

Représenter la classe `Fraction` en UML.

Exercice 21 - La classe `Fraction` : surcharge d'opérateurs

Question 1

Surcharger l'opérateur binaire + de façon à pouvoir effectuer la somme entre deux objets `Fraction`.

Question 2

Après avoir étudié la possibilité d'une conversion implicite du type **`int`** vers le type `Fraction`, surcharger (si besoin) l'opérateur + de manière à rendre possible cette opération entre valeurs de type **`int`** et valeurs de type `Fraction`.

Question 3

Surcharger l'opérateur ++ en version préfixe et postfixe pour la classe `Fraction`.

Question 4

Surcharger l'opérateur << qui pourra permettre d'afficher une fraction en utilisant un flux `ostream` comme `cout`. Faire attention au type de retour de la fonction.

Exercice 22 - La classe `Fraction` : Exceptions

Question 1

Réécrire la méthode `Fraction::setFraction` de façon à traiter l'erreur d'un éventuel dénominateur nul en déclenchant une exception. Le mot clé **`throw`** sera utilisé avec une littérale chaîne de caractère décrivant l'erreur.

Question 2

Dans la fonction `main`, ajouter les instructions qui permettent de capturer une éventuelle exception due à un dénominateur nul et d'afficher le message correspondant.

Question 3

Ecrire une classe `FractionException` destinée à gérer les situations exceptionnelles de la classe `Fraction`. Cette classe comportera un attribut `info` de type **`char`**`[256]` destinée à stocker un message d'erreur et une méthode `getInfo()` permettant d'accéder à cette information. La classe comportera également un constructeur permettant de construire un objet `FractionException` avec une chaine de caractères.

Question 4

Réécrire la méthode `Fraction::setFraction` de façon à traiter l'erreur d'un éventuel dénominateur nul en déclenchant une exception de type `FractionException`.

Question 5

Dans la fonction `main`, modifier les instructions de manière à capturer les exceptions de type `FractionException`.