# Classe Fraction

On se propose de créer une classe « Fraction » permettant de représenter des nombres fractionnaires.

## Rappels :

Un nombre fractionnaire est repésenté par deux valeurs situées de part et d'autre d'une barre de fraction. Ces deux valeurs sont appelées « numérateur » et « dénominateur ».

$\frac{a}{b}$ est une fraction. a est le numérateur ; b est le dénominateur

### Addition de deux fractions

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd}$$

### Soustraction de deux fractions

$$\frac{a}{b}-\frac{c}{d}=\frac{ad-cb}{bd}$$

### Multiplication de deux fractions

$$\frac{a}{b}x\frac{c}{d}=\frac{ac}{bd}$$

### Division de deux fractions

$$\frac{\frac{a}{b}}{\frac{c}{d}}=\frac{a}{b}x\frac{d}{c}$$

## Travail demandé

Créer une classe « Fraction » ayant deux valeurs de type *double* : *num* et *denom*

Vous ajouterez un attribut de type *double* : *reel* représentant la valeur numérique de la fraction

## Méthodes de la classe « Fraction »

```
//constructeurs
Fraction(double nume=1,double denomi=1);
Fraction(const Fraction & uneCopie);


//Les opérateurs internes de base:
Fraction &operator=(const Fraction &);
Fraction &operator+=(const Fraction &);
Fraction &operator-=(const Fraction &);
Fraction &operator*=(const Fraction &);
Fraction &operator/=(const Fraction &);
```

Public

```
//operateurs externes amies
friend Fraction operator+(const Fraction &, const Fraction &);
friend Fraction operator-(const Fraction &, const Fraction &);
friend Fraction operator*(const Fraction &, const Fraction &);
friend Fraction operator/(const Fraction &, const Fraction &);
//traitements internes
void valeurReelle();
```

Private

```
//Accesseurs
double getNum() const;
double getDenom() const;
double getReel() const;
```

Public

# SURCHARGE D'OPERATEURS

## Explications sur les méthodes

### *Les constructeurs*

```
Fraction(double nume=1,double denomi=1);
```

Il permet la création d'un objet éventuellement sans paramètres.
Si on ne donne aucun paramètre, alors les valeurs précisées dans le prototype (ci-dessus) sont utilisées pour initialiser les attributs, si tant est que vous ayez mis une liste d'initialisation !

```
Fraction(const Fraction& uneCopie);
```

Constructeur par recopie. « uneCopie » est une référence sur un objet précédemment créé. Le mot clé ***const*** est nécessaire pour éviter une modification intempestive de l'objet passé en référence.

### *Opérateurs internes*

#### Affectation :

```
Fraction& operator=(const Fraction&);
```

Il permettra d'écrire *fr2 = fr1*  *//fr1 et fr2 sont des objets « Fraction »*
Il arrive très souvent qu'il soit associé aux constructeurs alors qu'il n'a rien à voir.
Attention, si on écrit :

*Fraction fr2=fr1 ;*  *//le constructeur par recopie est appelé et pas //l'opérateur d'affectation !!!*

#### Addition et affectation :

```
Fraction &operator+=(const Fraction &);
```

Il permettra d'écrire *fr2+=fr1*  (ce qui revient à *fr2 = fr2 + fr1*)

#### Soustraction et affectation :

```
Fraction &operator-=(const Fraction &);
```

Il permettra d'écrire *fr2-=fr1*  (ce qui revient à *fr2 = fr2 - fr1*)

#### Multiplication et affectation :

```
Fraction &operator*=(const Fraction &);
```

Il permettra d'écrire *fr2*=fr1*  (ce qui revient à *fr2 = fr2 * fr1*)

#### Division et affectation :

```
Fraction &operator/=(const Fraction &);
```

Il permettra d'écrire *fr2/=fr1*  (ce qui revient à *fr2 = fr2 / fr1*)

### *Opérateurs externes*

Pourquoi externe ?

#### Rappel

On veut pouvoir écrire : *fr = fr1+fr2*
Si on a développé l'opérateur *operator+()* comme un opérateur interne, au même titre que *operator+=()*, alors l'écriture précédente est remplacée par le compilateur par :
*fr.operator=( fr1.operator+(fr2) )*  *//évidemment ce seront les références qui seront passées*
C'est donc l'objet *fr1* qui exécute son opérateur *operator+()*
Si, maintenant, on écrit : *fr = fr2+fr1* alors c'est *fr2* qui exécute son opérateur *operator+()*
On a donc recréer une addition pour la classe « Fraction » mais elle n'est pas commutative !
La solution : créer un opérateur *operator+()* en dehors de la classe « Fraction »

- ⇨ C'est une fonction comme en C !

Problème : il faut que cette fonction puisse accéder au contenu privé de la classe. Il est d'ailleurs conseillé de les déclarer en amies privées.

- ⇨ Il y a la possibilité d'indiquer dans la classe qu'il y a des fonctions « amies » qui auront le droit d'accéder au contenu privé en lecture (si on a précisé que les paramètres à ces fonctions sont des références sur des objets constants)

### *Accesseurs*

Pour le moment, on a juste besoin des méthodes *get*.
Remarque : désormais, on prendra l'habitude, pour les méthodes *get* de préciser que ce sont des méthodes constantes. C'est-à-dire qu'elles n'ont pas vocation à modifier le contenu de l'objet qui l'exécute.
Cela s'écrit de la façon suivante : *double getcequonveut() const ;*
A l'implémentation :

```
double getcequonveut() const
{
}
```

### *Traitements internes*

On désire avoir une fonction qui sera appelée systématiquement. Son rôle est de calculer la valeur réelle de la fraction et affecter cette valeur à l'attribut « *reel* ».

## Test de la classe

Dans la fonction *main()*, créer deux fractions « *fr1* » et « *fr2* » dont vous préciserez les valeurs des numérateurs et dénominateurs (ici 3/5 et 7/9). Créer une troisième fraction « *fr* » et procéder aux calculs succesifs de façon à obtenir l'affichage suivant:

```
FRACTIONS
(3/5)+(7/9) = 62/45 = 1.37778
(3/5)-(7/9) = -8/45 = -0.177778
(3/5)*(7/9) = 21/45 = 0.466667
(3/5)/(7/9) = 27/35 = 0.771429
```

## Aide

---

Voici le code de l'opérateur interne *operator=()*. Vous pouvez vous en inspirer pour coder les autres opérateurs

```
Fraction& Fraction::operator=(const Fraction& uneFraction)
{
    num = uneFraction.num;
    denom = uneFraction.denom;
    valeurReelle();
    return *this;
}
```

Remarque : on renvoie le contenu pointé *this*

- ⇨ C'est en fait une référence sur l'objet lui-même !
- ⇨ Cela va permettre de chaîner les opérations. Par exemple : *fr = fr1+fr2+fr3*

---

Voici le code de l'opérateur externe *operator+()*. Vous pouvez vous en inspirer pour coder les autres opérateurs

```
Fraction operator+(const Fraction &f1,const Fraction &f2)
{
    Fraction result = f1;
    return result += f2;
}
```

---

## Evolutions

Des problèmes pourront surgir en cours d'utilisation. Par exemple : on n'a pas géré le cas de la division par 0.

Les simplifications de fractions sont à ajouter dans la classe. Pour cela, il va falloir calculer le pgcd entre le numérateur et le dénominateur. Par exemple, si on fait

$$\frac{3}{5} x \frac{7}{9} = \frac{21}{45} = \frac{7}{15}$$

Le pgcd de 21 et 45 est 3. Donc on divise le numérateur et le dénominateur par 3 pour réduire la fraction

Proposer un algorithme de calcul du pgcd de deux nombres. Si possible, implémentez le et ajoutez l'appel dans vos opérateurs.