Programmation orientée objet en C++

Travaux pratiques séance 1

A la fin du TP, déposez une archive avec votre travail (code source) sur ecampus, dans le repertoire correspondant à votre groupe. Avant minuit, déposez également sur ecampus le compte-rendu détaillé (CR) correspondant à votre travail ; les modifications dans le code ultérieures à la fin du TP ne sont pas prises en compte - concentrez vous sur la rédaction du CR.
Conseils de base :

- le TP est **individuel** ; toute fraude sera sanctionnée drastiquement.
- dès que possible (toutes les quelques lignes idéalement), **compilez** et **testez** votre code. L'erreur la plus fréquente des débutants est d'écrire des dizaines de lignes de code sans tester, et de se retrouver tout à coup avec plein d'erreurs de compilation et de bugs
- si vous dupliquez du code dans vos méthodes, cela veut dire qu'il y a probablement un problème : soit une méthode devrait appeler l'autre, soit le code commun devrait être mis dans une méthode auxiliaire, qui est appelée chaque fois que sa fonctionnalité est nécessaire
- même si cela ne vous rapporte pas une meilleure note tout de suite, essayez d'aller au bout de ce sujet avant le TP suivant. Cela vous améliorera sûrement votre niveau, ainsi que les notes suivantes.

Des éléments pris en compte lors de la notation :

- vous avez respecté ce qu'on demande (la date limite pour l'envoi de votre code et compte-rendu, le format de la soumission etc.)
- votre code contient des **commentaires** qui justifient bien le rôle de chaque classe, membre de classe, méthode, et qui expliquent les détails moins évidents d'implémentation
- le compte-rendu est un document soigné et bien organisé (introduction, rappel bref de l'objectif, contenu, discussion finale) qui justifie tous vos **choix** de design des classes (i.e. pour quoi ce membre est initialisé par défaut ainsi, pour quoi l'allocation de ce tableau se fait dans cette méthode et pas dans le constructeur etc.), mais on ne met ni les détails d'implémentation (pour cela on a les commentaires), ni des copier-coller des classes et des méthodes entières pour faire du volume
- la syntaxe de votre programme est **correcte** (i.e. le code compile)
- votre programme fonctionne correctement (i.e. le résultat correspond à ce qui est demandé), les tests demandés sont implémentés
- votre code est de bonne qualité (i.e. pas de fuites de mémoire, listes d'initialisation pour les constructeurs, encapsulation efficace, utilisation correcte des attributs vus en cours (public, private, const, static etc.) bonne organisation du code en méthodes)

1 Classe Fract

Le but de ce TP est de réaliser une classe pour la manipulation de fractions, mettant en œuvre les bases du C++ , à savoir la forme de Coplien, la surcharge d'opérateurs, la maîtrise du passage d'objets par référence et la séparation entre fonctions membres (méthodes) de la classe et fonctions libres (comme en C). Ce TP mettra aussi en évidence l'utilisation de tests unitaires pour valider les différentes fonctions. Ce TP vous présente un grand nombre des bons usages à avoir lors de la conception d'une classe.

Exercice 1 — La forme de Coplien

La classe `Fract` possède deux variables `private` de type `int` qui servent à représenter la fraction : `num` qui est le numérateur et `denom` qui est le dénominateur. Les tests unitaires à réaliser se feront dans la fonction `void test_constructor(void)` (que vous invoquerez dans votre fonction `main`).

1. Écrire dans les fichiers `Fract.h` et `Fract.cpp` la forme de Coplien de cette classe, en vous inspirant du modèle de classe fourni dans les fichiers `fcc.h` et `fcc.cpp`. Le constructeur par défaut ne prend pas de paramètre ; justifiez et implémentez dans ce cas une initialisation cohérente par défaut pour `num` et `denom`.
2. Ajouter deux autres constructeurs, l'un prenant un seul paramètre (le paramètre second étant par défaut - justifiez votre choix), l'autre en prenant deux.
3. Écrire une méthode qui affiche la fraction en console, qui vous permettra de tester vos constructeurs dans la fonction `void test_constructor(void)`. L'affichage sera de la forme `[ 3 / 4 ]`.
4. Pour faciliter le debug, affichez dans chaque constructeur, destructeur, méthode un message spécifique d'information à la fonction respective (de la même manière que dans la classe modèle `fcc` fournie).
5. Quelle valeur est interdite pour une fraction ? Ajouter un test au niveau de la construction via la fonction `assert` pour la détecter, afin de vous assurer que les objets créés ne pourront jamais se retrouver dans cette configuration.

Exercice 2 — Constructeur de copie, opérateur d'affectation

1. Compléter le code du constructeur de copie et tester-le dans la fonction `void test_constructor(void)`.
2. Compléter le code de l'opérateur de copie `operator=` pour réaliser la copie entre deux objets `Fract`.
3. Dans le cas où les objets occupent beaucoup de place en mémoire (pas le cas ici), il est important d'éviter l'auto-affectation `(f = f;)`. Ajouter un test en `void test_constructor(void)` pour vérifier que le modèle fourni en `fcc.cpp` traite bien ce cas, et expliquer quel serait l'inconvénient de ne pas gérer ce cas particulier.

Exercice 3 — Assesseurs et mutateurs (méthodes "setter" et "getter")

Les tests unitaires à réaliser se feront dans la fonction `void test_get_set(void)`.

1. Écrire les assesseurs `get_num()` et `get_denom()` comme fonctions membre de classe.
2. Écrire les mutateurs `set_num(int a)` et `set_denom(int b)` comme fonctions membre de classe. Quelle est la condition que le mutateur doit verifier pour assurer la coherence des objets `Fract` ?
3. Si la forme des mutateurs est acceptable et lisible `f.set_num(1)`, la forme des assesseurs est plus discutable et moins lisible : `a = f.get_num()`. Ajouter les fonctions libres `numerator` et `denominator` afin de pouvoir écrire `a = numerator(f)`. Attention : quel est le type de l'argument `f` ?
4. Proposez en `void test_get_set(void)` des tests qui montrent l'implémentation correcte de ces méthodes et fonctions.

Exercice 4 — Affichage et debug

Les tests unitaires à réaliser se feront dans la fonction `void test_counter(void)`.

1. Afin d'améliorer les capacités de debug, on souhaite ajouter un compteur de fraction. Pour cela ajouter une variable `int id` dans chaque instance de la classe ainsi qu'un compteur global `static int gid` (unique pour toute la classe) et qui sera incrémenté lors de chaque création (attention à mettre à jour les constructeurs). Rappel : la variable `gid` doit être initialisée à zéro dans le fichier `Fract.cpp`. Modifier la fonction d'affichage pour que la forme de l'affichage soit `#4 : [ 1 / 2 ]`, si cette fraction est la quatrième à être créée.
2. Ajouter l'assesseur `get_id` et la fonction libre `id`.
3. Imaginons maintenant que l'affichage des message d'info soit coûteux et qu'on ne veuille l'activer que dans certains cas, durant la mise au point du code. Pour cela, ajouter dans le début du fichier `Fract.h` le define suivant : `#define ENABLE_DBG` qui, uniquement s'il est défini, fait que toutes les messages d'information pour chaque appel de méthode soient affichés. Afin de ne pas polluer le code avec des `#ifdef ENABLE_DBG`, on écrit une méthode `print_trace(const char*)` qui doit afficher l'argument ou pas en fonction du mode debug choisi.

Exercice 5 — Allocation dynamique

Les tests unitaires à réaliser se feront dans la fonction `void test_alloc_dyn(void)`.

1. Allouer **dynamiquement** des fractions en utilisant les différents constructeurs.
2. Allouer dynamiquement un **tableau de fractions** et copier des fractions dans ce tableau.
3. Allouer dynamiquement un **tableau de pointeurs sur fractions** et copier des fractions dans ce tableau.
4. Vérifier - grâce au mode debug - que toutes les allocations dynamiques sont bien désallouées grâce à un `delete` ou un `delete[]` écrit par vous (et pas automatiquement à la sortie de la fonction `main`).

Exercice 6 — Calcul de PGCD, réduction et normalisation de fraction

La méthode de calcul de fractions avec les 4 opérateurs arithmétiques fait que le numérateur et dénominateur peuvent grandir très vite - de manière quadratique en fait - à cause des produits en croix réalisés. Il est nécessaire de simplifier les fractions à chaque calcul. L'habitude veut aussi que si une fraction est négative, le signe moins soit porté par le numérateur et non le dénominateur.
Les tests unitaires à réaliser se feront dans la fonction `void test_reduce(void)`.

1. Calculer sur papier en utilisant l'algorithme fourni à la fin de ce document, en détaillant bien chaque pas, le pgcd de 36 et 24, et le pgcd de 72 et 25. Mettre ces calculs dans votre compte-rendu.
2. Coder la fonction libre `int pgcd(int a, int b)` qui calcule le pgcd de deux nombres et la tester.
3. Une fois la fonction validée, en faire une méthode privée de la classe `Fract`
4. Coder la fonction privée `reduce()` qui réduit une fraction. Lister l'ensemble des fonctions où elle doit être appelée.
5. Coder la fonction privée `normalize()` qui teste et positionne le signe d'une fraction sur le numérateur.
6. Proposer des tests unitaires qui couvrent tous les cas de gestion de signe, et vérifier que tout se passe bien. Testez également que la réduction à base de pgcd se fait automatiquement.

Exercice 7 — Opérateurs arithmétiques

Les tests unitaires à réaliser se feront dans la fonction `void test_operator_arithm(void)`.

1. Surcharger les opérateurs arithmétiques (fonctions membres de la classe) suivants : `+`, `-`, `*` et `/`. Cela vous permettra d'effectuer très facilement des opérations arithmétiques avec les objets de type `Fract`. Attention : veillez à tester ce qu'il faut dans toutes les situations ou la cohérence de la classe risque de ne pas être respectée.
2. Surcharger également l'opérateur de comparaison `==`, et utilisez le pour vérifier dans le cadre de la fonction `void test_operator_arithm(void)` que les opérateurs surchargés préservent les propriétés de commutativité, associativité, identité, inversibilité et distributivité :

$$\frac{a}{b}+\frac{c}{d}=\frac{c}{d}+\frac{a}{b} \qquad \frac{a}{b}\cdot\frac{c}{d}=\frac{c}{d}\cdot\frac{a}{b}$$

$$(\frac{a}{b}+\frac{c}{d})+\frac{e}{f}=\frac{a}{b}+(\frac{c}{d}+\frac{e}{f})$$

$$\frac{a}{b}+0=0+\frac{a}{b}=\frac{a}{b} \qquad \frac{a}{b}\cdot 1=1\cdot\frac{a}{b}=\frac{a}{b}$$

$$\frac{a}{b}\cdot\frac{b}{a}=\frac{b}{a}\cdot\frac{a}{b}=1$$

$$\frac{a}{b}\cdot(\frac{c}{d}+\frac{e}{f})=\frac{a}{b}\cdot\frac{c}{d}+\frac{a}{b}\cdot\frac{e}{f}$$

2 Rappel mathématique

Les propriétés du pgcd sont :

$$pgcd(0,0) = 0$$

$$pgcd(u,v) = pgcd(v,u)$$

$$pgcd(u,v) = pgcd(-v,u)$$

$$pgcd(u,0) = |u|$$

$$pgcd(u,v) = pgcd(v,u-qv), q \in N$$

L'algorithme conseillé pour calculer le pgcd est celui proposé par Knuth. Si u et v sont les deux entiers en entrée, on applique les pas suivants :

1. si $v = 0$, retourne u comme résultat
2. sinon, $r \leftarrow u \mod v$, $u \leftarrow v$, $v \leftarrow r$ et on retourne au pas (1)