INF1010

Programmation Orientée-Objet

Travail pratique #3

Héritage

Objectifs :	Permettre à l'étudiant de se familiariser avec l'héritage.
Remise du travail :	Lundi le 22 octobre 2018, 8h
Références :	Notes de cours sur Moodle & Chapitre 19 du livre Big C++ 2e éd.
Documents à remettre :	La solution ainsi que les fichiers .cpp et .h complétés réunis sous la forme d'une archive au format .zip.
Directives :	Directives de remise des Travaux pratiques sur Moodle Les en-têtes (fichiers, fonctions) et les commentaires sont obligatoires. Les travaux dirigés s'effectuent obligatoirement en équipe de deux personnes faisant partie du même groupe. Veuillez suivre le guide de codage

Le travail consiste à continuer l'application de gestion de dépenses commencée au TP1 et TP2. On imaginera que les transferts se feront de manière automatique au sein de l'application et de ce fait l'application PolyCount prend un **intérêt** sur la transaction.

L'héritage permet de créer une classe de base possédant ses propres caractéristiques et de l'utiliser pour créer de nouvelles classes auxquelles on peut attribuer la caractéristique « est un ».

Pour réussir ce TP il faudra vous familiariser avec la fonction C++ static_cast<T>(Objet). Cet opérateur permet de convertir à la compilation un objet d'une classe en objet d'une autre classe. Dans le cas du TP, par exemple il servira à transformer un pointeur du type Utilisateur vers un pointeur du type réel de ce produit (UtilisateurRegulier, UtilisateurPremium).

Pour vous aider, les fichiers du TP précédent vous sont fournis. Vous n'avez qu'à implémenter les nouvelles méthodes et les modifications (attributs et méthodes) décrites plus bas.

ATTENTION : Tout au long du TP, assurez-vous d'utiliser les opérateurs sur les objets et non sur leurs pointeurs ! Vous devez donc déréférencer les pointeurs si nécessaires.

ATTENTION : Tout au long du TP, veillez à adapter les méthodes en fonction de la nouvelle structure du code.

ATTENTION : Vous serez pénalisés pour les utilisations inutiles du mot-clé *this*. Utilisez-le seulement où nécessaire.

ATTENTION : Il est fortement recommandé d'utiliser les fichiers fournis, plutôt que de continuer avec vos fichiers du TP2.

Classe *Depense*

La classe de base Depense caractérise une dépense utilisée à des fins individuelle, c'est-à-dire pour un seul utilisateur.

Cette classe possède en plus les attributs suivants :

TypeDepense type_ : attribut lors de la création permet de connaître le type de l'objet en question (dérivé ou non).

Les méthodes :

Le constructeur : il faut modifier le constructeur pour initialiser le type à *individuelle.*
getType() : retourne la valeur de l'attribut type_;
setType() : permet de changer le type de depense;

Classe *DepenseGroupe*

Classe héritée de la classe *Depense*, elle caractérise une dépense utilisée pour un groupe.

Cette classe contient l'attribut privé suivant :

- nombreParticipants_ : indique combien de personne se partagent cette dépense, est initialisé à 0 et sera changée dans le groupe (voir plus bas)

Les méthodes:

- Un constructeur par défaut et par paramètres qui utilise le constructeur par paramètres de Depense et initialise également l'attribut nompreParticipants_ à 0 et le type à *groupe.*
- La méthode d'accès et de modification du nombre de participants.
- getMontantPersonnel() : retourne le montant à payer par personne (attention retourne 0 si le nombre de participants est à 0);
- La surcharge de l'opérateur **<<** pour afficher une dépense groupée. Utilisez une forme d'affichage similaire à la classe Depense. Affichez y également le nombre de personnes impliquées et le montant personnel.

Note : Penser à l'utiliser static_cast.

Classe *Utilisateur*

Cette classe caractérise un utilisateur, elle sera utilisée pour créer les classes dérivée : *UtilisateurPremium* et *UtilisateurRegulier*.

Cette classe contient les attributs privés suivants :

- double interet_ : représente l'intérêt prix par PolyCount.
- string nom_;
- TypeUtilisateur type_ : représente le type d'utilisateur
- vector <Depense*> depenses_;
- double totalDepense_;.

Les méthodes suivantes doivent être implémentées :

- Un constructeur par paramètres et par défaut utilisant le constructeur de la classe Produit et initialisant le interet à 0 et le type à *Regulier*;
- AjouterInteret() : prend en paramètre un montant à ajouter à l'intérêt.
- Les méthodes de modification d'accès aux attributs définis dans cette classe.

La surcharge de l'opérateur << affichant en plus l'intérêt.

Classe *UtilisateurRegulier*

Dérivée de la classe utilisateur, elle représente un utilisateur régulier (n'ayant pas souscrit d'abonnement premium). Elle a l'a particularité d'être sujette à un taux d'intérêt constant de 0.05 (soit 5%) (le taux d'intérêt représente la fraction prise par PolyCount sur le montant d'un transfert en tant que donneur), de plus un utilisateur régulier peut être inscrit dans un seul groupe à la fois.

Cette classe contient les attributs suivants :

estGroupe_ : un booléen représentant si l'utilisateur est groupé.

Les méthodes suivantes doivent être implémentées :

- Le constructeur de fournisseur initialise les attributs de la classe Usager, estGoupe à false;
- Le constructeur de copie.
- Les méthodes d'accès aux attributs.
- La méthode de modification de l'attribut estGroupe_;
- La surcharge de l'opérateur =. Utiliser la surcharge de l'opérateur = de la classe Utilisateur.

La fonction operator <<. Penser à l'utiliser l'opérateur << de la classe de base.
Afficher l'état de estGroupe_.

Classe *UtilisateurPremium*

La classe UtilisateurPremium dérive de la classe Utilisateur. Elle a la particularité de pouvoir être dans plusieurs groupes, de plus le taux d'intérêt est variable (débutant à 5%, il diminue de 1% toutes les deux dépenses).

Les attributs sont :

-joursRestants_ : un entier non signé représentant le nombre de jours restants avant que l'abonnement premium se finisse.

-taux_ : un double représentant le taux d'intérêt.

Les méthodes :

- Le constructeur de UtilisateurPremium initialise les attributs de la classe Utilisateur, le nombre de jour restant à 30 et le taux à 0.05.
- Le constructeur de copie.
- Les méthodes d'accès aux attributs.
- La méthode de modification du nombre de jours restants
- La méthode calculerTaux() qui calcul le taux en fonction du nombre de dépenses.
- La surcharge de l'operateur = (utilise la surcharge de la classe Utilisateur)
- fonction operator << . Penser à l'utiliser l'opérateur << de la classe de base. Afficher le taux et le nombre de jours.

Classe *Groupe*

Caractérise un groupe, tout comme dans les deux premiers TPs.

Les attributs suivants :

```
string nom_;
vector<Utilisateur*> utilisateurs_;
vector<DepenseGroupe*> depenses_;
vector<Transfert*> transferts_;
double totalDepenses_;
vector<double> comptes_;
```

Les méthodes suivantes doivent être implémentés ou changées :

- Les methodes d'accès.

- La méthode ajouterDepense, prend alors en paramètre un utilisateur (celui qui paie la dépense) et un vecteur d'utilisateur (ceux avec qui la dépense sera partagée) :
    - ✓ Verifier que tous les utilisateurs concernés soient là.
    - ✓ Verifie que la dépense soit bien une DepenseGroupe
    - ✓ Si tout est bon :
        - Ajoute la dépense aux utilisateurs concernés
        - Mets à jour les comptes des utilisateurs concernés
        - Ajoute la dépense au groupe
    - ✓ Sinon affiche une erreur
- L'opérateur += qui ajoute un utilisateur* (pensez à vérifier si l'ajout est possible en respectant la logique des abonnements premium)
- La méthode calculerTotalDepense() qui mets à jour le total dépense de tous les utilisateurs (en utilisant la méthode déjà implémentée) et du groupe.
- EquilibrerCompte() qui devra être modifiée dans le but d'ajouter l'intérêt aux utilisateurs ayant le rôle de donneur dans les transferts.
- L'opérateur << qui affiche les utilisateurs, et soit les transferts, soit les comptes.

Main.cpp

Le fichier main.cpp vous est donné, ne pas le modifier.

Spécifications générales

• Ajouter un destructeur pour chaque classe chaque fois que cela vous semble pertinent
• Utilisez la liste d'initialisation pour l'implémentation de vos constructeurs
• Ajouter le mot-clé const chaque fois que cela est pertinent
• Appliquez un affichage propre et joli.
• Documenter votre code source
• Note : Lorsqu'il est fait référence aux valeurs par défaut, pour un string cela équivaut à la chaîne vide, et pour un entier au nombre 0

Questions

1. Pourquoi est-il logique de dériver une classe UtilisateurPremium et une classe UtilisateurRegulier d'une classe Utilisateur?

2. Dans la surcharge de l'opérateur << dans la classe Groupe.
    1. Quelle est l'importance de l'utilisation d'un static_cast ?
    2. Quel effet aura-t-on si on ne le considère pas dans l'implémentation ?

Correction

La correction du TP se fera sur 20 points.

Voici les détails de la correction :

- **(4 points) Compilation du programme ;**
- **(4 points) Exécution du programme ;**
- **(4 points) Comportement exact des méthodes du programme ;**
- **(3 points) Utilisation adéquate de l'héritage ;**
- **(2 points) Documentation du code . Qualité du code**
- **(3 points) Réponse aux questions.**