Fondamentaux de la POO: Héritage Simple & (Héritage Multiple)

Les exercices sont présentés grossièrement dans un ordre croissant de difficulté. Comme vous le constaterez très souvent en Programmation, un problème peut être résolu de différentes manières. Il peut donc plusieurs solutions à un exercice. Cependant, les meilleures solutions sont souvent les plus simples, et définitivement celles qui suivent la logique du cours et des travaux dirigés.

Les exercices marqués d'une étoile (*) sont réservés aux plus avancés sur les notions de base.

Objectifs

— Comprendre l'héritage simple et les modalités d'accès à la classe de base
— Compatibilité entre objets d'une classe de base et objets d'une classe dérivée
— Assimiler la notion de rédéfinition des membres
— Comprendre le fonctionnement de l'héritage multiple

1 Héritage Simple

Pour étudier la notion d'héritage simple, on se propose d'étudier une espece particulière de la famille des canidés appartenant au règne animal : le Chien [1].

EXERCICE 1 :

Dans cette exercice, nous allons créer une classe générale regroupant des informations relatives à tout animal. Cette classe sera appelée la classe mère.

— Dans un fichier header, déclarez une classe `Animal` à deux champs (un entier *age* et une chaine de caractères *nom_du_cri*). Ajouter le constructeur par défaut (ne fait rien), le constructeur à deux paramètres (initialise aussi par défaut les champs), le constructeur de copie et le destructeur par défaut. Prendre soin de déclarer les champs protected.
— Définir les méthodes déclarées précédemment
— Déclarer et définir une méthode `viellir` qui augmente d'une unité.
— Déclarer et définir une méthode `presenter` qui affiche l'age et le type de cri de l'animal (aboiement, henissement, roucoulement, ...) :

"L'animal a `age` ans et `nom_du_cri`."

EXERCICE 2 :

Définissons maintenant une classe fille `Chien` qui hérite publiquement de la classe mère déclarée

1. L'énoncé original sur l'héritage simple est de David Auber

dans l'exercice 1. Nous supposerons et admettrons ici que l'aboiement du chien dégage différentes sonorités : "waaf" ou "woof" ou encore "grrrh". Essayez d'en trouver d'autres (sans faire du bruit ;-)).

1. Ajouter à la classe fille un champ *cri* et les constructeurs et destructeur habituels. Toutes les méthodes seront déclarées public et les champs protected. le champ `nom_du_cri` est initialisé à "aboie" dans la liste d'initialisation du constructeur à paramètre avec la syntaxe ci-dessous :

$$Chien : Animal(age, "aboie");$$

2. Rédéfinir dans cette classe fille la méthode `presenter`. Elle présentera l'animal différemment suivant son âge.
   — S'il est jeune (moins de 6 ans) : "Le chien a `age` ans et aboie : `cri cri cri`."
   — Sinon : "Le chien a `age` ans et aboie : `cri`."

Dans le corps de votre `main`, définir un objet de la classe `Chien` et lui appliquer la méthode *presenter*. Constater quelle méthode est appelée. Essayer ensuite avec $Animal :: presenter()$.
Déclarer un objet `Animal` *chouchou* et définir ensuite un objet `Chien` *rex* avant d'essayer d'exécuter `chouchou = rex`. Essayez aussi l'inverse. Discuter !
Tester les droits d'accès à la fonction membre `vieillir` en fonction du type de déclaration (`public/protected`) dans la classe mère et du type d'héritage (`private/protected`). Compléter le tableau ci-dessous !

Classe de base			**Dérivée publique**		**Dérivée protégée**		**Dérivée privée**	
Statut initial	Accès FMA	Accès util.	Nouv. statut	Accès util.	Nouv. statut	Accès util.	Nouv. statut	Accès util.
public	Oui		public		protégé			
protégé		Non						
privé							privé	Non

— Accès util. : Accès utilisateur
— Nouv. statut : Nouveau statut

EXERCICE 3 :

Pour mieux comprendre le fonctionnement l'héritage, on se propose de suivre le cycle d'existence des objets.

— Modifier les constructeurs et destructeurs afin d'afficher des messages de debug :*Animal créé, Animal mort, Chien créé, Chien mort*
— Juste déclarer un objet de la classe `Chien` dans le `main`. Commenter le résultat obtenu !

*EXERCICE 4 :

— Tester l'exécution du code ci-dessous :

```
{
    Animal *c;
    c = new Chien(4, "rrrrh");
    delete c;
}
```

Quelque chose vous choque ?
— Ajouter `virtual` devant la déclaration du destructeur de `Animal` et réessayez. A quoi sert ce qualificateur ici ?

2 Héritage Multiple

EXERCICE 5 :

Une confusion existe souvent quant à la nature du dauphin. En fait, cet animal appartient à deux branches taxonomiques différentes : les poissons et les cétacés (mammifères marins).

- Reécrire une classe `Animal` avec les déclarations des méthodes *deplacer* et *engendrer* qui n'est possible que lorsqu'il s'agit d'une femelle.
  Puisque toutes les espèces animales ne se déplacent ni n'engendrent de la même manière, comment assurer que les classe filles de chaque espèce décrivent les comportements spécifiques de ces méthodes ? Il est aussi nécessaire d'avoir dans la classe :
  1. un champ chaine de caracteres `nom` et un champ boolean `estFemelle` ainsi que deux champs entiers `x` et `y` pour la position.
  2. des methodes pour accéder à ces champs.
- Créer une classe fille `Poisson` qui hérite publiquement de `Animal` et définit un champ `profondeur` indiquant la profondeur du déplacement.
- Créer une classe fille `Mammifere` qui hérite publiquement de `Animal` et définit un nouveau champ `vitesse` indiquant la vitesse de déplacement.
- Définir une classe `Dauphin` héritant des deux classes filles précédentes. Comment écrire vos instructions pour distinguer entre les membres données dupliquées de la classe `Animal` dans `Dauphin` ? (pensez à l'opérateur de résolution de portée !)
- Nous cherchons à éviter la duplication. Comment peut on utiliser ic le mot clé `virtual` ?

EXERCICE 6 :

Qu'est-ce qu'une fonction virtuelle, une fonction virtuelle pure et quelle est l'implication majeur pour la classe déclarant une fonction du dernier type ?

- Définir une classe `Point` avec des champs de coordonnées x et y et une méthode virtuelle *affiche* qui affiche à l'écran les informations sur l'objet.
- Définir une classe fille `PointColore` qui hérite de `Point` et contient un champ short `couleur` pour définir la couleur du point. Ajouter la couleur à l'affichage.
- Discuter le résultat du code ci-dessous :

```
main()
{
    Point p(3,5); Point *adp = &p;
    PointColore pc(8,6,2); PointColore *adpc = &pc;
    adp->affiche(); adpc->affiche(); //instructions 1
    cout << "----------------------------------------" << endl;
    adp = adpc;
    adp->affiche(); adpc->affiche(); //instructions 2
}
```

*EXERCICE 7 :

On se propose de reprendre l'exercice précédent en utilisant l'héritage multiple. Désormais, un point peut avoir une couleur et posséder une masse.

- Définir une classe `Couleur` avec un champ short `couleur` et une méthode *affiche*.
- Définir une classe `Masse` avec un champ entier `masse` et une methode *affiche*.
- La classe `PointColore` est à modifier pour qu'elle hérite publiquement de `Point` et de `Couleur`. Créer de la même manière une classe `PointMasse` héritant de `Point` et `Masse`.
- On souhaite enfin définir une classe `PointColMas` héritant à la fois de `PointColore` et`PointMasse`. Pourquoi était-il necessaire d'avoir prévu que les classe `PointColore` et `PointMasse` dérivent virtuellement de `Point` ?
- Faites les modifications et constatez qu'il est aussi nécessaire que `Point` dispose d'un constructeur sans argument.
- Ajoutez des messages de debug dans les constructeurs et destructeurs des différentes et executez le programme ci-dessous :

```
main()                                                                          1
{                                                                               2
    PointColore p(3,9,2);                                                       3
    p.affiche();                                                                4
    PointMasse pm(12,25,100);                                                   5
    pm.affiche();                                                               6
    PointColMasse pcm(2,5,10,20);                                               7
    pcm.affiche();                                                              8
    cout << "Have you understood anything to polymorphism??" << endl;           9
}                                                                               10
```