Augustin Giraudier

Exercices de C++ de A à Z :

Introduction :

Le but de l'exercice va être de modéliser un petit système de gestion de stock d'un magasin avec divers articles. Pour cela, nous allons implémenter plusieurs fonctionnalités en revoyant tous les grands thèmes du C++ étape par étape !

I. Classes / Encapsulation :

- Créez une solution Visual studio ainsi qu'un fichier « main.cpp » dans le dossier des fichiers source.
- Créez une fonction main (int ou void comme vous le souhaitez)

Nous allons commencer par créer un système de gestion des articles :

- Créez des fichiers Article.h et Article.cpp respectivement dans les dossiers « fichiers d'entête » et « fichiers source ».
- Dans le .h, créez la classe CArticle.
- Munissez-la de 3 données privées m_fPrix de type float et m_nStock de type int et m_strNom de type string. (Avec les inclusions qui s'imposent)
- Munissez la classe d'un constructeur pour nos trois données membres. (Il sera déclaré dans le .h et son contenu sera dans le .cpp)
- Créez dans le main un objet de type CArticle avec ses 3 données. Le programme doit compiler (sinon vérifier que Article.h est inclus dans le main et dans Article.cpp)
- Créez les getters constants et les setters pour nos trois données.
- Créez une fonction « Achat » sans paramètre permettant de décrémenter le stock d'un. Effectuez une vérification : si le stock est déjà à 0 affichez le dans la console et sortez de la fonction. (Faites les inclusions nécessaires)
- Vérifiez votre fonction en faisant un achat sur l'article du main et en affichant son stock. (avec std ::cout et le getter par exemple)
- Maintenant améliorons notre fonction en faisant en sorte que l'on puisse passer en paramètre le nombre d'objet achetés (attention ce paramètre doit rester facultatif)
- Ajoutez une fonction « Restock » pour ajouter du stock avec un paramètre (sur le même principe)
- Enfin, définissez l'opérateur ami de sortie vers flux « << » afin de lui faire afficher les infos d'un article grâce à l'instruction std ::cout comme cela :

Code :

```
//création d'un article :
CArticle article1 (10, 100, "Tablette de chocolat");

//achat unique (par défaut) :
article1.Achat();

//affichage :
std::cout << article1 << std::endl;
```

Résultat :

```
Nom de l'article : Tablette de chocolat
Prix a l'unite : 10 euro(s)
Stock actuel : 99 article(s)
```

II. Héritage :

Maintenant que vous maitrisez à la perfection les classes, les opérateurs, les fonctions et les données membres, passons au concept de l'héritage. Il va être question d'améliorer la précisons que l'on pourra apporter pour chaque article.

- Tout d'abord, Créons une nouvelle classe « CVehicule » (avec ses 2 fichiers) et faisons-la hériter de la classe CArticle.
- Ajoutons lui 2 données membres privées qui seront le nombre de roue (m_nRoues) et la puissance en cm cube (m_nCylindree).
- Ajoutons les getters, setters et le constructeur qui initialise les deux données membres.
- Rendons maintenant la classe CArticle Virtuelle Pure (par exemple à travers la fonction Achat)
- Tentons de recompiler ! ET là… C'est le drame ! La fonction CArticle n'est plus instanciable !
- Modifiez le constructeur de la classe CVehicule afin qu'il soit capable d'initialiser les données de sa classe mère (le prix etc…)
- Maintenant, supprimons tout le contenu du main et créons un véhicule.
- Hélas CVehicule est également considéré comme abstrait, et on nous empêche de l'instancier. Mais pourquoi cela ? Car la fonction que l'on a choisie pour rendre notre classe mère virtuelle pure doit être redéfinie dans la classe fille afin de spécifier que la classe fille est, elle, non abstraite.

Ainsi, dans CVehicule.h :

```
void Achat(int Decrement = 1) override;
```

(si vous aviez choisi la fonction Achat)

dans CVehicule.cpp :

```
void CVehicule::Achat(int Decrement)
{
    CArticle::Achat(Decrement);
}
```

Le mot clé « override » est facultatif mais il nous assure que nous sommes bien entrain de redéfinir une fonction virtuelle de la classe mère.
Dans le corps de la fonction, on se contente de faire appel à la fonction de la classe mère car on ne souhaite pas réellement modifier son fonctionnement.
Note : Ici nous utilisons les getters pour avoir accès aux données de la classe mère, mais on peut également changer ces dernières pour le niveau de protection « protected » afin de les rendre accessibles dans les classes filles.

- Redéfinissez l'opérateur de sortie vers flux de CVehicule (vous ne pouvez pas rendre virtuel un opérateur ami, (ami = qui n'appartient pas à la classe) vous devez donc simplement créer un nouvel opérateur propre à cette classe fille)
- Ajouter une nouvelle classe COrdinateur héritant de CArticle. Elle aura pour donnée membre un type d'utilisation (m_Utilisation) qui pourra uniquement être « Bureautique » ou « Gaming » (utilisez une énumération) Faites les mêmes étapes que pour la classe CVehicule. Pour l'affichage de l'utilisation, vous pouvez par exemple utiliser un switch.

III. Pointeurs intelligents / Relation de composition :

Maintenant que vous êtes devenu(e) professionnel(le) dans l'art qu'est l'héritage en c++, penchons-nous sur l'utilisation des pointeurs intelligents et voyons comment faire une relation de composition (lorsqu'une de nos classes possède en donnée membre un objet du type d'une autre de nos classes).

- Pour commencer, Créons la classe qui va gérer tout le magasin et afin de coller à notre exemple nous l'appellerons « CMagasin ».
- Ajoutez les bibliothèques <vector> et <memory> à votre fichier Magasin.h afin d'ajouter la possibilité de créer des tableaux dynamiques ainsi que des pointeurs intelligents.
  Ajoutez également l'includion du fichier Article.h puisqu'on s'en servira.
- Ajoutez une donnée membre m_vecpArticles qui sera un vector de pointeurs vers des articles.
  Petite aide : `std::vector<std::shared_ptr<CArticle>> m_vecpArticles;`
- Nous n'ajouterons pas de constructeur puisque celui par défaut nous construira un vector de taille nulle qui nous satisfera parfaitement !
- Ajoutons maintenant un getter permettant de récupérer ce vector et un setteur (AddArticle) prenant en paramètre un pointeur (intelligent) vers un article à ajouter au magasin.

Ici nous pouvons voir un intérêt des pointeurs : nous pouvons faire une liste de pointeurs vers un CArticle alors que la classe est abstraite !
Tous nos articles que ça soient des ordinateurs ou des véhicules pourront être stockés dans une même liste !

- Maintenant, créons dans le main un objet de type CMagasin et ajoutons lui l'ordinateur et le véhicule que nous avions créé ! (Vous devrez utiliser la fonction std ::make_shared) Attention, CArticle étant abstraite, vous devez faire des make_shared avec le nom de la classe fille. Ex : std::make_shared<CVehicule>(vehicule1)
- Nous pouvons ajouter une fonction nous renvoyant le nombre d'articles différents en magasin afin de pouvoir tester si nos ajouts ont fonctionné.

Afin de nous familiariser avec la gestion de cette liste, créons maintenant 3 fonctions qui pourraient être utiles pour la gestion du magasin.

- Créons la fonction AfficherNbrTotalStock() qui fait un affichage du nombre d'article en prenant en compte le stock de chaque article. Cela devra ressembler à :

```
Nombre total des stocks : 12
```

- Créons la fonction AfficherPatrimoine() qui affichera le montant total que représentent les articles en magasin.

```
Patrimoine : 1200 euros
```

- Créons enfin la fonction Liquidation() qui diminue de moitié les prix de chaque article du magasin (les valeurs de toutes les données membre m_fprix doivent être changées). Testez son implémentation avec le fonction patrimoine, il doit être divisé par 2.

```
Patrimoine : 4200 euros

------------------------------------------------------------
------------------------------------------------------------
----------------------LIQUIDATION---------------------------
------------------------------------------------------------
------------------------------------------------------------

Patrimoine : 2100 euros
```

IV. Transtypage / Polymorphisme :

Si vous en êtes arrivés à là, c'est que vous êtes particulièrement persévérant ! Tout d'abord félicitation !
Plus tôt, nous avons réussi à mettre nos véhicules et ordinateurs dans une liste d'articles, cela était en réalité une première étape de polymorphisme…
Mais il s'agissait de transtypage ascendant qui était relativement automatique.
Maintenant il va s'agir de retransformer les articles en leur classe fille afin d'avoir accès à leur données membres plus spécifiques.
Avant de commencer : Petit Rappel :

- `std::static_pointer_cast<NouveauType>(pointeur)`
  ➔ Cette fonction permet de convertir de façon descendante un pointeur.
  Mais la conversion aura lieu dans tous les cas !! et si l'on tente ensuite d'accéder à une fonction membre du nouveau type et que l'objet pointé n'est pas réellement de ce type, le programme plantera.
  Néanmoins, elle consomme moins de ressources que sa concurrente.

- `std::dynamic_pointer_cast<NouveauType>(pointeur)`
  ➔ Cette fonction permet également de convertir de façon descendante un pointeur.
  Mais si le pointeur en paramètre n'a pas le même type que celui de la conversion : cette fonction reverra « nullptr ». Ainsi, impossible de faire planter le programme et l'on peut vérifier de cette manière si un pointeur appartient à une classe fille en particulier.
  Néanmoins, cette fonction consomme plus de ressources que sa concurrente.

(Pensez à retirer du main ou à commenter toutes les fonctions d'affichage présentes afin de ne pas polluer vos futur tests)

- Pour commencer, écrivons une fonction AfficherNbrOrdinateurs() qui comptera parmi les articles, ceux qui sont de classe fille COrdinateur. Utilisez la fonction de transtypage qui vous parait être la plus adaptée.
- Écrivons ensuite une fonction plus compliquée mais faisable avec les notions des questions précédentes. Créons la fonction TriArticles() qui renvoi un tableau de type « std::array » contenant 2 std::vector<std::shared_ptr<CArticle>> contenant respectivement tous les ordinateurs du magasin et tous les véhicules du magasin. (Pensez à inclure <array>)

Précisions pour la question suivante : le mot clé « static » devant la déclaration d'une fonction membre d'une classe permet de l'utiliser de cette Manière : CMaClasse::MaFonction(MonParametre) ; l'utilité est de pouvoir l'utiliser sans instance de la classe (pas besoin de faire objet.MaFonction(MonParametre) si l'on ne possède pas l'objet en question)

- Pour terminer, écrivons deux fonctions static AfficherTableauVehicules() et AfficherTableauOrdinateurs() prenants en paramètre un std::vector<std::shared_ptr<CArticle>> et faisant l'affichage complet (nombre de roues pour les véhicules et utilisation pour les ordinateurs) via l'opérateur de sortie vers flux qui avait été définit. Réfléchissez bien à quelle fonction de conversion sera la plus optimisée ! 😉
  (N'hésitez pas à commenter la fonction d'affichage créée précédemment afin de repartir avec une console propre !)

Félicitation !! vous avez réussi tous les exercices ! Les grands principes du C++ n'ont presque plus de secret pour vous ! Vous pouvez maintenant faire une pause et sortir afin de retrouver les sensations du contact avec d'autres humains !