ISIMA 3ème année - MODL/C++
TP 4 : Mouvements et *smart pointers*

Exercice 1 - Opérations de mouvement

On propose d'illustrer le mécanisme de « mouvement » introduit en C++11 avec la modélisation d'un vecteur de nombres complexes (code fourni) : la classe `Vecteur` contient des objets du type `complexe_t`. Les complexes sont manipulés à travers le type `complexe_t`, et non pas directement par la classe `Complexe`, car on souhaite tracer les constructions et les copies d'objets de cette classe. Pour cela on manipule des objets de la classe `Mouchard<complexe_t>` *a.k.a.* `complexe_t`. Après avoir terminé ce TP, vous pourrez si vous le souhaitez vous pencher sur le contenu de la classe `Mouchard`.

1) Etudier dans un premier temps le code du fichier `test_mouchard.cpp`, et dans un second temps, celui du fichier `test_vecteur.cpp`, et les résultats (décomptes de certaines opérations) qu'ils produisent dans le terminal, afin de bien comprendre d'où viennent toutes les copies et les mouvements d'objets `Complexe`.

   Utiliser les « autotests » (d'abord `autotest_mouchard`, puis `autotest_vecteur1`) fournis pour vous assurer d'avoir bien identifié toutes les opérations. **Conservez vos réponses, elles seront utiles à la question 3.**

   Attention : une même opération peut être comptabilisée plusieurs fois ; par exemple, si le constructeur de copie est appelé, l'opération est comptabilisée comme "construction" et comme "copie".

2) On constate que l'utilisation des opérateurs `+` et `*` sur les vecteurs créent de nombreuses copies. Pour limiter ces dernières, implémenter un opérateur d'affectation par mouvement pour la classe `Vecteur`.

   Implémenter aussi le constructeur de mouvement, même s'il n'est pas utilisé dans notre exemple.

3) Etudier à nouveau les résultats fournis par le fichier `test_vecteur.cpp` pour voir l'impact des opérateurs de mouvement. Utiliser le troisième autotest (`autotest_vecteur2`) fourni pour vous assurer d'avoir bien compris la notion de mouvement. Comparez vos réponses à celles de la question 1.

Exercice 2 - *Smart pointers*, le pointeur unique

Nous proposons ici d'illustrer la manipulation de pointeurs uniques (`std::unique_ptr`) avec la création d'un jeu de 52 cartes. L'idée repose en partie sur les *design patterns «abstract factory»* et *«singleton»* (dont nous parlerons plus tard dans le cours), elle consiste à déléguer la création d'objets à un tiers, ce dernier contrôlant ainsi la production des objets (quantité et contenu). Nous allons donc concevoir une classe `Carte` pour laquelle il sera impossible d'appeler le constructeur (ni l'opérateur d'affectation), à l'exception de la classe `UsineCarte` qui se chargera de produire exactement 52 cartes (toutes avec une valeur différente). Nous choisissons de fournir les cartes sous la forme de pointeurs uniques, l'idée étant de montrer que ce mécanisme permet bien d'obtenir une certaine sécurité face à la fuite de mémoire.

1) Créer la classe `Carte` avec un simple attribut qui contiendra la valeur de la carte (l'usine s'assurera que cette valeur est entre 0 et 51). Créer la classe `UsineCarte` avec une méthode `getCarte` chargée de créer une carte et de retourner son pointeur (unique, utiliser `std::unique_ptr`). Utiliser un attribut dans la classe `UsineCarte` pour compter les cartes et fournir une valeur différente à chacune. Une fois les 52 cartes fournies, la méthode `getCarte` doit retourner un pointeur nul. Test 1

2) Interdire l'utilisation du constructeur de copie et de l'opérateur d'affectation pour la classe `Carte`. Son constructeur *ad hoc* qui recevra la valeur de la carte en argument doit être privé. Pour accéder au constructeur privé, la classe `UsineCarte` doit être amie. Test 2

   Empêcher également toute copie (par construction ou affectation) d'objets `UsineCarte`. Test 3

3) Adapter la classe `UsineCarte` pour qu'elle puisse produire un nombre de cartes fixé à la construction de l'objet usine, et non plus spécifiquement 52 cartes. Test 4

4) Nous allons maintenant mettre toutes les cartes produites par une usine dans un « paquet », un vecteur de pointeurs uniques de cartes. Définir l'alias de type `paquet_t` pour ce vecteur (utiliser le mot-clé `using`).

   Ecrire la fonction `remplir` qui prend en arguments un paquet et une usine, et remplit le paquet avec toutes les cartes produites par l'usine. Vous utiliserez le fait que `getCarte` renvoie un pointeur nul pour stopper le remplissage du paquet. Test 5

5) Définir l'opérateur `<<` pour écrire le contenu d'un paquet dans un flux de type `std::ostream`, les valeurs seront simplement séparées par un espace. Test 6

   Consignes : utiliser les itérateurs pour parcourir le paquet dans l'opérateur de flux.

   Subsidiaire : utiliser la syntaxe simplifiée de la boucle `for` vue en cours.

6) Pour vous assurer qu'il n'y a pas de fuite mémoire, utiliser Valgrind. Ajouter aussi un compteur qui augmente à chaque construction de carte et décroît à chaque destruction de carte, afin de s'assurer là encore du bon fonctionnement des *smart pointers*. Pour cela, ajouter un attribut de classe à `Carte` pour le compteur et une méthode de classe `getCompteur` pour consulter la valeur du compteur. Test 7

Exercice 3 - *Smart pointers*, le pointeur partagé

Nous proposons ici d'illustrer la manipulation de pointeurs partagés (`std::shared_ptr`) avec un modèle de consommation de ressources : des consommateurs partagent à plusieurs des ressources (qui seront matérialisées par des objets `Consommateur` partageant un pointeur sur un objet `Ressource`) et puisent dans ces ressources jusqu'à épuisement de leur stock (chaque ressource a sa propre quantité et chaque consommateur son propre besoin en ressource). Lorsqu'un consommateur s'aperçoit qu'une ressource est épuisée, il s'en sépare (le pointeur sur la ressource est remplacé par le pointeur nul dans l'objet `Consommateur`).

1) Créer la classe `Ressource` avec un simple attribut qui contiendra le stock de la ressource. Elle est initialisée via un argument au moment de la construction. Ensuite, une méthode `consommer` avec en argument une quantité vient décroître le stock. Un accesseur permet de consulter le niveau du stock. Test 8

2) Créer la classe `Consommateur` avec deux attributs, l'un est un pointeur partagé sur la ressource exploitée, et l'autre contient son besoin en ressource. Ces attributs sont initialisés via des arguments au moment de la construction. Ensuite, une méthode `puiser` sans argument vient consommer dans la ressource la quantité dont le consommateur a besoin. Test 9

   Si le stock est épuisé, le pointeur sur la ressource est remplacé par le pointeur nul. Test 10

3) Afin de surveiller l'état des ressources, celles-ci vont être stockées dans un vecteur, plus exactement leurs pointeurs, et pour ne pas intervenir sur le partage de propriété des pointeurs, il s'agira d'utiliser des pointeurs faibles (`std::weak_ptr`). Définir l'alias de type `ressources_t` pour ce vecteur.

   Ecrire l'opérateur `<<` afin d'écrire le stock de chaque ressource dans un `std::ostream` (l'opérateur prend en argument un vecteur de ressources de type `ressources_t`). Attention, utiliser les fonctionnalités du pointeur faible pour vérifier que le pointeur est encore valide (et donc que la ressource associée existe encore) avant d'accéder à son stock. Si le pointeur n'est plus valide, écrire le caractère `-` à la place du nombre. Test 11

   Explication du test : l'idée est que si une ressource est épuisée, les consommateurs associés vont se séparer l'un après l'autre de la ressource (et donc ne plus pointer sur la mémoire associée) jusqu'à n'avoir plus aucun consommateur lié à la ressource (et donc aucun pointeur sur la mémoire associée, ce qui va déclencher sa libération, cf. fonctionnement des `shared_ptr`).