Université de Paris
Master 1

Langages à Objets Avancés
Année 2022—2023

TP n°4

Références et pointeurs

Exercice 1 [Références/pointeurs/valeurs]

1. Ecrivez rapidement les deux fichiers `.hpp` et `.cpp` d'une classe `BoxInt` qui encapsule un entier. Cette classe aura un attribut de type int, et deux méthodes : un "getter" get() et un "setter" set(int). Surchargez l'opérateur << pour l'affichage. Vous définirez également un constructeur qui prendra en argument un int ainsi qu'un destructeur trivial qui vous affiche un message.
2. Créez un fichier de test qui définit trois fonctions

```
void fonction1(BoxInt t) {
    t.set(36);
}

void fonction2(BoxInt *t) {
    t->set(666);
}

void fonction3(BoxInt &t) {
    t.set(1);
}
```

3. Essayez d'anticiper le comportement de la séquence suivante, en vous assurant de comprendre les symboles utilisés. Distinguez en particulier les usages de &.

```
BoxInt monTest{42};
std::cout << monTest;

monTest.set(0);
std::cout << monTest;

fonction1(monTest);
std::cout << monTest;

fonction2(&monTest);
std::cout << monTest;

fonction3(monTest);
std::cout << monTest;
```

   Vérifiez ensuite en exécutant ces instructions dans le main() de votre fichier test.
4. Avec votre définition de BoxInt, est-il possible de définir la fonction ci-dessous dans le fichier test ?

```
void fonction4(const BoxInt &t) {
 t.set(13);
}
```

Vérifiez votre réponse en compilant/exécutant.

5. même question avec

```
void fonction5(const BoxInt *t) {
 t -> set(13);
}
```

Vérifiez votre réponse en compilant/exécutant.

6. On veut pouvoir connaître le nombre d'instances existantes de BoxInt à tout moment. Ajoutez un attribut statique **int** à votre classe et une méthode statique alive_count() qui renvoie la valeur de cet entier. Adaptez le code du constructeur et du destructeur de sorte que l'on ait le comportement voulu. Testez votre comptage en créant et supprimant des objets de la classe BoxInt avec **new** et **delete** et en affichant ce que renvoie alive_count() entre ces opérations dans le fichier test.

7. Écrivez la fonction

```
void un_test(){
 BoxInt un_int{42};
 BoxInt un_autre_int {un_int};
}
```

dans votre fichier test et exécutez-la dans votre main affichez ensuite la valeur renvoyée par alive_count() avant et après. Vérifiez qu'on obtient bien la valeur attendue.

8. Ajoutez à la fonction un_test :

```
BoxInt *n = new BoxInt{54};
```

Affichez la valeur renvoyée par alive_count() après. Que remarquez-vous ? Pourquoi ?

Exercice 2 [vector] Dans cette exercice, nous allons donner une implémentation alternative de la classe vector de la STL qui représente des tableaux. Comme les templates n'ont pas encore été vus en cours, nous allons nous focaliser sur des vecteurs d'entiers.

1. Créez les deux fichiers `.hpp` et `.cpp` associés à une classe Vector. Cette classe contiendra un **int** qui représentera la taille courante du tableau et un pointeur **int*** vers un tableau d'entiers. Faites en sorte qu'un utilisateur de Vector ne puisse pas changer ces attributs directement. Créez un constructeur, un destructeur et et redéfinissez l'opérateur << qui affiche en premier la taille, puis les entiers du tableau en les séparant par des virgules. On rappelle que l'on crée et supprime des tableaux d'entiers avec les opérations pointeur = **new int**[taille] et **delete**[] pointeur.
2. Écrivez des méthodes get_at(**int**) et set_at(**int**,**int**) qui respectivement lisent et écrivent dans une case d'un Vector.
3. Écrivez une méthode push_back(**int**) à votre classe, qui ajoute un entier à la fin du tableau de Vector. Votre méthode devra créer un nouveau tableau **int***, recopier l'ancien dans le nouveau, et supprimer l'ancien. Similairement, écrivez une méthode push_front(**int**) qui ajoute un entier au début du tableau.

4. Écrivez des méthodes pop_back() et pop_front() qui suppriment et renvoient respectivement le dernier et le premier élément du tableau.
5. Concevez une procédure qui permet de tester toutes les méthodes de Vector définies jusqu'à présent, et écrivez-la dans un fichier `.cpp` de test. On pourra comparez le comportement de Vector avec celui de vector<int> en répliquant les opérations de la procédure de test sur une instance de cette dernière classe et en comparant les tableaux obtenus. On pourra utiliser srand() et rand() pour générer des tableaux aléatoires.
6. Copier des Vector peut être très coûteux. Comment s'assurer simplement en C++ qu'aucune copie de Vector n'a lieu lors de l'exécution ?
7. On souhaite pouvoir connaître la mémoire occupée par l'ensemble des objets Vector à tout moment. Pour cela, ajoutez une variable statique représentant l'espace occupé à la classe et adaptez les méthodes que vous avez déjà écrites.