

TP n° 4

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 trois méthodes : un "getter" `get()`, un "setter" `set(int)`, et une méthode d'affichage `print()`. Aussi, on définira un constructeur qui prendra en argument un `int` et un destructeur trivial.

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);
monTest.print();

monTest.set(0);
monTest.print();

fonction1(monTest);
monTest.print();

fonction2(&monTest);
monTest.print();

fonction3(monTest);
monTest.print();
```

Vérifiez votre compréhension en exécutant la séquence ci-dessus 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(1);
}
```

Vérifiez votre réponse en essayant de compiler avec cette nouvelle fonction.

- 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.
- Écrivez la fonction

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

dans votre fichier test et exécutez-la en affichant la valeur renvoyée par `alive_count()` avant et après. Que remarquez-vous? Comment corriger le problème?

- Peut-on définir une méthode statique de `BoxInt` qui utiliserait `get` ou `set` dans sa définition? Vérifiez votre réponse en essayant de compiler avec une telle méthode.
- On veut pouvoir utiliser `cout <<` avec `BoxInt` comme avec `int`. Déclarez et définissez une fonction (pas une méthode!)

```
ostream& operator<<(ostream& out, BoxInt& mon_int);
```

qui affiche la valeur de l'entier contenu dans `mon_int` (Indication : pour cela, on écrit la procédure que l'on ferait avec `cout` en remplaçant `cout` par `out`). Testez votre code avec

```
BoxInt mon_test(42);
cout << mon_test;
```

en vérifiant que 42 s'affiche.

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 `vectors` d'entiers.

- 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 une fonction d'affichage : cette dernière 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 comparer 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'assure-t-on simplement en C++ qu'aucune copie 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.

Liste doublement chaînée

[Si vous avez le temps ou à faire chez vous]

Nous allons maintenant donner une implémentation des listes doublement chaînées. Pour mémoire, une liste consiste essentiellement en une collection de cellules contenant chacune trois champs : son contenu, un pointeur vers la cellule précédente et un pointeur vers la cellule suivante. Ces pointeurs sont `nullptr` en cas d'absence de précédent ou de suivant.¹

Ces champs seront évidemment encapsulés et cachés au monde extérieur, qui n'accède à la liste qu'au travers d'un certain jeu de méthodes garantissant que la liste préserve une structure cohérente.

Comme dans l'exercice précédent, on se focalise sur les listes chaînées contenant des entiers.

Exercice 3 [Cellule]

1. Écrire la classe `Cell`.

Cette classe contient, outre les 3 champs déjà mentionnés, un constructeur adéquat, une méthode `connect` permettant de connecter deux cellules

¹Au fait, pourquoi faut-il utiliser des pointeurs et non des références ?

(pensez à modifier le champs `next` de l'une et `previous` de l'autre) et les méthodes `disconnect_next` et `disconnect_previous` (idem : pensez à mettre à jour l'ancienne cellule voisine).

2. Si on veut faire jouer un rôle symétrique aux deux cellules que l'on connecte, en permettant un appel de la forme `Cell::connect(c1, c2)` (au lieu de `c1.connect(c2)`), quelle sera la déclaration correcte de cette méthode?
3. Faites en sorte que le monde extérieur ne puisse pas modifier des cellules de façon incohérente (notamment, pour toute cellule `c`, il faut que la cellule précédente de la suivante de `c` soit toujours `c`). Pour cela, jouez sur les modificateurs de visibilité (`private`) et ajoutez des accesseurs en lecture seule s'il le faut.

Exercice 4 [Liste]

On écrit maintenant la classe `List` qui, en s'appuyant sur la classe `Cell` de l'exercice précédent, fournit les méthodes usuelles d'accès à une liste :

- `int length()` : longueur de la liste ;
- `int get(int idx)` : valeur du `idx`-ième élément de la liste ;
- `int find(int val)` : indice de la valeur `val` si elle existe dans la liste, `-1` sinon ;
- `void set(int idx, int val)` : affecte la valeur `val` à la position `idx` de la liste ;
- `void insert(int idx, int val)` : insère la valeur `val` en position `idx` (et décale les éléments qui suivent) ;
- `void delete(int idx)` : supprime la valeur d'indice `idx` (et décale les éléments qui suivent).

1. Écrivez la classe `List`, munie de champs privés pointant la première et la dernière de ses cellules (`nullptr` si liste vide), d'un constructeur instanciant une liste vide, un destructeur qui désalloue les cellules de la liste et des méthodes mentionnées ci-dessus.
2. Ajustez l'encapsulation de la classe `Cell`, afin que seule la classe `List` puisse instancier et manipuler des cellules (qui ne sont qu'un intermédiaire technique pour implémenter une liste chaînée et n'ont pas vocation à être visibles pour les autres classes).

Indice : il faudra utiliser `private` et `friend`.

3. Testez toutes les méthodes ! Comment peut-on faire pour tester les valeurs des champs et méthodes privés, et malgré tout regrouper tous les tests dans une classe séparée ?