Université Paris Diderot **Langages à objets avancés**
M1 **Année 2019-2020**

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, ainsi qu'un fichier de test. Votre classe contiendra un constructeur, un setter, ainsi qu'une méthode d'affichage.

2. Ajoutez une méthode publique `void setMonAttribut(int a)` , en séparant toujours déclaration et implémentation.

3. Insérez le code suivant dans votre fichier de test :

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

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

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

4. Essayez d'anticiper le résultat de la séquence suivante, en vous assurant d'assimiler les symboles utilisés. Distinguez en particulier les usages de `&`.

```
BoxInt monTest(42);
monTest.affiche();

monTest.setMonAttribut(0);
monTest.affiche();

fonction1(monTest);
monTest.affiche();

fonction2(&monTest);
monTest.affiche();

fonction3(monTest);
monTest.affiche();
```

5. Remplacez la `fonction3` par :

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

   Quel est le changement ?

Exercice 2 [Modélisation] Nous allons modéliser de manière grossière la gestion d'un aéroport, des avions et des passagers. Vous pouvez utiliser Bouml pour votre représentation, l'implémentation est l'objet de l'exercice suivant.

1. Un **Aéroport**, identifié par son nom, connait la liste des **Avion**s, également identifiés par des noms, présents sur la piste.

2. Chaque Avion contient 5 Sièges identifiés par leur numéro. Ajoutez à votre diagramme la classe **Siege** et précisez ses relations.

3. Chaque Siège peut-être réservé, ou non, par une Personne identifiée par son nom. Ajoutez à votre diagramme la classe **Personne** et précisez ses relations.

Exercice 3 Implémenter précisément à la main le système que vous avez modélisé dans l'exercice précédent. Soyez très précis vis-à-vis de la gestion du cycle de vie des différents objets : par exemple les sièges d'un avion doivent êtres créés à la construction d'un avion et détruits à sa destruction. Comme dans le cours vous pouvez ajouter des affichages pour témoigner des constructions et des destructions.

Pour gérer les ensembles/listes vous utiliserez `#include <vector>` vous pouvez consulter la documentation officielle ou des exemples

Procédez **pas à pas** en testant au fur et à mesure. Conservez vos tests. Assurez vous en particulier de ne pas créer d'objets inutiles.

1. Commencez par implémenter les classes `Aeroport` et `Avion`.

2. Ajoutez à l'aéroport une méthode `void atterrissage(Avions& a)` permettant d'ajouter un avion (déjà existant) à un aéroport. Testez la !

3. La copie d'objets peut-être une opération coûteuse. Comment contrôler qu'aucune copie n'a lieu lors de l'exécution de votre code ?

4. On aimerait que le nom des avions soit automatiquement généré lors de leur création : `Avion#1`, `Avion#2`, `Avion#3` etc. Comment pouvons nous procéder pour maintenir le compte des instances d'`Avion` ? Implémentez votre solution.

5. Implémentez les classes `Siege` et `Personne`, ainsi que des méthodes de votre choix pour permettre à des Personnes de réserver un siège.

Liste doublement chaînée

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

Dans l'exercice précédent nous avons utilisé la librairie standard avec `vector`. Nous pouvons également nous entrainer à implémenter la structure que vous connaissez bien, celle 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.

Dans l'exercice, on implémente une liste chaînée contenant des nombres entiers.

Exercice 4 [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 (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 5 [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 ;

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

- `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 (`null` 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 ?