

## TP n° 5 : static, public, private et friend

### Liste doublement chaînée (repris du TP 4)

Une liste doublement chaînée 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.<sup>1</sup>

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 1** [Cellule] *Pensez à vérifier régulièrement que vous êtes en mesure de compiler (au moins séparément) votre code. Ne laissez pas les erreurs s'accumuler !*

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 2 [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 del(int idx)` : supprime la valeur d'indice `idx` (et décale les éléments qui suivent).

1. Écrivez la classe `List`, munie
  - de champs privés pointant sur la première et la dernière de ses cellules (`nullptr` si liste vide),
  - d'un constructeur instanciant une liste vide,
  - d'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?

**Exercice 3** [Liste de vecteurs] Écrivez les classes `VectorCell` et `VectorList`. Les instances de `VectorList` représentent des listes de vecteurs en chaînant des instances de `VectorCell`. Ces dernières comportent un champ de type `std::vector<int>`<sup>2</sup>.

Ces classes sont quasiment identiques aux classes `Cell` et `List` que vous venez de programmer, mais le type `int` est remplacé par `std::vector<int>`.

Faut-il modifier les destructeurs pour prendre en compte le changement de nature du contenu de la liste (un simple `int` ayant été remplacé par un objet complexe...)? Si oui, faites-le. Dans tous les cas expliquez pourquoi et comment<sup>3</sup> toute la mémoire qu'il faut libérer est libérée à la destruction de la liste.

## Voici venu le temps des élections

**Exercice 4** On souhaite modéliser un scrutin pour des élections. Pour un scrutin donné, on gère plusieurs bureaux de vote (avec dans chacun une urne). Le nombre d'options de vote possibles change à chaque scrutin : par exemple, pour un référendum, il y a 3 options "oui", "non" et "vote nul ou blanc".

On va avoir une classe `Scrutin` qui contiendra le nombre de bureaux de Vote (et donc d'urnes), le nombre d'options de vote et un tableau de pointeurs sur les urnes. On fera aussi une classe `Urne` qui contiendra une référence sur `Scrutin`, un entier représentant le numéro du bureau de vote (utilisez un

---

2. En "vrai" on programmerait un *template* de liste, capable de travailler avec du contenu dont le type est paramétré. Nous verrons cela plus tard.

3. Si vous avez besoin de détails sur le fonctionnement de `std::vector`, de ses constructeurs et destructeurs, lisez la documentation : <http://www.cplusplus.com/reference/vector/>.

compteur « static ») et un tableau d'entier comptabilisant les votes pour chaque option.

De plus, `Urne` aura une méthode `bool voter(int choix)`, qui retournera `false` si l'option est impossible, et vous ajouterez les méthodes nécessaires pour pouvoir obtenir les résultats d'un bureau de vote, de celui du scrutin entier et d'afficher ces résultats.

**Indication :** Vous avez dû remarquer qu'une urne contient une référence à un scrutin qui lui contient un tableau d'urnes. Si vous essayez de mettre `#include "Scrutin.hpp"` dans le fichier `Urne.hpp`, et vice-versa, le compilateur refusera.

Pour résoudre le problème, dans le fichier `Urne.hpp`, on déclare `class Scrutin`; avant la déclaration de la classe `Urne` et on fait un `#include "Urne.hpp"` dans `Scrutin.hpp`. La déclaration `class Scrutin`; suffit car, dans la déclaration de la classe `Urne`, on n'utilise pas d'autre information que le fait que cette classe existe.

**Exercice 5** Écrire les destructeurs des classes `Urne` et `Scrutin`. À la fin d'un scrutin, on détruit les urnes (dans la réalité leur contenu). Vérifiez avec des sorties écran appropriées que l'on détruit bien les urnes.

**Exercice 6** Pour éviter qu'on puisse fabriquer des urnes et les rattacher à un scrutin indûment. On va rendre les constructeurs et destructeurs d'`Urne` privées. Pour que `Scrutin` puisse construire des Urnes et les détruire, on va déclarer la classe `Scrutin` amie de `Urne` en écrivant dans la déclaration de classe de `Urne` : `friend class Scrutin`; Cette déclaration d'amitié va permettre à `Scrutin` d'utiliser les membres de `Urne` qui ne sont pas publiques.

**Exercice 7** Ajoutez des `const` partout où c'est possible : attributs, méthodes, ...

## Pour aller plus loin

**Exercice 8** Comment éviter que l'on puisse voter après la fin du scrutin et que l'on puisse afficher les résultats avant la fin du scrutin ?

**Exercice 9** Pour l'instant, on ne contrôle pas qui vote et combien de fois. Proposez une modélisation plus complexe qui résolve le problème.