Université de Paris

Langages à Objets Avancés
Année Master 1

TP n°4

Pas de grandes nouveautés pour ce TP, profitez-en pour gagner en confiance en programmation c++. Coder des listes chaînées est un exercice très classique que vous avez déjà certainement fait dans d'autres langages, essayez de faire le travail proprement et testez régulièrement votre travail sur des exemples.

Listes doublement chaînées

On rappelle qu'on implémente les listes en utilisant deux classes : les cellules, et une encapsulation de cellule (ce qui permet de définir la liste vide en tant qu'objet qui encapsule `nullptr`)

Lorsque la liste est dite doublement chaînée, les cellules sont composées de trois champs : son contenu, un pointeur vers la cellule précédente et un pointeur vers la cellule suivante. Ces pointeurs valent `nullptr` en cas d'absence de précédent ou de suivant. [1]

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

On se focalisera dans ce TP sur les listes chaînées d'entiers.

Exercice 1 [Cellule]

1. Écrire la classe `Cell`.

   Cette classe contient, outre les 3 champs déjà mentionnés, un constructeur adéquat, les méthodes `disconnect_next` et `disconnect_previous` et `connect` permettant de se séparer des cellules ou d'en connecter. Pensez à :
   - mettre à jour l'ancienne cellule voisine
   - ne vous préoccupez pas des destructions : les cellules qui peuvent rester pendantes lors de ces opérations devrons être gérées par le programmeur, on ne peut pas savoir si son intention est de les réarranger ou de les supprimer
   - définir une signature pour `connect` qui garantit que la cellule à connecter à la suite de la cellule courante existe forcément.
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)`, quelle sera la déclaration correcte de cette méthode ? Ecrivez là.
3. Faites en sorte que le monde extérieur (sauf la classe Liste définie plus tard) ne puisse pas agir du tout sur les cellules.

Exercice 2 [Liste]

Notre classe `List` qui doit fournir les méthodes usuelles :
- `int length()` : longueur de la liste ;
- `int get(int idx)` : valeur du `idx`-ième élément de la liste ;

1. Nous espérons que vous comprenez pourquoi on utilise des pointeurs et non des références ou des cellules ... sinon discutez-en avec votre enseignant

— `int find(int val)` : indice de la valeur `val` si elle est présente 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 conserve les éléments qui suivent).

1. Écrivez la classe `List`, munie de champs privés pointant sur la première et la dernière de ses cellules, d'un constructeur instanciant une liste vide, un destructeur qui désalloue les cellules de la liste, l'opérateur d'affichage [2], et les méthodes mentionnées ci-dessus (au moins la moitié).
2. Réfléchissez à ce que serait un constructeur de copie sur les listes.
3. Pourquoi le code suivant est dangereux ?

```
List l1;
l1.insert(0,10);
{
        List l2;
        ....
        l2=l1;
}
cout << l1;
```

4. Si vous avez le temps, développez un exemple plus significatif (le tri bulle d'une liste, la fusion de deux listes triées ...)

2. vous constaterez que l'amis d'un amis n'est pas un amis ...