
TD 8

Variations sur les listes, les files et les piles

8.1 Objectif

Jusqu'à présent, nous avons introduit l'héritage comme une manière de réaliser la relation *est-un*: un `Item_Paragraph` *est un* `Paragraph`, il en exhibe les propriétés, il en possède l'interface, il peut lui être *substitué* dans tout contexte. Pour ces raisons, cette relation est aussi appelée *sous-typage* : `Item_Paragraph` est un sous-type de `Paragraph`. Elle est réalisée en C++ par l'héritage public, qui valide la conversion d'héritage dans tout contexte.

L'héritage public de C++ consiste donc essentiellement en un *héritage de l'interface* de la classe. Mais, bien entendu, il s'accompagne aussi de l'héritage d'une partie du code de la classe de base : les membres de données en sont hérités et aussi le code de toutes les fonctions-membres.

Cet exercice introduit une nouvelle manière d'utiliser l'héritage afin d'assurer le partage (ou la réutilisation) de code sans hériter de l'interface. Cette technique est particulièrement utile dans la conception de bibliothèques, où l'accent est essentiellement placé sur la réutilisation et l'évolutivité.

Au passage, cet exercice permet aussi de revoir les classes génériques, et d'illustrer leur comportement en cas d'héritage (ce comportement ne présente d'ailleurs aucun mystère !). Il permet aussi, vers la fin, un retour sur le mécanisme de fonction virtuelle.

8.2 Une simple classe liste générique

On trouvera, dans le répertoire `List_Queue_Stack`, la réalisation d'une classe générique `List<ELEM>` (`ELEM` est le type des éléments). En fait c'est exactement celle que vous avez du réaliser dans le TD 4 (plus précisément en 4.3.2). Le répertoire comporte quatre fichiers : `List.h`, la définition de la classe générique `List<ELEM>` ; `List.cpp`, la définition du corps de ses fonctions-membres et amies ; `main_List.cpp`, un programme de test ; et une `Makefile` (que vous aurez à compléter pour la suite de cet exercice). Notez que le fichier `List.cpp` est inclus à la fin de `List.h` et non pas compilé séparément (la `Makefile` ne génère pas de `List.o`). Rappelez-vous que c'est ainsi qu'il convient de faire dans le cas des classes *template* (voir 1.3).

La liste proposée a la structure décrite à la figure 8.1. La classe `List<ELEM>` est composée simplement de deux pointeurs, `head` et `tail`, qui pointent respectivement sur le début et la fin d'une liste simplement chaînée de cellules (de type `Cell`, ou plus précisément `List<ELEM>::Cell`, car il s'agit d'une structure privée interne à `List<ELEM>`). Chaque cellule est elle-même composée d'une *valeur* de type `ELEM` (champ `val`) et d'un pointeur sur la cellule suivante (`next`). Le champ `next` de la dernière cellule de la liste est le pointeur nul.

La liste est dotée d'un constructeur par défaut (construisant la liste vide), d'un destructeur, et des opérations de copie. Elle dispose en outre des cinq fonctions-membres suivantes :

- `is_empty()` retourne un booléen indiquant si la liste est vide ;
- `append(ELEM)` ajoute un élément (et donc en particulier une cellule) à la fin de la liste ;
- `prepend(ELEM)` ajoute un élément (et donc en particulier une cellule) au début de la liste ;
- `insert(ELEM)` insère un élément devant le premier élément de la liste qui lui est supérieur ou égal (ou à la fin, s'il n'existe pas de tel élément) ; cette opération suppose que

le type `ELEM` dispose des opérateurs de comparaison et en particulier d'`operator<` ; si on construit la liste uniquement à partir d'appels de `insert()`, la liste est donc toujours ordonnée ;

- `get_first()` retourne la valeur du premier élément de la liste et *retire (et détruit) la cellule correspondante* ; `get_first()` lance l'exception `List<ELEM>::Empty` en cas de liste vide.

La classe dispose aussi d'une fonction amie d'affichage sur `ostream` (`operator<<`).

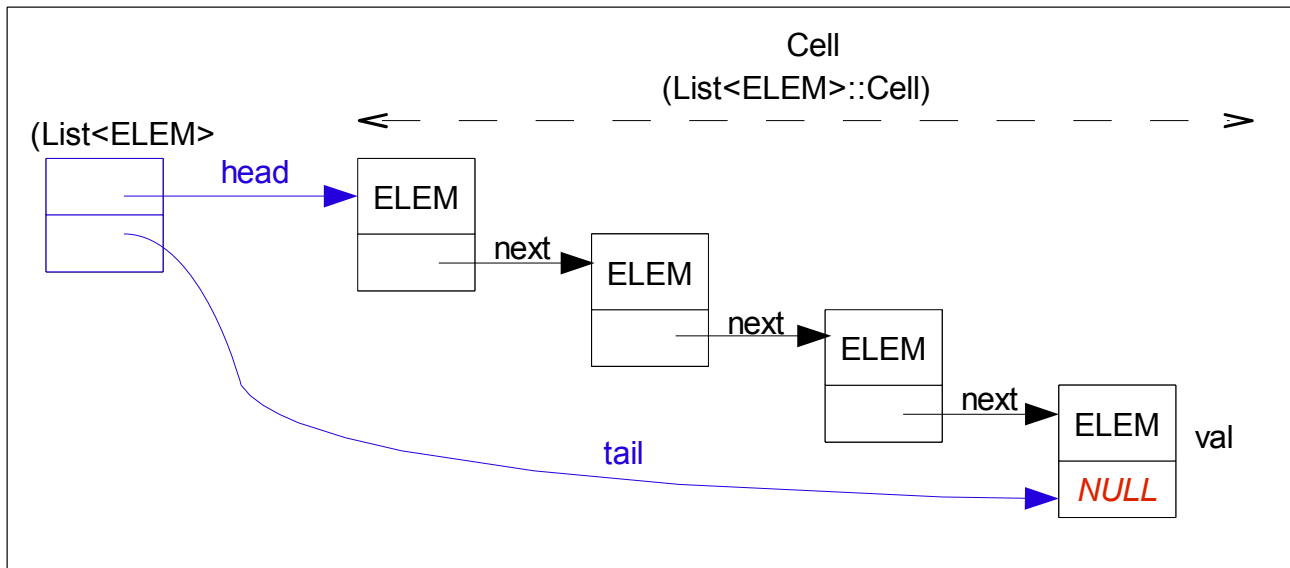


FIGURE 8.1 : Structure de `List<ELEM>`

8.3 Les listes comme implémentation des files et des piles

Votre mission est d'utiliser cette classe `List<ELEM>` afin de réaliser de manière économique des collections bien connues, les piles et les files.

8.3.1 Les classes Queue et Stack

En prenant `List<ELEM>` comme classe de base, dérivez les deux classes génériques `Queue<ELEM>` et `Stack<ELEM>`. Vous utiliserez ici la dérivation publique, la seule vue en cours.

La classe `Queue<ELEM>` réalise la stratégie `FIFO` (*First In, First Out*) habituelle et doit disposer des fonctions membres suivantes :

- `put(ELEM)` qui ajoute un élément à la file ;
- `get()` qui retourne la valeur de l'élément le plus ancien dans la file, et en retire ce dernier.

La classe `Stack<ELEM>` réalise la stratégie `LIFO` (*Last In, First Out*) habituelle et doit disposer des fonctions membres suivantes

- `push(ELEM)` qui ajoute un élément à la file ;
- `pop()` qui retourne la valeur de l'élément le plus récent dans la file, et en retire ce dernier.

Outre ces fonctions-membres les deux classes doivent être constructibles par défaut, disposer du prédicat `is_empty()` et être affichables sur les `ostream` par l'opérateur `<<`.

Vous devrez définir les fonctions spécifiques de ces classes le plus économiquement possible, en utilisant celles de `List<ELEM>` et en évitant toute duplication de code. En particulier, est-il nécessaire de définir pour ces deux nouvelles classes les opérations suivantes :

- le constructeur ?
- la fonction `is_empty()` ?
- les opérations de copie ?
- la fonction d'affichage (`operator<<`) ?

Bien entendu, votre programme de test devra vérifier ces points. En outre vous ne devez pas modifier le code de la classe `List<ELEM>` fournie ; vous installerez le code des nouvelles classes dans de nouveaux fichiers (`Queue.h` et `Stack.h`, il n'y a vraiment pas besoin de `.cpp` !).

8.3.2 Dérivation privée

Dans notre cas, l'utilisation de l'héritage public présente un défaut majeur sur le plan de l'architecture logicielle. Les instances de `Queue<ELEM>` et `Stack<ELEM>` étant substituables (partout) à celles de `List<ELEM>`, il est possible de leur appliquer directement n'importe quelle fonction-membre de cette dernière classe (`append()`, `prepend()`, `insert()`) ce qui a pour effet de briser la stratégie `FIFO` ou `LIFO` à laquelle elles sont censées obéir !

L'héritage privé permet de pallier cette inconvénient. Quand une classe dérive de manière privée d'une autre classe comme dans

```
class A
{
    // détails omis
};
class B : private A
{
    // détails omis
};
```

tous les membres publics et protégés¹ hérités de la classe de base (A) deviennent privés dans la classe dérivée (B). De plus, la conversion implicite d'héritage est alors invalide, sauf dans le corps des fonctions-membres et amies de la classe dérivée (un B n'est un A que dans le contexte d'un B !).

Modifiez la définition de vos classes `Queue<ELEM>` et `Stack<ELEM>` en utilisant l'héritage privé. Qu'en est-il alors de la question précédente sur la nécessité de redéfinir les fonctions suivantes :

- le constructeur ?
- la fonction `is_empty()` ?
- les opérations de copie ?
- la fonction d'affichage (`operator<<`) ?

Modifiez et testez vos classes en conséquence.

8.3.3 La classe `Priority_Queue`

Pour terminer, définissez une nouvelle classe générique, celle des files à priorité, `Priority_Queue<ELEM>`. Cette classe possède exactement la *même interface* que `Queue<ELEM>`, et donc elle en dérivera *publiquement*. La différence est dans la sémantique de ses deux principales fonctions :

1. Rappelons que les membres *privés* hérités de A, même s'ils sont présents dans la classe dérivée, n'y sont de toute façon pas accessibles. Il ne sont donc pas concernés par ce mécanisme.

- `put(ELEM)` ajoute un élément dans la file, en faisant en sorte que la file à priorité reste ordonnée (dans l'ordre de l'opérateur `<` des `ELEM`, supposé exister) ;
- `get()` retourne la valeur du premier élément de la file à priorité et retire cet élément de la file.

N'avez-vous pas un problème en essayant de définir la fonction `put()` « à l'économie » ? L'héritage privé entre `List<ELEM>` et `Queue<ELEM>` ne pose-t-il pas problème ? Voyez vous une solution possible (autre que le rétablissement de l'héritage public à cet endroit) ?

Posez-vous également la même question sur les fonctions dont la redéfinition est nécessaire et modifiez et testez votre nouvelle classe en conséquence.

8.3.4 Typage dynamique

Dans la classe `List<ELEM>` fournie, aucune des fonctions-membres n'est déclarée virtuelle, à part le destructeur (tiens donc, et pourquoi lui ?). Quelles sont les fonctions pour lesquelles ce serait nécessaire (*indispensable*¹) dans le cadre de l'architecture proposée ? Même question pour les fonctions spécifiques des classes `Queue`, `Stack` et `Priority_Queue` ? En particulier, réfléchissez au cas suivant :

```
Priority_Queue<int> prioq;  
Queue<int> *pq = &prioq;  
pq->put(3);  
pq->put(2);
```

Quel doit être le résultat logique ? Codez en conséquence.

1. J'insiste : il ne s'agit pas ici de goûts ou de couleurs, ni de discipline ou d'habitudes de programmation. Il convient d'identifier les fonctions où l'oubli de la déclaration en tant que `virtual` mettrait en péril (logique ou mécanique) l'architecture de réutilisation de code mise en place.