HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

POO2 - Labo 3 :
Liste doublement chaînée

Rapport

Auteurs : Alison Savary et Luc Wachter
Le 1 mai 2019

Table des matières

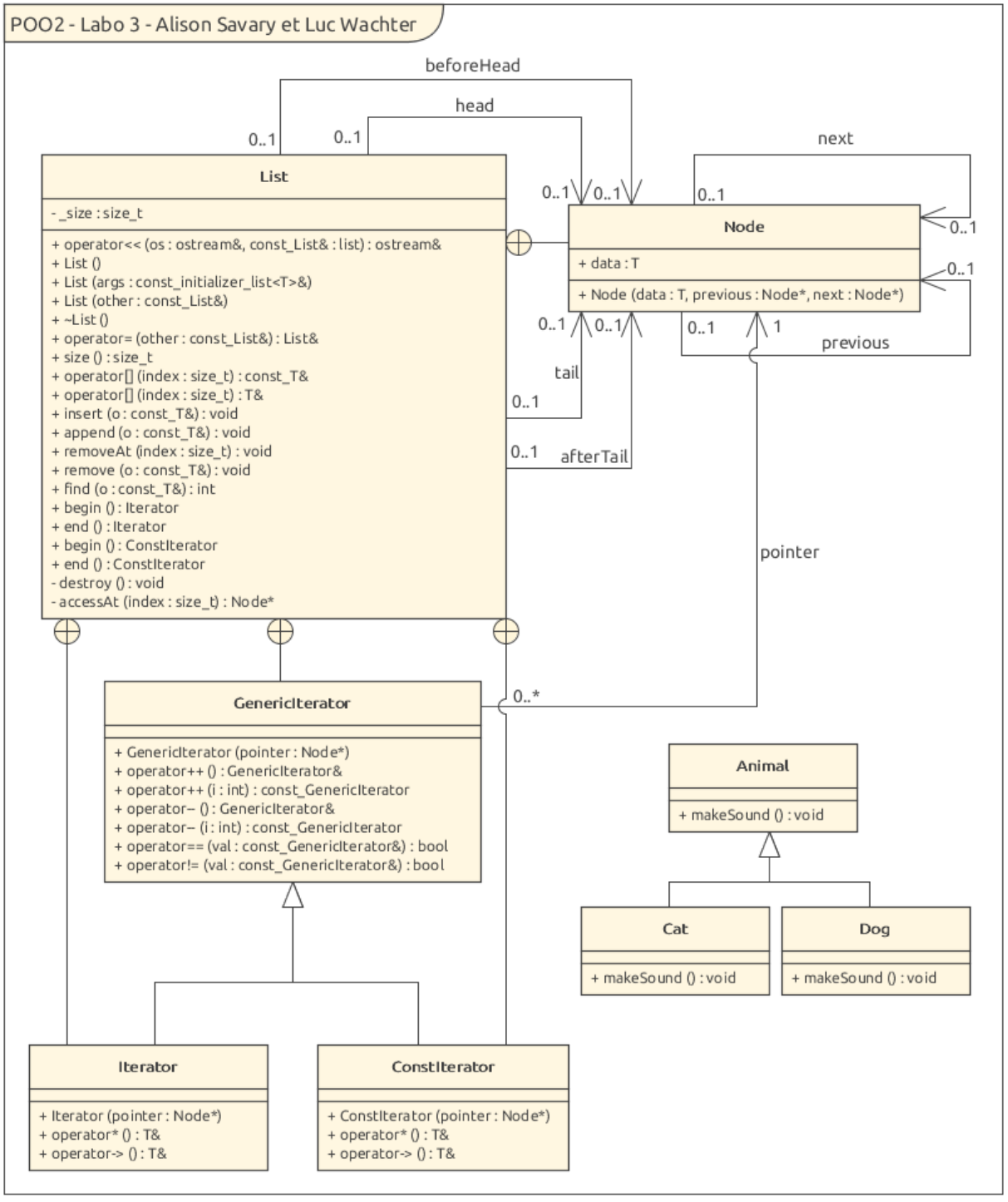
Introduction

Ce laboratoire a pour but l'implémentation d'une liste générique doublement chaînée en C++ ainsi que de trois classes d'itérateurs permettant de parcourir la liste. Cela nous permettra de mettre en pratique nos connaissances de programmation orientée objet appliquée au C++.

Ce document vise à expliquer notre conception de ladite liste, nos choix d'implémentation et les tests effectués.

Conception

Diagramme de classes



Choix d'implémentation

Structure de fichiers

Nous avons choisi d'implémenter notre liste en deux fichiers hpp (`List.hpp` et `ListImpl.hpp`) afin de séparer la déclaration de la définition. Etant donné que nous implémentons une liste générique, nous ne pouvions pas effectuer cette séparation avec un fichier hpp et un fichier cpp, sans devoir restreindre l'utilisation de notre classe à un un instanciation avec des classes choisies à l'avance.

Node

Pour stocker un élément de la liste, nous avons défini une structure interne `Node`. `Node` possède trois membres, `data` (permettant de stocker la donnée), `previous` (un pointeur sur le noeud précédent) et `next` (un pointeur sur le noeud suivant).

Les deux pointeurs contenus dans la structure `Node` peuvent être nul si le noeud est le seul présent dans la liste, ou s'il est le noeud de début ou de fin de la liste. Afin de simplifier sa construction, nous avons défini un constructeur.

Itérateurs

Comme le code de test de la donnée nous le demande, nous avons implémenté les itérateurs en tant que classes internes de `List`. Il y a trois classes d'itérateur : `GenericIterator` et ses deux classes enfants `Iterator` et `ConstIterator`.

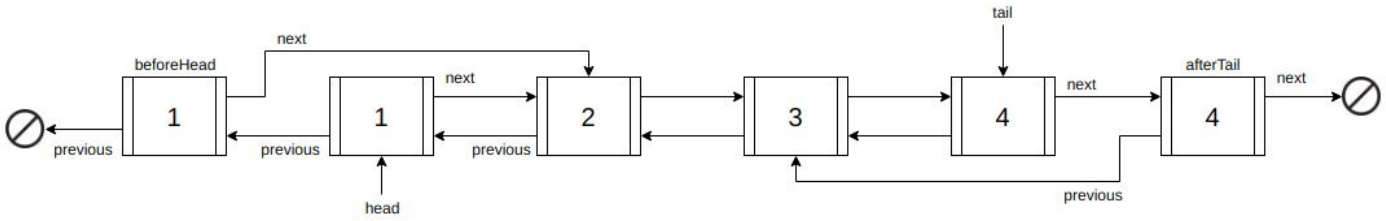
La classe parent implémente les opérateurs `++` et `--` (postfixes et préfixes), `==` et `!=`. Les classes enfants utilisent ces implémentations et définissent les opérateurs `*` et `->`, puisque le type de retour de ceux-ci diffèrent dans le cas d'un `Iterator` (renvoie `T&`) et d'un `ConstIterator` (`const T&`).

Afin de faire fonctionner les itérateurs de manière bi-directionnelle, nous avons réfléchi à un mécanisme en classe, avec plusieurs groupes. Nos implémentations vont donc potentiellement se ressembler. La nôtre, en tout cas, est conçue comme suit :

Nous avons deux `Nodes` sentinelles : l'une avant la `head` (`beforeHead`) et l'autre après la `tail` (`afterTail`). `beforeHead` a pour `next` la `Node` qui suit `head` (`head->next`) et pour valeur la valeur de `head`. De l'autre côté, `afterTail` a pour `previous` l'élément avant la `tail` (`tail->previous`) et pour valeur la valeur de `tail`.

Ainsi, nous pouvons faire en sorte que nos méthodes `begin` et `end` renvoient respectivement `beforeHead` et `afterTail` pour mettre en place un fonctionnement bidirectionnel.

L'illustration suivante permet de transformer l'amas de confusion généré par les précédents paragraphes en claire et délicieuse compréhension.



Hiérarchie de classes `Animal`, `Cat` et `Dog`

Afin d'effectuer des tests sur une liste contenant des éléments plus complexes, nous avons créé trois classes très basiques (contenant toutes la même méthode). Les classes `Cat` et `Dog` héritent de la classe `Animal`. Cette structure très simple nous permet de tester le polymorphisme et la liaison dynamique.

Procédure de tests

Nous avons tenté de reproduire très vaguement des tests unitaires en utilisant uniquement le main. Cela nous a permis de valider les fonctionnalités de la liste au fur et à mesure, et de s'assurer que nous n'introduisions pas de régression.

Afin de s'assurer que nous ne laissions pas de fuites mémoire ou d'erreurs, nous avons régulièrement analysé le projet à l'aide de valgrind.

Description du test	**Résultat attendu**	**Résultat obtenu**
Test simple constructor	Ligne vide	Ligne vide
Test initializer list constructor	12 13 56 90 0	12 13 56 90 0
Test copy constructor	12 13 56 90 0	12 13 56 90 0
Test dynamically allocated list of strings	Rohan Gondor	Rohan Gondor
Test assignment operator	12 13 56 90 0	12 13 56 90 0
Test append method l2.append(42)	12 13 56 90 0 42	12 13 56 90 0 42
Test insert method l2.insert(23)	23 12 13 56 90 0 42	23 12 13 56 90 0 42
Test insert l1.insert("Hello")	Hello Rohan Gondor	Hello Rohan Gondor
Test append l4.append("Ca boom ?")	Hello Rohan Gondor Ca boom ?	Hello Rohan Gondor Ca boom ?

Test removeAt l4.removeAt(1)	Hello Gondor Ca boom ?	Hello Gondor Ca boom ?
Test removeAt l2.removeAt(3)	23 12 13 90 0 42	23 12 13 90 0 42
Test find l2.find(90)	Index of 90 is 3	Index of 90 is 3
Test remove(object) l4.remove("Hello")	Gondor Ca boom ?	Gondor Ca boom ?
Test remove(object) l4.remove("NotHere")	Object not in the list!	Object not in the list!
Test write with brackets operator	23 12 9 90 0 42	23 12 9 90 0 42
Test for each with iterators	what a beautiful world	what a beautiful world
Test for each with const iterators	12 13 56 90 0	12 13 56 90 0
Tests given in the assignment	un deux trois 14 3 42	un deux trois 14 3 42
Test dynamic linking	I am a dog woof, woof ! I am a cat Miaou ! I am a cat Miaou ! I am an animal and make a sound.	I am a dog woof, woof ! I am a cat Miaou ! I am a cat Miaou ! I am an animal and make a sound.

Conclusion

Ce laboratoire nous a permis de gagner de l'expérience avec la programmation orientée objet et avec la généricité en C++. À première vue complexe mais faisable, sa complexité a grandement augmenté lorsque nous avons réalisé que les itérateurs devaient fonctionner dans les deux sens sans recourir aux mêmes solutions que la STL met en pratique (`rbegin()`, `rend()`).

Cependant, sa difficulté était bienvenue, puisqu'il nous a permis d'appliquer bon nombre de concepts et de nous poser des questions intéressantes.

Nous sommes satisfaits du travail que nous fournissons, malgré les quelques factorisations que nous n'avons pas faites avant la fin du temps imparti.