ÉCOLE POLYTECH'NICE SOPHIA
SI4
2020-2021

Julien Deantoni

# Programmation multi paradigmes en CPP

## TD numéro 3

***Constructeur, destructeur, etc***

## Objectif

L'objectif de cet exercice est de manipuler et comprendre l'utilisation de constructeur et de destructeur. De plus, ce TD commence "`from scratch`" afin de continuer votre familiarisation avec la structure d'un programme C++ On peut noter que ce TD utilise des pointeurs. Un pointeur peut être initialisé à `nullptr` lorsqu'il ne pointe sur aucun object. Lors de ce TP, vous manipulerez les opérateurs `new` et `delete` permettant respectivement l'allocation et la libération de mémoire.

## Énoncé du problème

### Avant goût et structure

L'exemple utilisé est celui des arbres binaires. Un arbre binaire possède un noeud racine. Un noeud possède une valeur et référence deux autres noeuds: le fils droit et le fils gauche. Lorsque le noeud est une feuille, il ne possède pas de fils (confère figure 1). Les classes `Tree` et `Node` seront à terme toutes deux des classes paramétrées. Dans un premier temps, il est conseillé de considérer que la valeur d'un noeud est un entier (`int`).

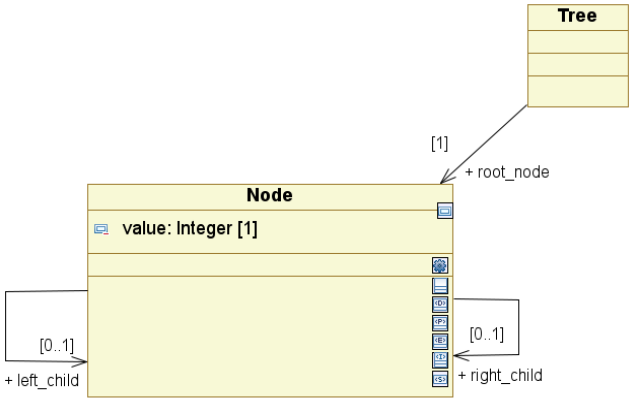


Figure 1: Structure de l'arbre binaire

## Pour commencer

Définissez dans un premier temps la classe `Node` permettant de représenter la structure précédente (`Tree` n'étant qu'un accesseur à la racine de l'arbre). On remarque qu'il n'y a pas de contenance entre un noeud et ces fils. Cela signifie qu'un noeud n'est pas responsable de la création / desctruction de ces fils. Qu'est-ce que cela signifie ? Quel est l'impact ?
Réalisez, bien sûr le `main` correspondant. Compiler. Tester.
Ajouter les Constructeurs qui vous parraissent nécessaires (constructeur par défaut, constructeur par copie, etc) et les instrumenter (par exemple en ajoutant au début et à la fin du code correspondant un affichage sur la sortie standard). Qu'en est-il du destructeur ?
Manipulez les `Node` (création, copie, etc), commentez les résultats obtenus[1]

## La classe `Node`

La classe `Node` possèdent plusieurs fonctions membres dont des accesseurs à chacun de ces attributs. Faites différents tests et validez les différents appels aux constructeurs / destructeurs obtenus sur la sortie standard. De ce fait, réfléchissez sur le prototype de vos fonctions (passage des paramètres par référence, copie ?)
En plus de ces accesseurs, nous ajouterons :

- une fonction membre permettant de supprimer chacun de ces fils (i.e., d'élaguer l'arbre (appelée étrangement `remove_all_children`, figure 2))
- une fonction membre réalisant un parcours infixe de l'arbre et donnant le résultat sous forme d'un vecteur (voir Figure 3).

La classe `Node` doit maintenant ressembler à la description donnée figure 2.

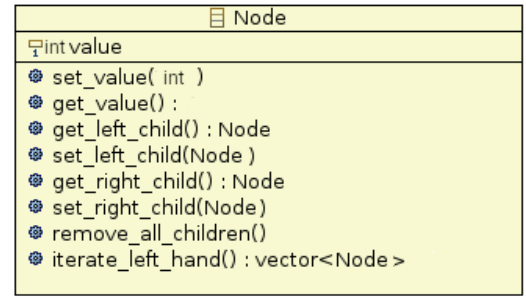


Figure 2: La classe Node

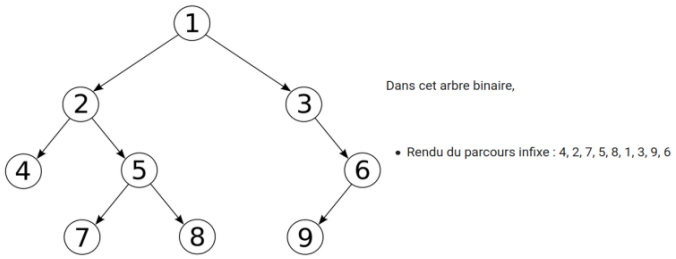


Figure 3: Parcours infixe d'un arbre

[1] n'effacez pas vos test, commentez les si vous n'en avez plus besoin

## La classe `Tree`

Après avoir rigoureusement testée votre classe `Node`, passez maintenant à la réalisation de la classe `Tree`. Imaginez les fonctions membres intéressantes et implémentez les !!

## Une deuxième version...

Maintenant vous allez modifiez votre code afin de prendre en compte la création / desctruction dynamique d'objet. Pour cela vous n'avez qu'à modifier votre code pour refléter la structure de la figure 4. Pourquoi l'implémentation de la figure 4 demande de la création / destruction dynamique ? Quel est l'impact sur le code ? Quelles sont les modifications nécéssaires ?

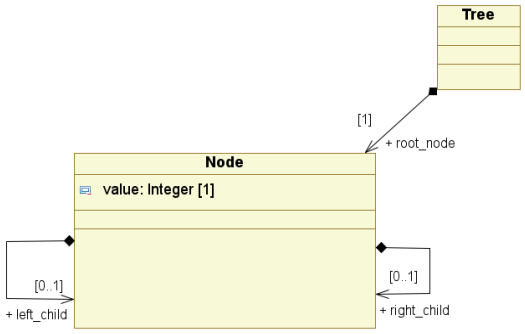


Figure 4: Structure de l'arbre binaire, version 2

## Passons aux `Templates`

Modifiez maintenant votre programme (et votre `Makefile`) afin de pouvoir instancier votre arbre avec différents types. Ensuite, instanciez effectivement votre arbre avec différents type et testez le (N'hésitez pas à prendre des types complexes !). Quelles remarques pouvez-vous faire ? Quelles sont les fonctions membres qui doivent nécessairement être implémentées par les types utilisés ?

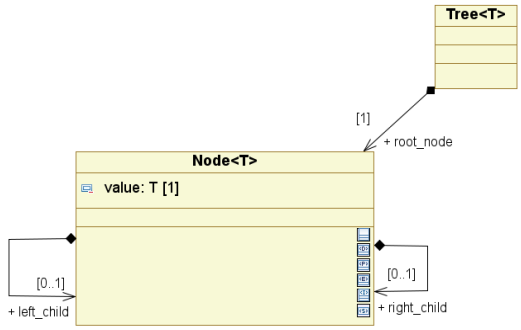


Figure 5: Structure de l'arbre binaire, version *templated*

## Remarques

La gestion de la mémoire est souvent considérée comme quelque chose de difficile par les programmeurs. En particulier par les programmeurs java pour lesquels un "garbage collector" détruit la mémoire au-

tomatiquement lorsque celle ci n'est plus utilisée. Afin de faciliter cette tâche des outils d'analyses existent. Par exemple, valgrind[2], permet, entre autre, d'analyser un programme et de dire si oui ou non des fuites mémoires sont possibles. Plus le programme est complexe et plus il est difficile de comprendre où l'éventuelle erreur a été faite. Cependant cela permet au moins de savoir s'il existe une erreur ou non.
Sur votre programme vous pouver lancer :

```
valgrind ./your_executable
```

Si tout se passe correctement vous devriez obtenir la sortie suivante :

```
==5845== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 1)
==5845== malloc/free: in use at exit: 0 bytes in 0 blocks.
==5845== malloc/free: 2 allocs, 2 frees, 24 bytes allocated.
==5845== For counts of detected errors, rerun with: -v
==5845== All heap blocks were freed -- no leaks are possible.
```

Essayer maintenant d'oublier volontairement de libérer de la mémoire (dans le detructeur de `Node`) et réessayer à nouveau. Vous pourrez si le coeur vous en dit approfondir l'utilisation de valgrind afin d'avoir une confiance plus grande en votre gestion de la mémoire.

## Ajouts

Ajouter à votre classe tous les opérateurs qui vous semble pertinents...
rq: il peut être utile de redéfinir l'opérateur d'affichage pour la classe `Node`. Cet opérateur est de la forme suivante si vous avez utilisé un classe template :

```
    template <typename ELEM>
    std::ostream& operator<<(std::ostream& os, const Node<ELEM>& n);
```

Cette fonction est classiquement une fonction "amie" de la classe Node (même s'il n'existe pas de règle claire sur quelles sont les classes qui doivent être amie ou non) afin de la déclarer comme classe "amie" on utilise le mot clef `friend`. Afin de déclarer cette classe comme amie, votre fichier `Node.h` doit avoir la forme suivante :

```
[...]
template <typename ELEM>
class Node;

template <typename ELEM>
std::ostream& operator<<(std::ostream& os, const Node<ELEM>& n);

template <typename ELEM>
class Node
{
  [...]
    //! overload the printing method
    friend std::ostream& operator<< <>(std::ostream& os, const Node& n);
};
[...]
```

Inspirez-vous en pour les autres opérateurs

[2] http://valgrind.org/