

---

TD 7

---

Fonctions virtuelles

---

**Objectif**

Ces deux exercices illustrent la notion de *fonctions virtuelles* (aussi dénommées *méthodes*). Le premier repose sur la classe `Expr`, introduite en cours, et qui permet de représenter des arbres d'expression arithmétique. Il montre comment l'approche objet bien conçue permet d'augmenter les fonctionnalités d'un programme sans modifier en aucune manière ce qui existe déjà (une propriété que l'on nomme souvent par *incrémentalité*). Il fournit aussi une occasion d'utiliser l'une des collections très utiles de la STL, la *map*.

Le second exercice est une incursion dans le monde de l'*héritage multiple*, mais dans un cas où ce type d'héritage ne pose vraiment aucun problème. Bien au contraire, il permet ici une grande élégance et une réelle économie de moyen.

**7.1 Extension de la classe Expr**

Vous trouverez dans le répertoire `Expr`, à l'endroit habituel, les fichiers source de la classe `Expr` et de ses dérivées. À quelques compléments évidents près, c'est le code vu en cours.

**7.1.1 Deux extensions simples**

Votre première mission est d'ajouter à la hiérarchie des noeuds d'expression deux nouveaux opérateurs :

1. l'opérateur (binaire) de modulo (opérateur `%` en C),
2. l'opérateur (ternaire) conditionnel (`? :` en C).

**Attention !** Vos additions doivent se faire sans modifier en aucune manière les fichiers source contenant les classes fournies (`Expr.h` et `Expr-abstract.h`). Vous introduirez donc vos modifications dans deux nouveaux fichiers, disons `Modulo.h` et `Ternary.h`. En revanche vous aurez bien sûr à ajouter des tests pour les nouveaux opérateurs dans `main_Expr.cpp`.

**7.1.2 Une extension un peu plus compliquée**

Jusqu'à présent nous ne pouvons manipuler que des expressions dont les feuilles (noeuds terminaux) sont des constantes entières (`Constant`). Nous souhaiterions pouvoir manipuler aussi des *variables*.

Une variable a un nom (une simple `string`) et une valeur associée (un `int`). Pour traiter les variables, vous avez trois points à aborder :

- stocker les variables (c'est-à-dire représenter la correspondance nom-valeur) ;
- définir un nouveau type de noeud d'expression 0-aire (feuille), la *référence à une variable*
- définir un nouveau type de noeud d'expression, l'*affectation*.

**Stockage de la correspondance nom-valeur** Notre premier problème est donc de ranger cette correspondance nom-valeur dans une table unique pour toutes les variables (une table globale ou un champ statique de classe). Pour cela, nous allons utiliser une collection de la STL, une *map*.

Une map est une collection générique dépendant de deux paramètres génériques :

`map<K, T>`

$K$  est le type de la « clé » (dans notre cas `string`) ;  $T$  est le type de la valeur associée (dans notre cas `int`). Dans une map, il ne peut y avoir duplication de la même clé : pour une clé donnée, il y a donc au plus une valeur associée dans la map. Les maps se construisent évidemment par défaut (map vide). Enfin (et c'est vraiment la seule opération sur les maps dont vous avez besoin dans cet exercice), une map se manipule comme un tableau<sup>1</sup> dont le type de l'index est  $K$  et  $T$  le type des composantes. Cette indexation est bien entendu représentée par l'opérateur `[]`. Plus précisément soit  $m$  une `map<K, T>` et  $k$  une valeur de type  $K$  ; alors  $m[k]$  réalise l'algorithme suivant :

1. si il existe une entrée correspondant à la valeur de clé  $k$  dans la map (si elle existe, elle est unique par définition des maps), retourner une *référence* ( $T\&$ ) sur la valeur associée ;
2. sinon, allouer une nouvelle entrée correspondant à la valeur  $k$  de la clé, initialiser le  $T$  associé au *zéro* de son type et retourner une référence sur ce  $T$  ainsi initialisé.

On rappelle que le *zéro du type  $T$*  est ce que construit le constructeur par défaut (`T : T()`) de ce type. Dans le cas d'un `int`, c'est évidemment 0.

Les recherches et insertions dans une map sont garanties logarithmiques (réalisation sous forme d'un arbre binaire équilibré).

Pour accéder aux maps, vous devez inclure le fichier d'entête `<map>` et signifier que vous utilisez l'espace de nommage `std` (`using namespace std`).

**Référence à une variable** Pour utiliser les variables dans nos arbres d'expression nous avons besoin d'un nouveau type de noeud, la référence à une variable (`Variable_Ref`). Ces noeuds sont en fait des feuilles (tout comme `Constant`) ; ils contiennent juste le nom de la variable référencée et leur fonction `eval()` retourne simplement la valeur couramment associée à ce nom (dans la map).

Nous ferons l'**hypothèse simplificatrice** (et réaliste) suivante : si on évalue une référence à une variable qui n'a pas encore été définie, ce n'est pas une erreur : au contraire, cela a pour effet de définir la variable correspondante (c'est-à-dire de l'enregistrer dans la map) avec la valeur initiale 0.

**Affectation** Pour changer la valeur d'une variable, nous introduisons un nouveau type de noeud, l'affectation (`Assignment`). Bien qu'il ait deux opérandes, il ne s'agit pas d'une classe dérivée de `Binary_Expr` (pourquoi ?). Le premier opérande est (un pointeur sur) un noeud de référence à une variable (`Variable_Ref`). Le second est (un pointeur sur) une expression quelconque (`Expr`). La fonction `Assignment::eval()` calcule la valeur de cette expression et en fait la nouvelle valeur de la variable (*i.e.*, la range dans la map).

**Attention !** Comme dans la question précédente, vous devez introduire cette extension sans modifier les fichiers source fournis. Vous introduirez donc un nouveau fichier `Variable.h`. Le fichier de test `main_Expr.cpp` contient (en commentaire) un exemple d'utilisation des variables (reprenant les notations indiquées ci-dessus).

---

1. On parle de tableau *associatif*.

## 7.2 Menus en cascade

On souhaite définir un ensemble de classes afin de réaliser une gestion simple (et même simpliste) de menu. Pour cela on se donne la spécification suivante de la classe Menu (fichier Menu/Menu.h, à l'endroit habituel) :

```
class Menu
{
private:
    string _title; // titre du menu
    vector<Menu_Item *> _items;
                                // tableau de pointeurs sur les items
public:
    Menu(const string& t, int n, const vector<Menu_Item *>& its)
        : _title(t), _items(its)
    {}
    const string& title(void) const {return _title;}
    int nitems(void) const {return _items.size();}
    void activate(void) const;
};
```

Un menu est donc constitué d'un titre et d'un certain nombre d'*items*, instances de la classe Menu\_Item. Un menu est initialisé à l'aide d'un tableau (std::vector, la classe vecteur de la STL) de pointeurs sur ses items (pourquoi des pointeurs ? voir plus loin). La fonction activate() active (!) le menu :

- elle affiche le titre du menu suivi de la liste des choix possibles, chaque choix étant identifié par un entier ;
- elle demande à l'utilisateur de faire sa sélection et lit la réponse (un entier) sur l'entrée standard;
- elle exécute la sélection de l'utilisateur puis se termine ;
- si la sélection de l'utilisateur est incorrecte, elle émet un message d'erreur et demande à nouveau une sélection à l'utilisateur.

On voit donc qu'à une exécution de la fonction activate() correspond l'exécution d'une seule sélection.

La classe Menu\_Item est *abstraite* ; en voici la spécification telle qu'elle apparaît dans le fichier Menu/Menu\_Item.h :

```
class Menu_Item
{
private:
    string _text; // le message de selection
public:
    Menu_Item(const string& t) : _text(t) {}
    virtual ~Menu_Item() {}
    virtual string text(void) const {return _text;}
    virtual void execute(void) const = 0;
};
```

Un item est donc constitué d'un texte (qui est affiché pour identifier la sélection) et d'une action associée que l'on peut exécuter grâce à la fonction-membre execute() .

Il existe deux sortes d'items de menu :

- les *items simples* (classe Simple\_Menu\_Item) pour lesquels l'action est représentée par un simple pointeur sur une fonction (disons de type void (\* )()),

- et les *menus déroulants* (classe `Walking_Menu`) pour lesquels l'action est d'activer un sous-menu (c'est-à-dire d'en exécuter la fonction `active()`).

On voit donc qu'un menu déroulant est à la fois un menu et un item de menu. Son message de sélection (en tant qu'item) sera son titre (en tant que menu) que l'on fera suivre de la chaîne de caractères " ->".

Le but de l'exercice est de spécifier et d'implémenter les classes `Simple_Menu_Item` et `Walking_Menu`, de terminer d'implémenter la classe `Menu` conformément aux spécifications données ici, et de fournir un programme de test de l'ensemble. Bien entendu, la solution doit être élégante, économique en code écrit comme en code déroulé, et permettre *un nombre quelconque de niveaux* de menus déroulants.

Voici un exemple d'exécution attendue (sous **zsh**) où l'on reconnaît aisément un menu principal avec un sous-menu. Les actions ont été simplifiées et se contentent d'afficher un message d'exécution. Le programme principal exécuté ici est une boucle infinie d'appel de la fonction `active` du menu. On en sort seulement sur fin de fichier ou par sélection de *quitter*. Dans cet exemple, les caractères entrés par l'utilisateur sont en **gras italique**, ceux affichés par la machine en police normale.

```
2-borobudur% test_Menu

      LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 0
***** Execution de EMACS

      LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 2

      COMMUNICATIONS
0- news
1- xwais
2- xftp
3- xarchie
Votre choix? 2
***** Execution de XFTP

      LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 3
***** Execution de QUITTER
2-borobudur%
```