TP N°4

Programmation C++ - L2

2016

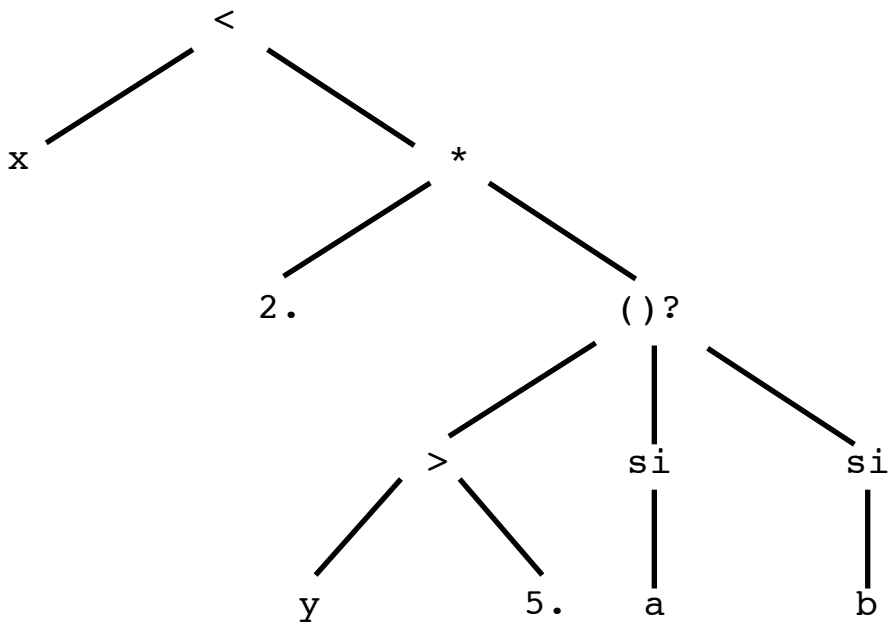
Interpréteur avancé d'expressions arithmétiques

Introduction

On veut écrire un programme capable d'interpréter des expressions arithmétiques classiques comportant :

- des constantes numériques de type réel : `3.7`, `-0.78` ...
- des variables numériques nommées de type réel : «`x`», «`y`» ...
- des opérateurs unaires appliqués à une sous-expression `exp` - tels que `cos(exp)` ou `sin(exp)` - dont le résultat est l'application de l'opérateur sur le résultat de l'évaluation de la sous-expression.
- des opérateurs binaires sur deux opérandes `exp1` et `exp2` - tel que `exp1+exp2` - dont le résultat est l'application de l'opérateur sur les évaluations des deux opérandes.
- des affectations de l'évaluation d'une expression `exp` dans une variable `x` : `x <- exp`.
- des opérateurs ternaires conditionnels `(cond)? alt1 : alt2` dont le résultat est l'évaluation de la sous-expression `alt1` si l'évaluation de la condition `cond` est vraie (ie. ≠ `0.0`) ou le résultat de l'évaluation de `alt2` sinon.

Le principe est de représenter une expression sous la forme d'un arbre dont chaque nœud représente une expression particulière. Les sous-arbres sont les sous-expressions. Ainsi l'expression `x <- 2.0*((y>5.0)?sin(a):sin(b))`, qui retourne la valeur de `x`, pourra être représentée par l'arbre :



On souhaite implémenter chaque expression sous la forme d'une instance d'une classe C++ particulière. On décide donc de créer une classe par type d'expression :

- une classe `Constante`, caractérisée par sa valeur réelle non modifiable,
- une classe pour chaque opérateur unaire, caractérisée par l'opérateur en question et par la sous-expression à laquelle il s'appliquera, par exemple `Cos`, `Sin`...
- *etc...*

Le problème est qu'on ne peut prévoir à l'avance quels seront les types des opérandes (sous-expressions) d'un opérateur donné. On est donc obligé de prévoir un type de nœud générique capable de représenter n'importe quel type particulier à un moment donné.

Quel mécanisme important des langages de programmation objet va-t-on utiliser et comment peut-on le mettre en œuvre en C++ ?

Premiere partie

Dans un premier temps, on se contente des expressions de type constante et opérateurs unaires. L'ensemble du code est en annexe /* expression.h */. Il est concentré dans un fichier en-tête et les instructions de gestion d'erreur ont été retirées pour plus de clarté.

Classe Expression

Que peut-on dire de la méthode `eval` *de la classe* `Expression` *? Qu'est-ce que cela implique pour la classe* `Expression` *? Comment appelle-t-on ce genre de classe ?*

Expliquez ce que permet de faire l'attribut `_pool` *? Est-ce un attribut d'instance ou un attribut de classe ?*

Que fait le constructeur de la classe `Expression` *? Que fait son destructeur ?*

Pourquoi le destructeur de la classe `Expression` *est-il* `virtual` *?*

De quel type est la méthode `toutLiberer` *? Quel est son rôle ? Quand sera-t-elle certainement appelée dans le programme principal ?*

La fonction `friend ostream& operator<<(ostream&, const Expression&)` *est-elle une méthode de la classe* `Expression` *? Cette dernière ne pouvant pas être instanciée, à quoi sert cette fonction ?*

Quel est le nom (`_nom`) *donné à une instance de la classe* `Constante` *?*

Classe Unaire

Décrire exactement ce que fait la méthode `str()` *de la classe* `Unaire`.

Dessiner le diagramme de classes du fichier `expression.h`. *Faites ressortir les classes abstraites.*

Synthese

Quelle est la sortie standard obtenue par l'exécution du programme principal suivant ?

```
#include "expression.h"
void main() {
     cout << "Premiere partie" << endl;
     Cos      *c = new Cos(new Constante(M_PI/3.0));

     cout << *c << " = " << c->eval() << endl;
     Expression::toutLiberer();
}
```

Deuxieme partie : operateurs binaires

On s'intéresse maintenant aux opérateurs binaires arithmétiques ainsi qu'aux comparaisons. On considérera celles-ci comme des opérateurs binaires dont le résultat de l'évaluation est faux si égal à `0.0` et vrai si différent de `0.0`. Voir le code en annexe /* binaire.h */.

Ecrivez les classes `Produit` *et* `InferieurEgal` *qui permettent respectivement de représenter la multiplication et la comparaison <= de deux expressions.*

À partir de maintenant, pour simplifier l'impression des traces d'exécution, on supprime les instructions d'affichage des constructeur et destructeur de la classe `Expression` : ~~`cout << "enregistrer " << _nom << endl;`~~ et ~~`cout << "liberer " << _nom << endl;`~~

Quelle est la sortie standard obtenue par l'exécution du programme principal suivant ?

```
#include "binaire.h"
void main() {
     cout << "Deuxieme partie" << endl;
     Somme    *s = new Somme(     new Constante(1.0),
```

```
                        new Produit(new Constante(2.0),
                                    new Sin(new Constante(M_PI/6.0))
                        )
            );
    cout << *s << " = " << s->eval() << endl;

    Superieur comp(s, new Constante(1.8));
    cout << comp << " = " << (bool)comp.eval() << endl;
    Expression::toutLiberer();
}
```

TROISIEME PARTIE : VARIABLES

On s'intéresse maintenant à une expression particulière : la variable. Une même variable peut se retrouver dans plusieurs expressions différentes ou plusieurs fois dans une même expression. Par exemple l'expression `x*cos(x)` fait deux fois référence à la variable `x`. Dans notre représentation d'un expression, nous aurons donc besoin de deux instances d'une même classe `Variable` portant le même nom «x». Voir le code en annexe /* variable.h */.

Pourquoi stocker la valeur d'une variable dans un attribut d'instance de la classe Variable pose problème et ne constitue pas une bonne solution ?

Expliquez ce que permet de faire l'attribut de classe `_memoire` *?*

De quel type est la méthode `effacerMemoire` *? Quel est son rôle ? Quand sera-t-elle certainement appelée dans le programme principal ?*

Décrire en détail ce que fait la méthode `eval()` *de la classe* `Affectation`*. Quelle est la valeur retournée ?*

Quelle est la sortie standard obtenue par l'exécution du programme principal suivant :

```
void part3() {
    cout << "Troisieme partie" << endl;

    Variable x("x", 3.0);
    Variable y("y");
    cout << x << " = " << x.eval() << endl;
    cout << y << " = " << y.eval() << endl;

    Expression *exp = new Somme( new Constante(1.0),
                        new Produit(new Constante(2.0), new Variable("x")));
    Affectation     *a = new Affectation(new Variable("y"), exp);
    cout << *a << " = " << a->eval() << endl;
    cout << y << " = " << y.eval() << endl;

    Variable::effacerMemoire();
    cout << y << " = " << y.eval() << endl;
}
```

Représentez graphiquement l'état en mémoire de l'expression `*a` *ainsi que l'état de l'attribut de classe* `_memoire` *lors de l'exécution du programme précédent, juste avant l'appel de* `Variable::effacerMemoire();`.

QUATRIEME PARTIE : CONDITIONNEL

On veut maintenant créer une classe qui permet de représenter l'opérateur ternaire `(cond)?e1:e2` qui retourne le résultat de l'évaluation de l'une ou l'autre sous-expression `e1` ou `e2` en fonction du résultat de l'évaluation d'une expression conditionnelle `cond`.

Ecrivez l'interface et l'implémentation d'une classe `Conditionnel`*, dans l'esprit de cet exercice, qui permette d'écrire et d'exécuter le programme principal suivant :*

```
#include "conditionnel.h"
void main () {
     Conditionnel *test =
          new Conditionnel(new InferieurEgal(new Variable("x"), new Constante(0.0)),
                    new Cos(new Variable("x")),
                    new Cos(new Produit(new Constante(2.0), new Variable("x"))));
     Variable *x = new Variable("x", M_PI/3.0);
     cout << *x << " = " << x->eval() << endl;
     cout << *test << " = " << test->eval() << endl;
     x->set( -M_PI/3.0);
     cout << *x << " = " << x->eval() << endl;
     cout << *test << " = " << test->eval() << endl;

     Variable::effacerMemoire();
     Expression::toutLiberer();
}
```

CINQUIEME PARTIE : BOUCLES ET BLOC

BOUCLES

Analysez la classe `Pour` du fichier `boucle.h` en annexe.

Décrivez ce que fait la fontion eval de cette classe ?

Dessinez l'arbre correspondant à une expression qui affecte la factorielle d'une variable x (dont la valeur a été affectée précédemment) à une variable y.

Ecrivez le programme principal qui construit un tel arbre en mémoire et évalue l'expression pour `x=3.0` *puis* `x=5.0`.

BLOC

Créer une classe `Bloc` *qui représente une séquence d'instructions (expression) sur le même modèle. On considère que la valeur retournée par un bloc est celle retournée par la dernière instruction/expression du bloc.*

SIXIEME PARTIE : SYNTHESE

Ecrivez le programme principal qui construit et évalue l'arbre d'interprétation qui calcule la factorielle d'une variable x. Créer les classes d'opérateur manquantes si nécessaire.

Il ne reste plus qu'à écrire un analyseur syntaxique pour créer des arbres / expressions à partir de fichiers ou de chaînes de caractères, mais ceci est une autre histoire !...

ANNEXES

/* expression.h */

```
#include <iostream>
#include <sstream>
#include <string>
#include <math.h>
#include <map>
#include <set>
using namespace std;

/* retourne une valeur reelle (val) sous forme de chaine de caracteres */
string string_from_double(double val) { ostringstream os; os << val; return os.str(); }

class Expression {
protected:
     string _nom;
private:
     static set<Expression*> _pool;
public:
     /* applique itérativement l'operateur delete au premier element
        du pool tant que le pool n'est pas vide */
     static void toutLiberer() {
       set<Expression*>::iterator  it;
       while ( (it = _pool.begin()) != _pool.end() )
             delete (*it);
     }
public:
     Expression(const string& nom) : _nom(nom) {
       _pool.insert(this);
       cout << "enregistrer " << _nom << endl;
     }
     virtual ~Expression() {
       _pool.erase(this);
       cout << "liberer " << _nom << endl;
     }

     virtual double eval() const = 0;
     virtual string str() const { return _nom; }
     friend ostream& operator<<(ostream& os, const Expression& exp) {
       return os << exp.str();
     }
};

class Constante : public Expression {
     double _value;
public:
     Constante(double val) : Expression(string_from_double(val)), _value(val) {}
     virtual double eval() const { return _value; }
};

class Unaire : public Expression {
```

```
protected:
     Expression *_op;
public:
     Unaire(const string& nom, Expression* op) : Expression(nom), _op(op) {}

     virtual double eval() const = 0;
     virtual string str() const { return _nom + "(" + _op->str() + ")"; }
};

class Cos : public Unaire {
public:
     Cos(Expression* op) : Unaire("cos", op) {}

     double eval() const { return cos(_op->eval()); }
};

class Sin : public Unaire {
public:
     Sin(Expression* op) : Unaire("sin", op) {}

     double eval() const { return sin(_op->eval()); }
};
```

/* binaire.h */

```
#include "expression.h"

class Binaire : public Expression {
protected:
     Expression      *_opleft, *_opright;
public:
     Binaire(const string& nom, Expression* left, Expression* right) : Expression(nom),
                                        _opleft(left), _opright(right) {}
     virtual double eval() const = 0;
     virtual string str() const {
       return "(" + _opleft->str() + " " + _nom + " " + _opright->str() + ")";
     }
};

class Somme : public Binaire {
public:
     Somme(Expression* left, Expression* right) : Binaire("+", left, right) {}

     double eval() const { return _opleft->eval() + _opright->eval(); }
};

class Superieur : public Binaire {
public:
     Superieur(Expression* left, Expression* right) : Binaire(">", left, right) {}

     double eval() const { return (double)(_opleft->eval() > _opright->eval()); }
};
```

/* variable.h */

```
#include "binaire.h"
class Variable : public Expression {
public:
     Variable(const string& nom) : Expression(nom) { }
     Variable(const string& nom, double init) : Expression(nom) {
       _memoire[_nom] = init;
     }
      /*      retourne la valeur de la variable dont le nom est dans _nom
       si _nom n'est pas trouve dans _memoire alors une nouvelle variable
       est cree avec la valeur 0.0 qui est retournee */
     double eval() const { return _memoire[_nom]; }
     double set(double val) { _memoire[_nom] = val; return val; }
private:
     /*ensemble des variable sous la forme d'un tableau
       associatif : la cle est le nom de la variable stockee */
     static map<string,double>    _memoire;
public:
     /* vide completement le tableau associatif _memoire */
     static void effacerMemoire() { _memoire.clear(); }
};

class Affectation : public Binaire {
public:
     Affectation(Variable* var, Expression* exp) : Binaire("<-", var, exp) {}
     double eval() const { return ((Variable*)_opleft)->set(_opright->eval()); }
};
```

/* boucle.h */

```
#include "variable.h"
class Pour : public Expression {
protected:
     Expression *_init, *_condition, *_incremente, *_calcul;
public:
     Pour(Expression *init, Expression *cond, Expression *inc, Expression *calc) :
     Expression("for"), _init(init), _condition(cond), _incremente(inc), _calcul(calc) {}

     double eval() const {
       double res;
       _init->eval();
       while (_condition->eval() != 0.0) {
              res = _calcul->eval();
              _incremente->eval();
       }
       return res;
     }
     string str() const {
       return _nom + " (" + _init->str() + " ; " + _test->str() + " ; "
                          + _incremente->str() + ") { " + _calcul->str() + " }";
     }
};
```

set

std::set<T,...>

La classe set permet de décrire un ensemble ordonné et sans doublons d'éléments de type T. Le type T doit disposer d'un constructeur vide T().

Exemple :

```
#include <set>
#include <iostream>
int main(){
  std::set<int> s; // équivaut à std::set<int,std::less<int> >
  s.insert(2); // s contient 2
  s.insert(5); // s contient 2 5
  s.insert(2); // le doublon n'est pas inséré
  s.insert(1); // s contient 1 2 5
  std::set<int>::const_iterator
    sit (s.begin()),
    send(s.end());
  for(;sit!=send;++sit) std::cout << *sit << ' ';
  std::cout << std::endl;
  return 0;
}
```

Attention : le fait de supprimer ou ajouter un élément dans un std::set rend invalide ses iterators. Il ne faut pas modifier un std::set dans une boucle for basée sur ses iterators

map

std::map<K,T,...>

Une map permet d'associer une clé (identifiant) à une donnée (table associative).
La map prend au moins deux paramètres templates :

1. le type de la clé K
2. le type de la donnée T

À l'image du std::set, le type K doit être ordonné. Le type T impose juste d'avoir un constructeur vide.

Attention : le fait de supprimer ou ajouter un élément dans un std::map rend invalide ses iterators. Il ne faut pas modifier un std::map dans une boucle for basée sur ses iterators.

Attention : le fait d'accéder à une clé via l'opérateur [ ] insère cette clé (avec la donnée T()) dans la map. Ainsi l'opérateur [ ] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find. De plus, il ne garantit pas la constance de la map (à cause des insertions potentielles) et ne peut donc pas être utilisé sur des const std::map.

Exemple :

```
#include <map>
#include <string>
#include <iostream>

int main(){
  std::map<std::string,unsigned> map_mois_idx;
  map_mois_idx["janvier"] = 1;
  map_mois_idx["février"] = 2;
  //...
  std::map<std::string,unsigned>::const_iterator
    mit (map_mois_idx.begin()),
    mend(map_mois_idx.end());
  for(;mit!=mend;++mit)
    std::cout << mit->first << '\t' << mit->second << std::endl;
  return 0;
}
```

extraits de *http://www.commentcamarche.net/faq*