Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°7

Exercice 1: Pointeurs de fonction

On veut créer un arbre permettant de stocker et d'évaluer une expression de calcul arithmétique univarié (i.e. dépendant au plus d'une variable x) sur des flottants. Cette expression sera stockée sous forme d'une arbre, avec aux nœuds les opérateurs (binaires ou unaires) et aux feuilles les valeurs numériques ou une variable. Les opérateurs devront être stockés sous forme d'un pointeur de fonction.

1. comment doivent être définie les fonctions ?
2. donner la structure générale de la classe, les structures internes, les fonctions et méthodes nécessaires.
3. donner la classe permettant de représenter un nœud générique.
4. donner la classe permettant de représenter un nœud binaire.
5. donner la classe permettant de représenter un nœud unaire.
6. donner la classe permettant de représenter une feuille représentant un valeur.
7. donner la classe permettant de représenter une feuille représentant un variable.
8. définir la fonction permettant de lancer l'évaluation de l'arbre.
9. définir un constructeur permettant de construire un exemple d'arbre.
10. écrire un code d'exemple d'utilisation.

Solution:

1. elles peuvent être déjà définie, ou définie dans la classe (en statique), mais doivent avoir le prototype float(*)(float,float) pour les fonctions binaires, et float (*)(float) pour les fonctions unaires.
2. structure générale :

```
class Arithm {
public:
    // définition des fonctions de base (non déjà existante)
  static float add(float x, float y) { return x + y; }
  static float mul(float x, float y) { return x * y; }
  using Uoper = float (*)(float);
  using Boper = float (*)(float, float);
```

```
private:
  struct Gnode; // pour représenter un noeud générique
  struct Bnode; // pour représenter un noeud binaire
  struct Unode; // pour représenter un noeud unaire
    struct lVal;  // pour représenter un feuille valeur numérique
    struct lVar;  // pour représenter un feuille variable (x)
  Gnode   *root;  // la racine de l'arbre
```

```
public:
  Arithm() // constructeur
  float eval(float x) // évaluation
    // il faudrait une fonction de parsing d'une chaine pour construire l'arbre
};
```

3. nœud générique.

```
struct Gnode {
  virtual float eval(float x) = 0;
    virtual ~Gnode() {}
};
```

4. nœud binaire

```
struct Bnode final : Gnode {
  Boper fun;
  Gnode   *left, *right;
  Bnode(Boper f, Gnode *l, Gnode *r) : fun(f), left(l), right(r) {}
  float eval(float x) override { return (*fun)(left->eval(x), right->eval(x)); }
};
```

5. nœud unaire

```
struct Unode final : Gnode {
  Uoper fun;
  Gnode   *down;
  Unode(Uoper f, Gnode *d) : fun(f), down(d) {}
  float eval(float x) override { return (*fun)(down->eval(x)); }
};
```

6. feuille représentant un valeur

```
struct lVal final : Gnode {
  float value;
  lVal(float v) : value(v) {}
  float eval(float x) override { return value; }
};
```

7. feuille représentant un variable

```
struct lVar final : Gnode {
  lVar() {}
  float eval(float x) override { return x; }
};
```

8. fonction permettant de lancer l'évaluation de l'arbre

```
float eval(float x) {
  return root->eval(x);
}
```

9. constructeur d'exemple

```
Arithm() {
  root = new Bnode( add,
    new Unode(sinf, new lVal(4.2f)),
    new Bnode(mul, new lVal(7.1f), new lVar())
    );
}
```

10. exemple d'utilisation

```
Arithm r;
cout << r.eval(2.3f) << endl;
```

Exercice 2: Fonctor

on veut effectuer le seuillage d'une valeur v avec une fonctionnelle telle que si $v < v_{min}$ alors $v = v_{min}$, et si $v > v_{max}$ alors $v = v_{max}$.

1. écrire la fonction qui réalise cette tâche.
2. quel est l'inconvénient de cette fonction ?
3. écrire un fonctor qui stocke les valeurs v_{min} et v_{max}.
4. ajouter deux états internes permettant de compter le nombre d'overflows et d'underflows, et ajouter les méthodes nécessaires à la gestion de ces compteurs.
5. y-a-t-il des inconvénients à transformer ce fonctor en template ?

Solution:

1. fonction :

```
int fseuil(int v, int vmin, int vmax) {
  return (v < vmin ? vmin : (v > vmax ? vmax : v));
}
```

2. le v_{min} et le v_{max} doivent être passés à chaque appel, même si ces valeurs sont les mêmes à chaque fois.

3. fonctor :

```
class Seuil {
private:
  int vmin, vmax;
public:
  Seuil(int x, int y) : vmin(x), vmax(y) {}
  int operator()(int v) const {
        // noter le const qui permet de passer le fonctor par référence
        // const et continuer à l'utiliser
    return (v < vmin ? vmin : (v > vmax ? vmax : v));
  }
  // setter vmin et vmax
};
```

```
Seuil seuil(0, 255);
int x = seuil(45);
```

seul l'objet fonctionnel doit être passé (peut contenir un nombre quelconque de paramètres).

4. voir le code :

```
class Seuil {
private:
  int    vmin, vmax;
  mutable size_t  underflow{}, overflow{};
    // mutable afin de les modifier même si l'objet est const
public:
  Seuil(int x, int y) : vmin(x), vmax(y) {}
  int operator()(int v) const {
    if (v < vmin) {
      ++underflow;
      return vmin;
    }
    else if (v > vmax)  {
      ++overflow;
      return vmax;
    } else return v;
  }
  void reset() const { underflow = overflow = 0; }
  size_t getUnderflow() const { return underflow; }
  size_t getOverflow() const { return overflow; }
};
```

Le fonctor peut ainsi non seulement stocker des valeurs de paramètres, mais des états résultant de ses propres opérations.

5. non (même utilisation) :

```
template <class T> class Seuil {
private:
  T    vmin, vmax;
  mutable size_t  underflow{}, overflow{};
  // mutable afin de les modifier même si l'objet est const
public:
  Seuil(const T& x, const T& y) : vmin(x), vmax(y) {}
  int operator()(int v) const {
    if (v < vmin) { // exige surcharge < pour T
      ++underflow;
      return vmin;
    }
    else if (vmax < v)  { // idem
      ++overflow;
      return vmax;
    } else return v;
  }
  void reset() const { underflow = overflow = 0; }
  size_t getUnderflow() const { return underflow; }
  size_t getOverflow() const { return overflow; }
};
```

Exercice 3: Création d'un patron de fonction de comptage

On veut dans cet exercice construire une fonction générique pouvant utiliser une fonctionnelle quelconque sur un tableau générique (on suppose que ce tableau générique implémente la méthode `size()` pour retourner sa taille, et surcharge l'opérateur `[]` pour accéder aux éléments).

1. écrire une fonction template `count` qui permet de compter combien d'éléments sont égaux à une certaine valeur `v`, et donner un exemple d'utilisation.

2. on veut maintenant fournir sa propre implémentation de l'opérateur de comparaison. Que doit-on passer en paramètre si celui-ci est un pointeur de fonction ? Donner alors le code utilisant ce pointeur de fonction.
3. peut-on en argument passer un fonctor ou une lambda fonction ?
4. que peut-on en déduire ?
5. on veut passer en paramètre supplémentaire au template (=un type supplémentaire) pour stocker le type fonctionnel, mais sans passer de paramétre supplémentaire.
   (a) quel est le type d'objets fonctionnels qui peuvent étre transmis au template pouvant étre utilisé pour effectuer la comparaison ?
   (b) donner le code associé.
   (c) y-a-t-il moyen de donner une valeur par défaut aux paramètres du template ?
   (d) donner le code associé.
6. on passe maintenant le paramètre supplémentaire du template comme argument additionnel à la fonction afin de tenter de gérer d'autres types fonctionnels que les fonctors.
   (a) donner le code associé.
   (b) avec quels types de fonctionnels ce code peut-il marcher ?
   (c) peut-on mettre un type fonctionnel par défaut pour ce template ?
   (d) quelle est alors la solution ?
7. comment utiliser cette fonction `count` afin de compter le nombre d'éléments différents de la valeur passée ?
8. comment utiliser cette fonction `count` afin de compter le nombre d'éléments pairs ?
9. peut-on utiliser la fonctionnelle afin de changer les éléments du tableau ?

Solution:

1. Utiliser la fonction suivante :

```
template <class T> size_t count(const vector<T> &array, const T& val) {
  size_t cnt = 0;
  for (size_t i = 0; i < array.size(); i++) if (array[i] == val) cnt++;
  return cnt;
}
```

```
vector<int>  v = { 4, 8, 2, 5, 2, 5, 5 };
size_t c = count(v, 5);
```

2. type du pointeur `bool (*)(const T&, const T&)`

```
using PtCmpType = bool (*)(const T\&, const T\&);

template <class T> size_t count2(const vector<T> &array, T val, PtCmpType cmp) {
  size_t cnt = 0;
  for (size_t i = 0; i < array.size(); i++)
    if (cmp(array[i],val)) cnt++;
  return cnt;
}

bool CmpInt(const int& x, const int& y) { return (x == y); }
```

```
size_t cnt2 = count2(v, 5, CmpInt);
```

3. directement non, les types ne peuvent pas compris comme des pointeurs de fonction.
   En revanche, une lambda expression peut être convertie en pointeur de fonction.

```
using ptFunType = bool(*)(const int&, const int&);
ptFunType FunPt1 = [](const int& a, const int& b) -> bool { return (a == b); };
size_t cnt3 = count2(v, 5, FunPt1);
```

4. que mettre comme type de paramètre un pointeur de fonction réduit le type de fonctionnelle que l'on peut passer à une fonction.
5. paramétre de template seul
   (a) le type seul n'est jamais un objet fonctionnel. Par contre, dans le cas d'un fonctor, s'il existe un constructeur par défaut, un objet fonctionnel peut être construit à partir du type.
   (b) code

```
class FunctorCmpInt {
public:
  bool operator()(const int& a, const int& b) { return a == b; }
};

template <class T, class Cmp> size_t count3(const vector<T> &array, T val) {
  size_t cnt = 0;
  Cmp    cmp;  // fonctor
  for (size_t i = 0; i < array.size(); i++)
    if (cmp(array[i], val)) cnt++;
  return cnt;
}
size_t cnt4 = count3<int, FunctorCmpInt>(v, 5);
```

   (c) oui, mais il faut alors que la valeur par défaut soit aussi un fonctor template.
   (d) code

```
template <class T, class Cmp = FunctorCmp<T>>
    size_t count4(const vector<T> &array, const T& val) { ... }
size_t cnt4 = count4(v, 5);
```

6. paramétre fonctionnel
   (a) code :

```
template <class T, class Cmp>
    size_t count5(const vector<T> &array, const T& val, Cmp fun) {
  size_t cnt = 0;
  for (size_t i = 0; i < array.size(); i++)
    if (fun(array[i], val)) cnt++;
  return cnt;
}
```

   (b) ce code fonctionne avec les types suivants :

```
// pointeur de fonction
// Cmp = bool (*)(const int&, const int&)
// fun = CmpInt
size_t cnt6a = count5(v, 5, CmpInt);
// lambda expression
// Cmp = type de la lambda-expression
// fun = instance de la lambda-expression
size_t cnt6b = count5(v, 5, [](int a, int b) -> bool { return (a == b); });
// instance d'un fonctor
// Cmp = FunctorCmpInt (ceci est bien un type)
// fun = FunctorCmpInt() (constructeur par défaut, donc instance de ce type)
size_t cnt6c = count5(v, 5, FunctorCmpInt() );
```

   Noter que le type des arguments n'est plus du tout contraint puisqu'il s'agit d'un template.
   (c) comme la fonction est un template, la valeur du paramétre par défaut doit elle-aussi étre de type template afin de produire une valeur par défaut pour le type indiqué. Donc,

— cela ne peut pas être un pointeur de fonction (il est typé)
— cela ne peut pas être une lambda expression (elle est aussi typé)
— cela ne peut pas être une fonction template instancié pour le type T. Par exemple :

```
template <class T> bool CmpTemplate(const T& a, const T& b) {
  return a == b;
}
// et prendre comme valeur par défaut CmpTemplate<T>.
```

car le compilateur ne peut pas déduire l'argument du modéle (la déduction de type ne peut pas être faite).
— cela ne peut pas être non plus un fonctor temporaire (même problème que pour le template).

(d) il faut préciser le type par défaut utilisé comme argument par défaut dans les arguments du template :

— avec un pointeur de fonction sur l'instanciation d'un template

```
template <class T, typename Cmp = bool (*)(const T&,const T&)>
  size_t count5d(const vector<T> &array, T val, Cmp fun = CmpTemplate<T>)
    { ... }
```

— avec l'instance d'un fonctor temporaire

```
template <class T> class FunctorCmp {
  public: bool operator()(const T& a, const T& b) { return a==b; }
};

template <typename T, typename Cmp = FunctorCmp<T>>
size_t count5d(const vector<T> &array, const T& val,
          Cmp fun = FunctorCmp<T>() ) {
  ...
}
```

7. nombre d'éléments différents de 5 :

```
// Cmp = std::not_equal_to<int>
// fun = std::not_equal_to<int>() instance du fonctor (constructeur par défaut)
cnt = count5e(v, 5, std::not_equal_to<int>());

// Cmp = type de la lambda-expression
// fun = instance associée à la lambda-expression
cnt = count5e(v, 5, [](int a, int b) { return (a != b); });
```

8. nombre d'éléments pairs :

```
struct ModuloN {
  int N;
  ModuloN(int n): N(n) {}
  bool operator()(int x, int y) { return (x % N == 0); }
};
// noter que operator() ignore le paramètre y
```

```
ModuloN  mod2(2);
cnt = count5e(v, 5, mod2);
```

9. non, le tableau est passé en `const&`.

Exercice 4: Pointeurs de fonction vs Héritage

On veut donner un exemple concret qui permet de comparer les deux approches en terme de flexibilité.

1. soit un objet de type `bind` pour lequel il existe deux grandes familles (`standard`, `safe`) d'implémentation de deux méthodes `open`, `close`. Expliquer comment obtenir organiser les classes afin d'obtenir les implémentations choisies.
2. en cours de connexion, il devient possible de passer de standard à safe et vice-versa. Quel problème ce changement pose-t-il dans l'approche sous forme de classe ?
3. proposer une alternative avec des classes.
4. proposer une implémentation utilisant les pointeurs de fonction.
5. quel est un autre avantage de l'implémentation par pointeur ?

Solution:

1. voir le code :

```
struct iConnect {
  virtual void open() = 0;
  virtual void close() = 0;
};
```

```
class Bind : iConnect {
  // partie commune
};
```

```
class StandardBind : public Bind {
public:
  void open() override { /* open for standard bind */ }
  void close() override { /* close for standard bind */ }
};
```

```
class SafeBind : public Bind {
public:
  void open() override { /* open for safe bind */ }
  void close() override { /* close for safe bind */ }
};
```

```
Bind *b = new StandardBind();
b->open();
...
b->close();
```

2. suppose que l'interface peut être changée après création de l'objet. Par exemple de la manière suivante :

```
struct iConnect {
  virtual void open() = 0;
  virtual void close() = 0;
};
```

```
class StandardBind : public iConnect {
public:
  void open() override { /* open for standard bind */ }
  void close() override { /* close for standard bind */ }
};
```

```
class SafeBind : public iConnect {
public:
  void open() override { /* open for safe bind */ }
  void close() override { /* close for safe bind */ }
};
```

```
class Bind : iConnect {
public: enum Mode { standard, safe };
private:
  iConnect *connect;
  Mode      mode;
```

```
public:
  Bind(Mode m) : mode(m) {
    if (mode == standard)  connect = new StandardBind;
    else if (mode == safe) connect = new SafeBind;
  }
```

```
  void open() { connect->open(); }
  void close() { connect->close(); }
```

```
  void SwitchToSafe() {
    if (mode != safe) {
      delete connect;
      connect = new SafeBind;
      mode = safe;
    }
  }
  void SwitchToStandard() {
    if (mode != standard) {
      delete connect;
      connect = new StandardBind;
      mode = standard;
    }
  }
};
```

3. Avec le polymorphisme, le type d'un objet défini la totalité de ses fonctions. Il faudrait donc changer le type de l'objet en cours de l'exécution.
4. On propose d'avoir une classe ne contenant que les implémentations qui est allouée et réallouée en fonction de l'implémentation souhaitée.

```
struct iConnect {
  virtual void open() = 0;
  virtual void close() = 0;
};
```

```
class StandardBind : public iConnect {
public:
  void open() override { /* open for standard bind */ }
  void close() override { /* close for standard bind */ }
};
```

```
class SafeBind : public iConnect {
public:
  void open() override { /* open for safe bind */ }
  void close() override { /* close for safe bind */ }
};
```

```
class Bind : iConnect {
public: enum Mode { standard, safe };
private:
  iConnect *connect;
  Mode      mode;
```

```
public:
  Bind(Mode m) : mode(m) {
    if (mode == standard)  connect = new StandardBind;
    else if (mode == safe) connect = new SafeBind;
  }
```

```
  void open() { connect->open(); }
  void close() { connect->close(); }
```

```
  void SwitchToSafe() {
    if (mode != safe) {
      delete connect;
      connect = new SafeBind;
      mode = safe;
    }
  }
  void SwitchToStandard() {
    if (mode != standard) {
      delete connect;
      connect = new StandardBind;
      mode = standard;
    }
  }
};
```

5. implémentation avec pointeurs

```
class Bind {
public:
  enum Mode { standard, safe };
  using PtFun = void(*)();
  PtFun openFun;
  PtFun closeFun;
```

```
private:
  static void standard_open() { cout << "open for standard bind\n"; }
  static void standard_close() { cout << "close for standard bind\n"; }
  static void safe_open() { cout << "open for safe bind\n"; }
  static void safe_close() { cout << "close for safe bind\n"; }
  Mode       mode;
```

```
public:
  inline void open() { (*openFun)(); }
  inline void close() { (*closeFun)(); }
  Bind(Mode m) {
    if (m == standard)  SwitchToStandard();
    else if (m == safe) SwitchToSafe();
  }
```

```
  void SwitchToSafe() {
    if (mode != safe) {
      openFun = safe_open;
      closeFun = safe_close;
      mode = safe;
      cout << "switch to safe\n";
    }
    else cout << "already in safe mode\n";
  }
```

```
  void SwitchToStandard() {
    if (mode != standard) {
      openFun = standard_open;
      closeFun = standard_close;
      mode = standard;
      cout << "switch to standard\n";
    }
    else cout << "already in standard mode\n";
  }
};
```

```
void exec() {
  Bind b(Bind::standard);
  b.open();
  b.SwitchToSafe();
  b.close();
}
```

6. dans le cas de l'implémentation pointeur, chaque pointeur de fonction étant local à un objet, chaque instance de l'objet pourrait choisir une implémentation différente pour chacune de ses méthodes.