Des structures de données évoluées

1. Piles
2. Exceptions
3. Listes comme objets
4. Arbres (ou composites)
5. Visiteurs

Piles

Les opérations d'une pile sont

* `void push(element)` pour empiler;
* `element top()` pour récupérer l'élément en haut de pile;
* `element pop()` supprime et retourne l'élément en haut de pile;

Dans une structure exogène, seul un pointeur vers l'élément est stocké, dans une structure endogène, il y a copie. Le comportement du constructeur par recopie dépend de cette approche.

La version la plus simple

```
class Pile {
  int sp;
  char* K[10];
public :
  Pile() : sp(0) {}
  void push(char* x) { K[sp++] = x; }
  char* top() { return  K[sp - 1]; }
  char* pop() { return  K[--sp]; }
};
```

La valeur de sp est l'indice de la première place disponible dans K.

```
void main() {
  Pile P;
  char* x; char* y;

  P.push("Hello world !");
  P.push("Bonjour, monde !");
  x = P.top();
  cout << x << endl;
  y = P.pop();
  x = P.pop();
  cout << x << ' ' << y << endl;
}
```

Avec le résultat

```
Bonjour, monde !
Hello world ! Bonjour, monde !
```

L'ajout et le retrait d'un élément s'écrivent parfois en surchargeant les opérateurs de flots << et >> .
On les surcharge en méthodes:

```
Pile& Pile::operator<<(char* x) {
  push(x);
  return *this;
}
Pile& Pile::operator>>(char*& x) {
  x = pop();
  return *this;
}
```

On écrit alors:

```
void main() {
  Pile P;
  char* x; char* y;

  P << "Hello world !" << "Bonjour, monde !";
  P >> x >> y;
  cout << x << ' ' << y << endl;
  //Affiche: Bonjour, monde ! Hello world !
```

Pile par liste de cellules

On réalise une pile par une liste chaînée de cellules

```
class Pile {
 public :
  Pile(): sp(NULL) {};
  void push(string x) { sp = new Maillon(x, sp); }
  bool isEmpty() { return (!sp); }
  string top() { return sp -> cont; }
  string pop() { string x = sp -> cont; Maillon* t = sp; sp = sp -> next;
   delete t; return x; }
  void imprimer() { for (Maillon* t = sp; t; t = t -> next)
                  cout << t -> cont <<" ";  cout << endl;  }
 private :
      class Maillon {  public:
        Maillon* next;    string cont;
        Maillon(string c, Maillon* n = NULL) : cont(c), next(n) {}
      };
  Maillon* sp;
};
```

```
void main() {
  Pile p;
  p.push("Bonjour, monde");
  p.imprimer();
  p.push("Hello, world");
  p.imprimer(); //etc
}
```

Question : Quelle implantation choisir ?

Réponse : aucune des deux, mais une classe abstraite, représentant les services que rend une pile:

```
class PileAbstraite {
 public :
  virtual void push(string x) =0;
  virtual bool isEmpty() =0;
  virtual string top() =0;
  virtual string pop() =0;
  virtual void imprimer() =0;
};
```

utilisée soit avec

```
void main() {
  PileAbstraite* p = new Pile(); // PileAbstraite& p = Pile();
  // le choix de la classe Pile concrète étant retardé
  // jusqu'au tests de rapidité
  p -> push("Bonjour, monde");
  p -> imprimer();
  p -> push("Hello, world");
  p -> imprimer();
  cout << p -> top();
  string x = p -> pop();
  p -> imprimer();
}
```

Exceptions des Piles

Des cas sans solution :

* Quand la pile est vide, on ne peut faire ni pop() ni top()
* Quand la pile est pleine, on ne peut pas faire de push()

Dans ces cas:

* Ce sont les méthodes de la classe qui doivent détecter ces situations;
* mais c'est le programme appelant qui dont en tirer les conséquences.

Les méthodes doivent signaler les débordements.

Les exceptions constituent le moyen privilégié de gestion d'erreurs, mais servent aussi comme une structure de contrôle générale.

Exceptions

Les exceptions sont un mécanisme de déroutement conditionnel.

* Une instruction `throw` lève une exception;
* Un bloc `try ... catch` capte l'exception.

```
void f(int i) {
          if (i)       throw "Help!";
          cout <<"Ok!\n";
}

void main() {
  try {
    f(0);
    f(1);
    f(0);
  }
  catch(char *) {
    cout << "A l'aide !\n";
  }
}
```

Compilé (parfois avec l'option `-fhandle-exceptions`), le résultat est

```
Ok!
A l'aide !
```

Trois acteurs interviennent dans le mécanisme des exceptions:

* on lance ou lève une exception par une instruction formée de throw suivi d'une expression;
* on effectue une exécution conditionnée d'une suite d'instructions dans un bloc commençant par try;
* on traite ou gère une exception par une clause catch de capture. Les clauses de capture suivent physiquement le groupe `try`.

Les exceptions sont levées, directement ou indirectement, dans le bloc `try`.

Le bloc `try` est suivi d'une ou plusieurs clauses `catch`. Chacune est un gestionnaire d'exception. Un gestionnaire est paramétré par un type (suivi éventuellement d'un argument).

La séquence d'exécution est la suivante:

* si aucune exception n'est levée dans le bloc try, le contrôle passe à l'instruction suivant les gestionnaires d'exception;
* si une exception est levée, le bloc try est quitté, on n'y revient pas, et le contrôle passe au premier gestionnaire rencontré et dont le type s'``accorde'' à l'argument du throw.
* après exécution du gestionnaire, le contrôle passe à l'instruction suivant le dernier gestionnaire.
* si aucun gestionnaire ne convient, une fonction prédéfinie terminate() est appelée.

Une exception est interceptée en filtrant un type.

A la levée, c'est un objet qui est passé, et non un type.

Une fonction peut (en Java doit) spécifier les exceptions qu'elle est susceptible de lever et qu'elle ne traite pas. Par exemple

```
void push(char *) throw (Overflow);
```

Exemple

Deux types d'exceptions sont définies, comme classes en C++:

```
class Underflow{};
class Overflow{};
```

La classe pile est redéclarée:

```
class Pile {
private:
  int sp;
  char* K[4];
public:
  Pile() : sp(0) {}
  char* top() throw (Underflow);
  char* pop() throw (Underflow);
  void push(char* x) throw (Overflow);
};
```

Les méthodes sont redéfinies:

```
char* Pile::top() throw (Underflow) {
  if (sp == 0) throw Underflow(); // objet de la classe
  return K[sp-1];
}
char* Pile::pop() throw (Underflow){
  if (sp == 0) throw Underflow();
  return K[--sp];
}
void Pile::push(char* x) throw (Overflow) {
  if (sp > 10) throw Overflow();
  K[sp++] = x;
}
```

C'est l'utilisateur de la classe qui capte les messages:

```
void main()
{
  Pile P;
  char c;
  for (;;) {
    cin >> c;
    try {
      switch(c){
      case '+':
        P.push("Hello world!"); // peut lever exception
        break;
      case '-':
        cout << P.pop() << endl;// peut lever exception
      }
    }
    catch (Overflow o)  { cout << "Pile deborde!\n";}
    catch (Underflow u) { cout << "Pile vide!\n";}
  }
}
```

On signal on ne termine pas l'excution.

Listes comme objets

On peut voir les listes

* comme mécanisme de chaînage d'objets;
* comme une réalisation des suites finies d'objets.

Dans le premier cas, une liste est un pointeur vers son premier élément, et le chaînage est réalisé par une classe de ``cellules".

Dans le deuxième cas, on a ``réifié" la liste en objet. Le mécanisme de chaînage est secondaire (ce pourrait être un tableau). On s'intéresse ici à

* l'itération;
* le transfert des opérations des éléments sur les listes (le fameux mapcar);
* et on verra comment on obtient gratis des listes de listes (un avant-goût des ``composites").

Listes (exogènes) de pointeurs vers des "objets" (ici une classe d'entiers).

```
class Entiers {
protected:
  int t;
public:
  Entiers(int t = 0) : t(t) {}
  virtual void show() { cout <<' '<< t ;}
};
```

Une liste contient un pointeur vers une première cellule, un pointeur vers la dernière, et un entier contenant la taille.

La classe des maillons est locale à la classe des listes.

```
class Liste {
protected:
  class Maillon {
  public:
    Maillon* suiv;
    Maillon* prec;
    Entiers*   d;
    Maillon(Maillon* p, Maillon* s, Entiers* d)
      : suiv(s), prec(), d(d) {}
   };
  Maillon* prem;
  Maillon* dern;
  int taille;
...
};
```

Constructeur par défaut:

```
Liste::Liste() : prem(NULL), dern(NULL), taille(0) {}
```

Deux méthodes d'interrogation naturelles:

```
  int getTaille() const { return taille;}
  int isEmpty() const { return taille==0;}
```

On manipule par des méthodes usuelles:

```
  void addHead(Entiers* n);
  void addTail(Entiers* n);
  Entiers* getHead();
  Entiers* removeHead();
```

Par exemple:

```
void Liste::addHead(Entiers* n) {
  Maillon* nouv = new Maillon (NULL, prem, n);
  if (prem != NULL)
    prem -> prec = nouv;
  else // liste vide
    dern = nouv;
  prem = nouv;
  taille++;
}
```

Noter que `removeHead`() supprime le maillon et retourne l'adresse de l'Entiers.

L'impression se fait en appelant la méthode adéquate pour chaque objet de la liste:

```
void Liste::show() {
  cout <<"(";
  for (Maillon* p = prem; p; p = p -> suiv)
    p -> d -> show();
  cout <<")";
}
```

Exemple:

```
  Liste a;

 a.addHead(new Entiers(2));
  a.addTail(new Entiers(5));
  a.addTail(new Entiers(6));
  cout << "Taille : " << a.getTaille() << endl; // 3
  a.show();cout << endl; // ( 2 5 6)
```

Listes récursives

Une liste récursive est une liste dont les éléments peuvent être des listes, comme
(4(1 3) 2 5 6)

Ceci est réalisé en déclarant qu'une liste est un cas particulier d'un Entiers.

```
class Liste : public Entiers {
...
};
```

Rien ne doit être modifié dans les fonctions précédentes.
En fait, le constructeur de la classe Liste fait appel au constructeur par défaut de la classe Entiers (qui doit être défini).

```
Liste a, b, c;
a.addHead(new Objet(2)); // (2)
a.addTail(new Objet(5)); // (2 5)
b.addHead(new Objet(7));
b.addHead(new Objet(8)); // (8 7)
c.addHead(new Objet(3));
c.addHead(&b);           // ((8 7) 3)
c.addHead(new Objet(1)); // (1 (8 7) 3)
a.addHead(&c);
a.addHead(new Objet(4));
a.show(); cout << endl;
```

donne

```
( 4( 1( 8 7) 3) 2 5)
```

Arbres (ou composites)

* Les listes récursives précédentes sont un cas particulier du schéma conceptuel Composite.
* Ce schéma conceptuel décrit une représentation d'informations hiérarchisées.

Exemples d'application:

* Un élément graphique peut être
  - une ligne
  - un rectangle
  - un texte
  - ou une figure qui groupe des éléments graphiques
* Un arbre est
  - une feuille
  - un groupe d'arbres

L'objectif de Composite est de composer des objets en une structure arborescente, en vue de pouvoir traiter de manière uniforme un objet et un groupe d'objet.

Structuration: La hiérarchie des classes comporte

* une classe (abstraite) Composant (élément graphique, sommet)
* des classes dérivées Feuilles (ligne, rectangle, texte, ou feuille d'un arbre)
* une classe dérivée Composite (figure, ou nud)

Avantage : un client manipule les objets en tant que composants.

Une arborescence de cercles et de carrés

Description
* Les sommets forment une class Forme.
* Les feuilles sont soit de la class Carre, ou de la class Cercle.
* Les nuds sont de la class Groupe.

La hiérarchie est formée de

```
class Forme {...}
class Carre : public Forme {...}
class Cercle : public Forme {...}
class Groupe : public Forme {...}
```

Les opérations sont en général virtuelles sur la classe Forme. Ici l'affichage et le calcul de l'aire:

```
class Forme {
  public:
    virtual void show() = 0;
    virtual float getAire() = 0;
};
```

La classe Groupe a un mécanisme de parcours. Ici un simple tableau de taille arbitrairement fixée à 12.

```
class Groupe : public Forme {
    Forme* element[12];
    int nb;
  public :
    Groupe() : nb(0) {}
    void add(Forme* f) { element[nb++]= f; }
    float getAire();
    void show();
};
```

Il est plus approprié d'utiliser un conteneur de la STL:

```
#include <vector>
class Groupe : public Forme {
    std::vector<Forme*> element;
  public :
    Groupe() {}
    void add(Forme* f) { element.push_back(f); }
    float getAire();
    void show();

};
```

Les opérations sur un groupe font appel aux opérations sur les éléments:

```
float Groupe::getAire() {
  float s = 0;
  for (int i = 0; i < nb ; i++)
    s += element[i] -> getAire();
  return s;
}
void Groupe::show() {
  cout << "Forme (\n";
  for (int i = 0; i < nb ; i++)
   element[i] -> show();
  cout << ")\n";
}
```

Avec un vecteur, on utilise la méthode `element.size()` à la place de l'entier nb.

Quelques exemples d'appels

```
main() {
  Cercle * c; Carre * r; Groupe g;
  c = new Cercle(5); r = new Carre(3);
  c -> show(); r -> show();
  g.add(c); g.add(r); g.show();
  Groupe * h = new Groupe;
  h -> add(new Carre(6));
  g.add(h); g.show();
  cout << "Aire totale = " << g.getAire() << endl;
}
```

Résultat:

```
Cercle 5 78.5397
Carre 3 9
Forme (
   Cercle 5 78.5397
   Carre 3 9
)
Forme (
   Cercle 5 78.5397
   Carre 3 9
   Forme (
      Carre 6 36
   )
)
Aire totale = 123.54
```

Un annuaire

L'annuaire associe des noms et des numéros de téléphone. Ces couples sont associés aux feuilles d'un arbre. Les nuds contiennent juste le nom de la plus grande feuille du sous-arbre gauche.

Il y a donc une classe Sommet dont dérivent une classe Noeud et une classe Feuille. L'arbre est représenté par la classe Arbre. Voici les définitions

```
class Sommet {
 protected:
  string nom;
 public:
  Sommet(string nom);
  virtual void show(int d = 0) =0;
  virtual long findTel(string nom) =0;
  virtual Sommet* add(string nom, long tel) =0;
};

class Noeud : public Sommet {
  Sommet *gauche, *droit;
 public:
  Noeud(string nom, Sommet *gauche, Sommet*droit);
  Sommet* add(string n, long tel);
  void show(int d = 0);
  long findTel(string n);
};
```

```
class Feuille : public Sommet {
  long tel;
 public :
  Feuille(string nom, long tel);
  void show(int d = 0);
  Sommet* add(string n, long tel);
  long findTel(string n);
};

class Arbre {
  Sommet* racine;
 public:
  Arbre();
  void show();
  void add(string nom, long tel);
  long findTel(string nom);
};
```

On s'en sert par exemple comme suit

```
main() {
  Arbre a;
  a.add("Pierre", 1234);
  a.add("Paul", 1301);
  a.add("Alex", 1111);
  a.add("Vero", 1700);  a.show();
  cout << a.findTel("Pierre") << endl;
  cout << a.findTel("Marie") << endl;
}
```

Résultat:

```
Paul
   Alex
      Alex = 1111
      Paul = 1301
   Pierre
      Pierre = 1234
      Vero = 1700
1234
-1
```

D'autres modèles d'arbre existent.

Visiteurs

Le polymorphisme en C++ (et en Java) est limité: il ne s'applique qu'à l'objet appelant (le -ième argument). Sur les autres arguments s'applique la surcharge. En Smalltalk, le polymorphisme est total.

Reprenons l'évaluation des expressions polonaises postfixées.

* La classe Lexeme a les deux classes dérivées Entier et Operateur.
* L'évaluation se fait dans une boucle :

```
void main() {
  Pile p; int valeur;
  Lexeme* lex = makeLexeme();
  for (; lex != 0; lex = makeLexeme())
    lex -> eat(p);
  ...
}
```

* Logiquement, c'est la pile qui mange le lexème. On aurait dû écrire p.eat(lex), avec deux méthodes ``dérivées''

```
eat(Operateur*)
eat(Entier*)
```

Or, le compilateur demande une méthode de signature

```
eat(Lex*)
```

que l'on est bien incapable de fournir. Tout ceci parce que le polymorphisme ne s'applique pas à l'argument de eat().

Le schéma conceptuel des Visiteurs sert à remédier à cela par l'inversion des arguments.

Dans ce schéma

* la classe (abstraite) Lexeme a une méthode (virtuelle) traditionnellement appelée accept(Pile).
* Chaque classe dérivée (concrète) implémente cette méthode, par inversion des arguments:

```
void Operateur::accept(Pile& p) {   p.eat(this);}
void Entier::accept(Pile& p) {  p.eat(this); }
```

* La classe Pile implémente deux méthodes

```
void Pile::eat(Entier* e) {
  push(e->val);
}
void Pile::eat(Operateur* o) {
  int a = pop();
  int b = pop();
  int c = o->apply(b, a);
  push(c);}
```

L'évaluateur envoie le ``message''

```
lex -> accept(p);
```

Si lex pointe sur un Operateur o, ceci devient

```
  o -> accept(p);
```

et donc

```
 p.eat(o);
```

et la méthode `Pile::eat(Operateur*)` s'applique.

Le tour est joué !

Copie et affectation

1. Constructeurs
2. Constructeur de copie
3. Affectation
4. Conversions