Legal Notice

Université Paris XII - IUT
Route forestière Hurtault
F-77300 Fontainebleau
cegielski@univ-paris12.fr

Vous avez reçu une initiation à la programmation structurée et aux structures de données classiques (tableaux, structures ou enregistrements, fichiers, ...) en illustrant les propos à l'aide du langage C. Vous avez également étudié la programmation orientée objet en illustrant les propos à l'aide du langage C++ (ou du langage Java).

Nous allons maintenant voir l'intérêt des structures de données abstraites et étudier les plus importantes d'entre elles en voyant comment les implémenter et des utilisations classiques de celles-ci.

RÉFÉRENCES

Table des matières

Table des figures

Chapitre 2

Les piles

2.1 Notion

Une **pile** (*stack* en anglais) est une structure de données (linéaire) dans laquelle l'accès est restreint à l'élément le plus récemment inséré. C'est dans ce sens que l'on parle d'une pile d'assiettes ou d'une pile de journaux.

Insérer un élément dans une pile s'appelle **empiler** un élément (*to push* en anglais). Retirer un élément d'une pile (non vide) s'appelle **dépiler** un élément (*to pop* en anglais). On voit l'intérêt d'une fonction booléenne pour savoir si une pile est vide ou non.

2.2 Implémentation

2.2.1 Implémentation en langage C++

Un langage orienté objet tel que le langage C++ est bien adapté à l'implémentation des structures de données abstraites. C'est donc par lui que nous allons commencer.

Écrivons un programme en langage C++ qui implémente la notion de pile de caractères comme structure auto-référente et qui teste celle-ci en empilant les lettres qui constituent le mot 'Bonjour' puis qui dépile jusqu'à ce que la pile soit vide en affichant le contenu de celle-ci.

Vous devriez trouver un programme à peu près de la forme suivante :

```
// pile.cpp
#include <iostream.h>

// Element d'une pile

class StackElt

{
  // attribut
  char val;
  class StackElt *suivant;

  // methodes
public:
  StackElt();
  StackElt(char a);
  StackElt(char a, StackElt *);
  char getVal();
  StackElt * getSuivant();
};

StackElt::StackElt()
{
  val = '\0';
  suivant = NULL;
}

StackElt::StackElt(char a)
{
  val = a;
  suivant = NULL;
}

StackElt::StackElt(char a, StackElt *s)
{
  val = a;
  suivant = s;
}

char StackElt::getVal()
{
  return val;
}

StackElt * StackElt::getSuivant()
{
  return suivant;
}
```

```
// Pile

class Stack
{
  // attributs
  class StackElt *dessus;

  // methodes
public:
  Stack();
  void push(char a);
  char pop();
  int IsEmpty();
};

Stack::Stack()
{
  dessus = NULL;
}

void Stack::push(char a)
{
  StackElt *ptr;
  ptr = new StackElt(a,dessus);
  dessus = ptr;
}

int Stack::IsEmpty()
{
  if (dessus==NULL) return 1;
  else return 0;
}

char Stack::pop()
{
  char a;
  if (IsEmpty()) return '\0';
  else
    {
      a = dessus->getVal();
      dessus = dessus->getSuivant();
      return a;
    }
}

/* Test de la classe pile */

void main(void)
{
  Stack s;
  char c;

  s.push('r');
  s.push('u');
  s.push('o');
  s.push('j');
  s.push('n');
  s.push('o');
  s.push('B');

  while(!(s.IsEmpty()))
```

```
  {
    c = s.pop();
    cout << c;
  }

  cout << '\n';
}
```

dont l'exécution donne :

```
# g++ pile.cpp
# ./a.out
Bonjour
#
```

2.2.2 Implémentation en langage C

Un langage orienté objet tel que le langage C++ est bien adapté au test des structures de données abstraites mais, malheureusement, ne donne pas lieu à un code exécutable suffisamment efficace pour qu'on puisse l'utiliser pour les logiciels critiques tels que les systèmes d'exploitation.

Une fois le prototype bien au point dans un langage orienté objet, on a donc intérêt à le réécrire dans un langage qui donne lieu à un code exécutable plus efficace.

L'analogue de notre programme ci-dessus en langage C sera quelque chose comme ce qui suit :

```
/* pile.c */

#include <stdio.h>
#include <stdlib.h>

/* Element d'une pile */

struct StackElt
{
  char val;
  struct StackElt *suivant;
};

char getVal(struct StackElt se)
{
  return se.val;
}

struct StackElt * getSuivant(struct StackElt se)
{
  return se.suivant;
}

void init(struct StackElt * se, char a, struct StackElt * suiv)
{
  se->val = a;
  se->suivant = suiv;
}

/* Pile */

struct Stack
{
```

```
  struct StackElt *dessus;
};

void push(struct Stack * s, char a)
{
  struct StackElt *ptr;
  char c;

  ptr = (struct StackElt *) malloc(sizeof(struct StackElt));
  init(ptr, a, s->dessus);
  s->dessus = ptr;
}

int IsEmpty(struct Stack s)
{
  if (s.dessus==NULL) return 1;
  else return 0;
}

char pop(struct Stack * s)
{
  char a;
  if (IsEmpty(*s)) return '\0';
  else
    {
      a = getVal(*(s->dessus));
      s->dessus = getSuivant(* (s->dessus));
      return a;
    }
}

void initStack(struct Stack *s)
{
  s->dessus = NULL;
}


/* Test de la classe pile */

void main(void)
{
  struct Stack *s;
  char c;

  s = (struct Stack *) malloc(sizeof(struct Stack));
  initStack(s);
  push(s, 'r');
  push(s, 'u');
  push(s, 'o');
  push(s, 'j');
  push(s, 'n');
  push(s, 'o');
  push(s, 'B');

  while(!(IsEmpty(*s)))
  {
    c = pop(s);
    putchar(c);
  }
  putchar('\n');
}
```

dont l'exécution donne :

```
# gcc pile.c
pile.c: In function ‘main’:
pile.c:74: warning: return type of ‘main’ is not ‘int’
# ./a.out
Bonjour
#
```

Dans la mesure où nous traduisons ce qui a été conç en langage C++, nous obtenons un code propre. N'oublions pas cependant les erreurs que nous aurions pu commettre si nous avions écrit ce programme directement en langage C :

- les attributs des structures `struct StackElt` et `struct Stack` n'étant pas privés, nous aurions pu avoir la tentation de les utiliser directement sans passer par des méthodes adaptées sur ces structures ;
- ces méthodes adaptées, telle que la méthode `getVal()` de la structure `struct StackElt`, ne sont pas vraiment associées aux structures correspondantes ; elles apparaissent comme des fonctions ordinaires ;
- il n'y a pas de constructeur mais une méthode `init()`.

2.2.3 Implémentation en langage C orienté objet

Il existe une façon en langage C d'émuler l'orienté objet avec des méthodes associées aux structures. Réécrivons notre dernier programme de cette façon :

```
/* pile00.c */

#include <stdio.h>
#include <stdlib.h>

/* Element d'une pile */

struct StackElt
{
/* attribut */
  char val;
  struct StackElt *suivant;

/* methodes */
  char (*getVal)(struct StackElt *);
  struct StackElt * (*getSuiv)(struct StackElt *);
};

char getValeur(struct StackElt * se)
{
  return se->val;
}

struct StackElt * getSuivant(struct StackElt * se)
{
  return se->suivant;
}

void init(struct StackElt * se, char a, struct StackElt * suiv)
{
  se->val = a;
  se->suivant = suiv;

  se->getVal = &getValeur;
```

```
  se->getSuiv = &getSuivant;
}

/* Pile */

struct Stack
{
  /* attribut */
  struct StackElt *dessus;

  /* methodes */
  void (*push)(struct Stack *, char a);
  char (*pop)(struct Stack *);
  int (*IsEmpty)(struct Stack *);
};

void empile(struct Stack * s, char a)
{
  struct StackElt *ptr;
  char c;

  ptr = (struct StackElt *) malloc(sizeof(struct StackElt));
  init(ptr, a, s->dessus);
  s->dessus = ptr;
}

int estVide(struct Stack * s)
{
  if (s->dessus==NULL) return 1;
  else return 0;
}

char depile(struct Stack * s)
{
  char a;
  if (s->IsEmpty(s)) return '\0';
  else
    {
      a = (s->dessus)->getVal(s->dessus);
      s->dessus = (s->dessus)->getSuiv(s->dessus);
      return a;
    }
}

void initStack(struct Stack *s)
{
  s->dessus = NULL;

  s->push = &empile;
  s->pop = &depile;
  s->IsEmpty = &estVide;
}


/* Test de la classe pile */

void main(void)
{
  struct Stack *s;
  char c;

  s = (struct Stack *) malloc(sizeof(struct Stack));
```

```
  initStack(s);
  s->push(s, 'r');
  s->push(s, 'u');
  s->push(s, 'o');
  s->push(s, 'j');
  s->push(s, 'n');
  s->push(s, 'o');
  s->push(s, 'B');

  while(!(s->IsEmpty(s)))
  {
    c = s->pop(s);
    putchar(c);
  }
  putchar('\n');
}
```

dont ni la compilation, ni l'exécution n'indiquent rien de spécial :

```
# gcc pile00.c
pile00.c: In function ‘main’:
pile00.c:92: warning: return type of ‘main’ is not ‘int’
# ./a.out
Bonjour
#
```

Commentons ce programme :

- Le champ d'une structure, par exemple le champ `getVal` de la structure `struct StackElt`, peut être d'un type de fonction dont il faut déclarer le type de retour et le type des arguments. La syntaxe est la suivante :

  ```
  type (*nom)(typesArguments);
  ```

  pour le champ `nom`. Ceci nous permet d'obtenir des attributs et des méthodes.
- Un champ de type fonction d'une entité structurée doit être initialisé de la façon suivante :

  ```
  nom = &fonction;
  ```

  où `fonction` est une fonction définie (ou déclarée) précédemment.
- Il n'y a ni constructeur, ni destructeur (avec des noms prédéfinis) pour une structure en langage C. On remplacera un constructeur par une fonction d'initialisation, qui est une fonction ordinaire, sans oublier de commencer par allouer un emplacement mémoire pour le pointeur sur cette structure.
- Cela peut sembler assez sophistiqué de faire appel aux méthodes et non directement aux fonctions associées. C'est vrai dans notre cas particulier où nous n'avons fait intervenir qu'une seule instantiation de nos pseudo-classes.

  Dans le cas où il y a plusieurs instantiations, on peut utiliser des fonctions différentes pour des instantiations différentes et raisonner de façon générale avec la méthode. C'est le cas pour la conception des systèmes d'exploitation, par exemple. On définit un système virtuel de fichiers comme pseudo-classe. Pour chaque type de systèmes de fichiers, on instantie, par exemple, la fonction `read()` par une fonction adaptée à ce type.

2.3 Application 1 : expressions bien parenthésées

2.3.1 Le problème

Considérons une expression utilisant des parenthèses telle que :

```
(1 + [2*(3 - 4)] + 5)/6
```

Cette expression utilise deux types de parenthèses : le couple des parenthèses ouvrante '(' et fermantes ')' et celui des crochets ouvrant '[' et fermant ']'. Elle est *bien parenthésée* dans la mesure où une paire de parenthèses de même type (c'est-à-dire parenthèse ou crochet) ne se chevauche pas avec une paire de parenthèses d'un autre type.
Par contre l'expression :

```
([)]
```

n'est pas bien parenthésée dans la mesure où les paires de parenthèses se chevauchent : la parenthèse ouvrante apparaît avant le crochet fermant.

Les compilateurs, en particulier, vérifient le bon parenthésage, pour les expressions algébriques certes, mais aussi pour les commentaires ou pour les blocs.

2.3.2 Vérification du bon parenthésage

Le bon parenthésage d'une chaîne de caractères peut être vérifié grâce à une pile (de caractères, en fait une pile de parenthèses) :

> Instantier une pile vide. Lire les caractères jusqu'à la fin de la chaîne de caractères. Si le caractère est une parenthèse ouvrante (d'un type quelconque), l'empiler. Si le caractère est une parenthèse fermante et si la pile est vide, afficher une erreur. Sinon dépiler. Si le symbole obtenu n'est pas une parenthèse ouvrante du même type que la parenthèse fermante, afficher une erreur. À la fin de la lecture, si la pile n'est pas vide, afficher une erreur.

Remarquons que dans le cas où une "parenthèse" peut occuper plusieurs caractères (par exemple dans le cas de '/*' et '*/' pour les commentaires en langage C), on parle de **token** au lieu de caractère.

2.3.3 Implémentation en langage C++

Le programme suivant, écrit en langage C++, demande une chaîne de caractères (de longueur inférieure à 100) et en vérifie le bon parenthésage, uniquement pour le couple de parenthèses '(' et ')' :

```
// parentheses.cpp

#include <iostream.h>
#include <stdio.h>
#include <string.h>

// Element d'une pile

class StackElt
```

```
{
  // attribut
  char val;
  class StackElt *suivant;

  // methodes
public:
  StackElt();
  StackElt(char a);
  StackElt(char a, StackElt *);
  char getVal();
  StackElt * getSuivant();
};

StackElt::StackElt()
{
  val = '\0';
  suivant = NULL;
}

StackElt::StackElt(char a)
{
  val = a;
  suivant = NULL;
}

StackElt::StackElt(char a, StackElt *s)
{
  val = a;
  suivant = s;
}

char StackElt::getVal()
{
  return val;
}

StackElt * StackElt::getSuivant()
{
  return suivant;
}

// Pile

class Stack
{
  // attributs
  class StackElt *dessus;

  // methodes
public:
  Stack();
  void push(char a);
  char pop();
  int IsEmpty();
};

Stack::Stack()
{
  dessus = NULL;
}
```

```
void Stack::push(char a)
{
  StackElt *ptr;
  ptr = new StackElt(a,dessus);
  dessus = ptr;
}

int Stack::IsEmpty()
{
  if (dessus==NULL) return 1;
  else return 0;
}

char Stack::pop()
{
  char a;
  if (IsEmpty()) return '\0';
  else
    {
      a = dessus->getVal();
      dessus = dessus->getSuivant();
      return a;
    }
}

// Test des parentheses

void wp(char *mot)
{
  Stack s;
  char c, d;
  int n, i, test = 1;

  n = strlen(mot);
  i = 0;
  while((i < n) && test)
    {
      c = mot[i];
      if (c == '(') s.push(c);
      if (c == ')')
{
  if (s.IsEmpty()) test = 0;
  else
            {
            d = s.pop();
    if (d != '(') test = 0;
    }
}
      i++;
    }
  if (!s.IsEmpty()) test = 0;

  if (test) cout << "Expression bien parenthse" << '\n';
  else cout << "Expression mal parenthse" << '\n';
}

// Essai

void main(void)
{
  char mot[100];
```

```
  cout << "Entrez une expression parenthse : ";
  gets(mot);
  wp(mot);
}
```

2.3.4 Exercices

Exercice 1.- Implémenter l'algorithme vu ci-dessus en langage C.

Exercice 2.- Implémenter l'algorithme vu ci-dessus en langage C orienté objet.

Exercice 3.- Implémenter l'algorithme vu ci-dessus en langage Java.

Exercice 4.- Implémenter l'algorithme vu ci-dessus en prenant en compte les paires de parenthèses '('–')', '['–']' et '{'–'}' dans les divers langages de programmation.

Exercice 5.- Implémenter l'algorithme vu ci-dessus en prenant en compte les paires de parenthèses '('–')', '['–']', '{'–'}' et '/*'–'*/' dans les divers langages de programmation.

2.4 Application 2 : évaluation des expressions algébriques

2.4.1 Le problème

L'évaluation des expressions algébriques demande un petit peu d'astuce.
Pour une expression algébrique simple telle que :

```
1 + 2
```

il n'y a pas de problème. Mais cela devient un petit peu plus compliqué pour des expressions telles que :

```
1 + 2*3
10 - 4 - 3
```

2.4.2 Machines postfixe

Expression postfixe

Le logicien polonais Lesniewski a introduit la notion de **notation postfixe** dans les années 1930 pour montrer que l'on n'avait pas besoin, en théorie, des parenthèses dans les expressions. Ceci diminuait le nombre de symboles primitifs, ce qui était l'un des buts du jeu dans ces années-là.

Au lieu d'écrire :

1 + 2*3

on écrit :

1 2 3 * +

L'évaluation se fait comme ceci : lorsqu'on rencontre un opérande, on le met de côté ; lorsqu'on rencontre un opérateur, on récupère le nombre d'opérandes adéquats (en général deux), les derniers que l'on a mis de côté, on effectue l'opération et on met de côté le résultat.

Dans le cas de notre expression postfixe : on rencontre 1, que l'on met de côté ; on rencontre 2, que l'on met de côté ; on rencontre 3, que l'on met de côté ; on rencontre l'opérateur de multiplication, on récupère 3 et 2, on obtient 6, que l'on met de côté ; on rencontre l'opérateur d'addition, on récupère 6 et 1, d'où le résultat 7.

Bien entendu, "mettre de côté" peut se traduire immédiatement par empiler et "retirer" par dépiler.

Évaluation d'une expression postfixe

Le programme suivant, écrit en langage C++, permet d'évaluer une expression postfixe portant sur les quatre opérations, les constantes entières étant constituées d'un seul chiffre :

```
// postfix.cpp
#include <iostream.h>
#include <stdio.h>
#include <string.h>

// Element d'une pile
```

```
class StackElt

{
  // attribut
  int val;
  class StackElt *suivant;

  // methodes
public:
  StackElt();
  StackElt(int a);
  StackElt(int a, StackElt *);
  int getVal();
  StackElt * getSuivant();
};

StackElt::StackElt()
{
  val = 0;
  suivant = NULL;
}

StackElt::StackElt(int a)
{
  val = a;
  suivant = NULL;
}

StackElt::StackElt(int a, StackElt *s)
{
  val = a;
  suivant = s;
}

int StackElt::getVal()
{
  return val;
}

StackElt * StackElt::getSuivant()
{
  return suivant;
}

// Pile

class Stack
{
  // attributs
  class StackElt *dessus;

  // methodes
public:
  Stack();
  void push(int a);
  int pop();
  int IsEmpty();
};

Stack::Stack()
{
  dessus = NULL;
```

```
}

void Stack::push(int a)
{
  StackElt *ptr;
  ptr = new StackElt(a,dessus);
  dessus = ptr;
}

int Stack::IsEmpty()
{
  if (dessus==NULL) return 1;
  else return 0;
}

int Stack::pop()
{
  int a;
  if (IsEmpty()) return 0;
  else
    {
      a = dessus->getVal();
      dessus = dessus->getSuivant();
      return a;
    }
}

// Evaluation des expressions postfixes

int postEval(char exp[], int *res)
{
  int n, i, test = 1, a, b, c;
  Stack s;

  n = strlen(exp);
  i = 0;
  while ((i < n) && test)
    {
      c = exp[i];
      if (('0' <= c) && (c <= '9')) s.push(c - '0');

      if (c == '+')
{
  if (s.IsEmpty()) test = 0;
  else
    {
      a = s.pop();
      if (s.IsEmpty()) test = 0;
      else
                {
                b = s.pop();
        c = a + b;
        s.push(c);
}
    }
}

      if (c == '-')
{
  if (s.IsEmpty()) test = 0;
  else
    {
```

```
      a = s.pop();
      if (s.IsEmpty()) test = 0;
      else
                {
                b = s.pop();
        c = b - a;
        s.push(c);
}
    }
}

      if (c == '*')
{
  if (s.IsEmpty()) test = 0;
  else
    {
      a = s.pop();
      if (s.IsEmpty()) test = 0;
      else
                {
                b = s.pop();
        c = a*b;
        s.push(c);
}
    }
}

      if (c == '/')
{
  if (s.IsEmpty()) test = 0;
  else
    {
      a = s.pop();
      if (s.IsEmpty()) test = 0;
      else
                {
                b = s.pop();
        c = b/a;
        s.push(c);
}
    }
}

      i++;
    }

  if (!test) return -1;

  if (s.IsEmpty()) return -1;
  a = s.pop();
  if (!s.IsEmpty()) return -1;
  else
     {
       *res = a;
       return 0;
     }
}

// Test de l'evaluation des expressions postfixes

void main(void)
{
```

```
  char exp[100];
  int err, n;

  cout << "Entrez une expression postfixe : ";
  gets(exp);
  err = postEval(exp, &n);

  if (err < 0) cout << "expression non postfixe" << '\n';
  else cout << "resultat = " << n << '\n';
}
```

2.4.3 Conversion infixe à postfixe

L'idée pour évaluer une expression infixe est, dans une première étape, de la convertir en l'expression postfixe correspondante puis, dans une seconde étape, d'évaluer cette expression postfixe.

Algorithme de conversion

On initialise une pile de caractères à vide. On parcourt l'expression infixe de gauche à droite en regardant chaque caractère. On effectue une action suivant la nature de celui-ci :

- *Opérande* : on le reporte tel que dans l'expression destination.
- *Parenthèse ouvrante* : on l'empile.
- *Parenthèse fermante* : on dépile tous les caractères de la pile que l'on reporte dans l'expression destination jusqu'à ce qu'une parenthèse ouvrante soit rencontrée (que l'on ne reporte pas).
- *Opérateur* : dépiler tous les caractères de la pile jusqu'à ce qu'on aperçoit un symbole de priorité moindre ou égale. On reporte les symboles dans l'expression destination et on réempile le symbole.
- *Fin de l'expression source* : on dépile et on reporte dans l'expression destination tous les éléments restants.

Mise en place

Le programme C++ suivant transforme une expression infixe en une expression postfixe en suivant les règles de précédence suivantes : parenthèse et exponentiation au niveau le plus haut, multiplication et division au niveau intermédiaire, addition et soustraction au niveau le plus bas.

```
// infix2postfix.cpp
#include <iostream.h>
#include <stdio.h>
#include <string.h>

// Element d'une pile

class StackElt

{
  // attribut
  char val;
  class StackElt *suivant;

  // methodes
```

```
public:
  StackElt();
  StackElt(char a);
  StackElt(char a, StackElt *);
  char getVal();
  StackElt * getSuivant();
};

StackElt::StackElt()
{
  val = 0;
  suivant = NULL;
}

StackElt::StackElt(char a)
{
  val = a;
  suivant = NULL;
}

StackElt::StackElt(char a, StackElt *s)
{
  val = a;
  suivant = s;
}

char StackElt::getVal()
{
  return val;
}

StackElt * StackElt::getSuivant()
{
  return suivant;
}

// Pile

class Stack
{
  // attributs
  class StackElt *dessus;

  // methodes
public:
  Stack();
  void push(char a);
  char pop();
  char IsEmpty();
};

Stack::Stack()
{
  dessus = NULL;
}

void Stack::push(char a)
{
  StackElt *ptr;
  ptr = new StackElt(a,dessus);
  dessus = ptr;
}
```

```
char Stack::IsEmpty()
{
  if (dessus==NULL) return 1;
  else return 0;
}

char Stack::pop()
{
  char a;
  if (IsEmpty()) return -1;
  else
    {
      a = dessus->getVal();
      dessus = dessus->getSuivant();
      return a;
    }
}

// Conversion infixe vers postfixe

char * inf2post(char exp[])
{
  char post[100];
  Stack s;
  int n, i, j;
  char c, d;

  n = strlen(exp);
  j = 0;
  for (i=0; i < n; i++)
    {
      c = exp[i];
      if (('0' <= c) && (c <= '9'))
{
  post[j] = c;
  j++;
}
      if (c == ')')
{
  if (s.IsEmpty()) return NULL;
  d = s.pop();
  while(d != '(')
    {
      post[j] = d;
      j++;
      if (s.IsEmpty()) return NULL;
      d = s.pop();
    }
}
      if (c == '^' || c == '(') s.push(c);
      if (c == '+' || c == '-')
{
  d = '0';
  while (!s.IsEmpty() && (d != '('))
    {
      d = s.pop();
      if (d != '(')
{
  post[j] = d;
  j++;
}
```

```
      if (d == '(') s.push(d);
    }
  s.push(c);
}
      if (c == '*' || c == '/')
{
  d = '0';
  while (!s.IsEmpty() && (d != '(') && (d != '+') && (d != '-'))
    {
      d = s.pop();
      if (d != '(' && (d != '+') && (d != '-'))
{
  post[j] = d;
  j++;
}
      if (d == '(' || (d != '+') || (d != '-')) s.push(d);
    }
  s.push(c);
}
    }
  while (!s.IsEmpty())
    {
      post[j] = s.pop();
      j++;
    }
  post[j] = '\0';
  puts(post);
  return post;
}

// Test de l'valuation des expressions postfixes

void main(void)
{
  char *exp, *post;
  int n;

  exp = new char[100];
  post = new char[100];
  cout << "Entrez une expression infixe : ";
  gets(exp);
  post = inf2post(exp);

  cout << "expression postfixe associee : ";
  puts(post);
  cout << '\n';
}
```

2.5 Exercices

Exercice 1.- Écrire un patron de classe pile dont le type des éléments est une variable de type.

Exercice 2.- Écrire un programme pour les expressions bien parenthésées utilisant trois types de parenthèses: '(' et ')', '[' et ']', '' et ''.