TD n° 7 (Correction)

Types abstraits : Ensemble

Dans ce TD, on suppose déjà réalisé le type abstrait **Ensemble**. Les éléments d'un **Ensemble** sont de type **Objet** supposé lui aussi défini par ailleurs. Par exemple, le type **Objet** pourrait être le type **int**, ou une structure contenant deux entiers... ou n'importe quoi d'autre.

Nous utilisons une version du type abstrait **Ensemble** la plus simple possible et qui ne possède que les constructeurs et les opérateurs suivants :

Constructeur

- void `ensVide`(Ensemble &e)
  // Crée l'ensemble vide e

Opérateurs d'accès

- void `ajoute`(Ensemble &e, Objet o)
  // l'objet o donné est ajouté à l'ensemble e s'il n'y est pas, sinon rien n'est changé
- void `retire`(Ensemble &e, Objet o)
  // l'objet o donné est retiré de l'ensemble e s'il y est, sinon rien n'est changé
- objet `element`(Ensemble e)
  // retourne un objet quelconque de l'ensemble e si celui-ci n'est pas vide ;
  // comportement non spécifié si l'ensemble e est vide

Opérateurs de test

- bool `estVide`(Ensemble e)
  // retourne *true* si e est l'ensemble vide, *false* sinon
- bool `estDans`(Ensemble e, Objet o)
  // retourne *true* si l'objet o est dans l'ensemble e, *false* sinon

À partir de ces 6 primitives, on souhaite programmer d'autres constructeurs et d'autres fonctions plus élaborées, que l'on utilisera finalement pour vérifier une propriété ensembliste.

Remarque : le fait d'être obligé de passer par ces six primitives impose de programmer dans un style qui n'est pas très efficace : par exemple pour calculer le cardinal d'un ensemble, on doit sortir les éléments un par un de l'ensemble, en incrémentant simultanément un compteur. De même, pour réaliser l'union ou l'intersection. Un vrai type abstrait sera en général composé de plus de 6 primitives, pour améliorer les performances.

Nous proposons ce jeu restreint, pour que le td soit court, tout en illustrant comment l'utilisation d'un type abstrait permet la répartition des tâches : en aval, une personne peut réaliser les 6 primitives en C++ par exemple, tandis qu'en amont, une autre personne programmera ce td. L'exploitation conjointe des deux parties réalisées est ensuite immédiate puisque l'interface entre ces deux parties a été définie par l'explicitation du type abstrait, i.e. les signatures des constructeurs et des opérateurs du type.

Exercice 1.

1. Réaliser une fonction `cardinal` qui retourne le cardinal d'un ensemble passé en paramètre.

   Correction : *Faire remarquer aux étudiants le passage par valeur. Il est ici très important: On travaille sur une copie de l'ensemble passé en paramêtre pour ne pas modifier l'original. Cette remarque reste vraie dans les questions suivantes.*

```
int cardinal(Ensemble e) {
  int res = 0; Objet el;
  while (not estVide(e)) {
    el = element(e);
    retire(e, el);
    res++;
  }
  return res;
}
```

2. Réaliser la fonction `inclus` qui renvoie `true` si un premier ensemble d'objets donné est inclus dans un deuxième ensemble d'objets donné, `false` sinon.

   Correction :

```
bool inclus(Ensemble e, Ensemble f) {
  Objet el;
  while (not estVide(e)) {
    el = element(e);
    if (not estDans(f, el)) return false;
    retire(e, el);
  }
  return true;
}
```

3. Réaliser la fonction `égal` qui renvoie `true` si deux ensembles d'objets donnés sont égaux, `false` sinon.

   Correction :

```
bool egal(Ensemble e, Ensemble f) {
  return inclus(e, f) and inclus(f, e);
}
```

4. Réaliser la fonction `unionEns` qui retourne l'union de deux ensembles donnés. Les deux ensembles donnés ne doivent pas être modifiés.

   Correction :

```
Ensemble unionEns(Ensemble e, Ensemble f) {
  Ensemble res; Objet el;
  ensVide(res);
  while (not estVide(e)) {
    el = element(e);
    retire(e, el);
    ajoute(res, el);
  }
  while (not estVide(f)) {
    el = element(f);
    retire(f, el);
    ajoute(res, el);
  }
  return res;
}
```

5. Réaliser la fonction `interEns` qui retourne l'intersection de deux ensembles donnés, sans les modifier.

   Correction :

```
Ensemble interEns(Ensemble e, Ensemble f) {
  Ensemble res; Objet el;
  ensVide(res);
  while (not estVide(e)) {
    el = element(e);
    retire(e, el);
    if (estDans(f, el)) ajoute(res, el);
  }
  return res;
}
```

6. En supposant qu'il existe une procédure `ecrireObj`(Objet o), qui permet d'imprimer un objet, réaliser la procédure `ecrireEns` qui imprime tous les éléments d'un ensemble passé en paramètre.

   Correction :

```
void ecrireEns(Ensemble e) {
 Objet el;
 while (not estVide(e)) {
    el = element(e);
    retire(e, el);
    ecrireObj(el);
 }
```

7. En supposant qu'il existe une procédure : `randomObj`(Objet &o) qui renvoie un objet o aléatoire, réaliser la procédure `randomEns` qui crée un ensemble de n éléments, où n est un entier positif.

   Correction :

```
void randomEns (int n, Ensemble &E) {
   objet o; int i = 0;
   ensVide(E);
   while (i <= n) {
      randomObj(o);
      if (estDans(E,o) == false) {
          ajoute(E,o) ; i++;
      }
   }
}
```

8. **Principe d'inclusion-exclusion :** Écrire le programme principal qui vérifie si la propriété ensembliste suivante est vraie : si A, B, C sont des ensembles, alors

$$\begin{aligned} |A \cup B \cup C| = & |A| + |B| + |C| \\ & -|A \cap B| - |A \cap C| - |B \cap C| \\ & +|A \cap B \cap C| \end{aligned}$$

   où $|A|$ représente le nombre cardinal de A.

   On vérifiera expérimentalement cette propriété en la testant sur des ensembles construits de façon aléatoire et dont le cardinal est lui-même aléatoire (compris entre 8 et 12).

   Correction :

```
void test () {
  Ensemble A, B, C;
  \\creer des nombres au hasard entre 8 et 12 et les Ens
  int n1 = random(5)+8;
  randomEns(n1, A);
  n1 = random(5)+8;  randomEns(n1, B);
  n1 = random(5)+8;  randomEns(n1, C);

  n1 = cardinal( unionEns(unionEns(A,B), C) );
  int n2 = cardinal(A) + cardinal(B) + cardinal(C);
  n2 = n2 - cardinal(interEns(A,B) );
  n2 = n2 - cardinal(interEns(A,C) );
  n2 = n2 - cardinal(interEns(B,C) );
  n2 = n2 + cardinal(Inter(inter(A,B), C));
  if (n1==n2) {
      cout << "test reussi" << endl;
   } else { cout << "t rate" << endl; }
}
```

Exercice 2. On s'intéresse maintenant à la réalisation de ce type abstrait. Plusieurs types concrets peuvent être utilisés/choisis. On supposera que les objets sont des entiers et que le nombre d'objets d'un ensemble est borné par une constante `MaxE`.

Réalisez la fonction `estDans` et les procédures `ensVide`, `ajoute` et `retire` avec chacun des types concrets suivants :

Réalisation 1 Dans cette première solution, on ne stocke que des ensembles d'entiers qui sont entre 0 et `MaxE` -1. On peut donc utiliser comme type concret un tableau T de booléens où $T[i]$ vaut `true` si l'entier i est dans l'ensemble et faux sinon. Pour éviter les problèmes spécifiques au passage des tableaux en paramètre, on met le tableau dans une `struct`.

Réalisation 2 On utilise comme type concret une structure contenant un tableau d'entiers et un champ indiquant le nombre d'éléments de l'ensemble.

Réalisation 3 On utilise comme type concret une structure contenant un tableau d'entiers **triés** et un champ indiquant le nombre d'éléments de l'ensemble.

Correction :

Réalisation 1 *On utilise comme type concret un tableau de booléens.*

Note : Si l'on met un tableau `C`, lors d'un passage de paramêtre, il est systématiquement passé par un pointeur vers le premier élément et donc on ne peut pas faire de passage par valeur. De plus, on ne peut pas copier un tableau avec `=`. Si l'on met un tableau dans une `struct`. On n'a plus le problème : la struct se comporte normalement. J'ai expliqué ce point dans le cours.

Remarque pour les enseignants : En `C++` normal, on utilise le type `array` qui est un objet avec les méthodes `size` et autre.

```
const int MaxE = 100;
struct Ensemble {
  bool t[MaxE];
};

bool estDans (Ensemble e, int o){
return e.t[o];
}

void ensVide(Ensemble &e) {
  for (int i = 0; i < MaxE;  i++)
       e.t[i] = false;
```

```
}

void ajoute(Ensemble &e, int o) {
  e.t[o] = true;
}


void retire(Ensemble &e, int o) {
  e.t[o] = false;
}
```

Réalisation 2 *On utilise comme type concret une structure contenant un tableau d'entiers et un champ indiquant le nombre d'éléments de l'ensemble.*

```
const int MaxE = 100;
struct Ensemble {
  int t[MaxE];
  int nb
};

bool estDans (Ensemble e, int o) {
  for (int i = 0; i < e.nb;  i++)
      if (e.t[i] == o) return true;
  return false;
}

void ensVide(Ensemble &e) {
  e.nb = 0;
}


void ajoute(Ensemble &e, int o) {
  if (not estDans (e, o)) {
    if (e.nb >= MaxE) {
      cerr << "Plus de place...";
      exit(1);
    }
    e.t[e.nb] = o;
    e.nb++;
  }
}

void retire(Ensemble &e, int o) {
  int pos = 0;
  while (pos < e.nb and e.t[pos] != o) pos++;
  if (pos != e.nb) {
    for (int i = pos; i < e.nb -1; i++)
      e.t[i] = e.t[i+1];
    e.nb--;
  }
}
```

Réalisation 3 *On utilise comme type concret une structure contenant un tableau d'entiers* **triés** *et un champ indiquant le nombre d'éléments de l'ensemble.*
Par rapport au tableau non trié, le changement est que lorsqu'on cherche si un entier appartient au tableau ou qu'on veut l'y ajouter, dès que dans la boucle while l'élément atteint est plus grand que o, on sait qu'il n'est pas dans le tableau ou que c'est là qu'il faut l'ajouter.

```
bool estDans (Ensemble e, int o) {
  int i=0 ;
  while(i < e.nb AND e.t[i] <= o){
    if (e.t[i] == o) return true;
    i++ ;
  }
  return false;
}

La procedure ensVide ne change pas

void ajoute(Ensemble &e, int o) {
  if (not estDans(e, o) and e.nb < MaxE) {
     int i = 0;
     while( i < e.nb and e.t[i] < o) i++ ;
     // on sort de cette boucle avec l'indice i ou o doit etre range
     if (i != e.nb) {
          for(int j = e.nb; j >= i+1; j--) {
               e.t[j] = e.t[j-1] ;
          }
          e.t[i]= o ;
          e.nb++ ;
     }
}

void retire(Ensemble &e, int o){
    int i = 0;
    while(i < e.nb and e.t[i] != o ) i++ ;
    if (i != e.nb) {                          // sinon, il n'y est pas ?
         for(int j = i; j < e.nb -1; j++) e.t[j] = e.t[j+1] ;
         e.nb-- ;
    }
}
```