université PARIS-SACLAY

Travaux Pratiques de programmation n°9

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Encapsulation et types abstraits

Dans cette séance de travaux pratiques, nous travaillons autour d'un type abstrait `Ensemble`, défini par les méthodes ci-dessous :

```
Ensemble ();                    // Constructeur par défaut
Ensemble (vector<Objet> v);     // Constructeur a partir d'un vecteur
bool ajoute(Objet o);           // Ajoute un objet à un ensemble
bool retire(Objet o);           // Retire un objet à un ensemble
Objet element() const;          // Retourne un objet d'un ensemble
bool estVide() const;           // Teste si l'ensemble est vide
bool contient(Objet o) const;   // Teste si un objet est dans l'ensemble
```

On va travailler en deux temps :

- Dans un premier temps, on se place du point de vue des utilisateurs du type et on va faire des calculs sur les ensembles ;
- Dans un deuxième temps, on implémentera le type `Ensemble`.

Le TP n'est pas à finir pour cette semaine : les exercices 2 à 5 seront repris la semaine prochaine en TP. Par contre l'exercice 1 doit impérativement être fini cettte semaine. Si vous ne l'avez pas fini en TP, il faudra le finir à la maison avant la prochaine séance de TP.

Conventions de nommage

Quand on commence à écrire des programmes avec quelques dizaines de fonctions, pour éviter d'avoir en permanence à se poser des questions comme « Quel est le nom de la fonction qui ... » ou bien « Dans quel ordre dois-je passer les paramètres à la fonction ... », il est utile de suivre quelques conventions. Toutes les entreprises de développement de logiciels, tous les projets de logiciels sérieux fixent ainsi un certain nombre de règles concernant l'indentation du code, le choix des noms des fonctions et l'ordre des paramètres.

Les choix faits dans la convention suivante sont pour la plupart arbitraires, ils sont là seulement pour éviter des hésitations comme par exemple : dois-je écrire «`EstVide`» ou bien «`estVide`» où encore « `est_vide` » ? Rien ne vous interdit de vous en écarter si elles ne vous plaisent pas, mais ne pas suivre de conventions est un bon moyen de perdre bêtement du temps. Conventions choisies dans ce cours :

- Les noms des types et classes commencent par une majuscule (par exemple `Ensemble`).
- On utilise la convention dite `camelCase` pour les noms de fonctions et de méthodes (les mots accolés et commençant par une majuscule). Par exemple `estVide`.

Enfin, Il y a deux sortes de méthodes qui font des calculs :

1. soit on **modifie** l'objet ; dans ce cas le nom de la méthode sera un **verbe conjugué qui décrit l'action** que l'on fait sur l'objet. Par exemple `ajoute(x)` ajoute `x` à l'objet.
2. soit on **retourne un résultat** ; dans ce cas le nom de la méthode sera un **nom qui décrit le résultat de l'action**. Par exemple `union(a, b)` retourne l'union.

Les ensembles en mathématiques

Lors du cours nous avons vu le type `Sac` qui est aussi appelé «multi-ensemble», car un élément peut apparaître plusieurs fois. Au contraire, dans un *ensemble* un élément ne peut pas être dupliqué. Les mathématiciens utilisent les notions suivantes sur les ensembles :

— **Création d'un ensemble en extension**, à partir de la liste des éléments. L'ordre n'importe pas et l'on supprime les doublons. Ainsi les trois ensembles suivants sont tous égaux :

$$\{1, 2, 4, 7\} \qquad \{2, 7, 1, 4\} \qquad \{4, 2, 7, 7, 1, 4\}$$

mais ils sont différents de $\{1, 3, 2, 7, 4\}$ et de $\{1, 3, 2, 7\}$.

— **Appartenance à un ensemble**. L'expression $e \in E$ vaut vrai ou faux selon que e apparaît dans E ou non.

$$2 \in \{2, 7, 1, 4\} \text{ est vrai} \qquad 3 \in \{2, 7, 1, 4\} \text{ est faux}$$

— **Inclusion de deux ensembles**. L'expression $E \subset F$ vaut vrai si tous les éléments de E apparaissent dans F.

$$\{2, 4\} \subset \{2, 7, 1, 4\} \text{ est vrai} \qquad \{2, 5\} \subset \{2, 7, 1, 4\} \text{ est faux}$$

— **Union de deux ensembles :** $E \cup F$. C'est l'ensemble des éléments qui sont soit dans E soit dans F. Bien sûr un élément qui apparaît à la fois dans E et F n'apparaît qu'une fois dans $E \cup F$.

$$\{1, 2, 4, 7\} \cup \{2, 3, 4, 8\} = \{1, 2, 3, 4, 7, 8\}$$

— **Intersection de deux ensembles :** $E \cap F$. C'est l'ensemble des éléments qui sont à la fois dans E et dans F.

$$\{1, 2, 4, 7\} \cap \{2, 3, 4, 8\} = \{2, 4\}$$

Le type ensemble

Dans le premier exercice, on suppose déjà réalisé le type abstrait `Ensemble`. Une implantation est fournie dans deux fichiers `Ensemble.hpp` et `Ensemble.cpp`. Il n'est pas utile que vous compreniez le contenu du fichier `Ensemble.cpp` qui utilise le type ensemble de la bibliothèque standard du `C++`. En revanche, vous pouvez garder à portée de main le fichier `Ensemble.hpp` qui décrit en détail l'interface.

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 tout autre type.

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

- Constructeur par défaut qui construit un ensemble vide.
- Constructeur à partir d'un vecteur (pour les tests).
- `bool ajoute(Objet o)` : l'objet *o* donné est ajouté à l'ensemble s'il n'y est pas, sinon rien n'est changé.
- `bool retire(Objet o)` : l'objet *o* donné est retiré de l'ensemble s'il y est, sinon rien n'est changé
- `Objet element() const` : renvoie un objet quelconque de l'ensemble si celui-ci n'est pas vide ; comportement non spécifié si l'ensemble est vide.
  Attention ! Il n'est pas précisé que la méthode `element` est déterministe. Il est possible que si on l'appelle deux fois, elle donne un élément différent.
- `bool estVide() const` renvoie *true* si l'ensemble est l'ensemble vide, *false* sinon.
- `bool contient(Objet o) const` : renvoie *true* si l'objet *o* est dans l'ensemble, *false* sinon.

À partir de ces 7 fonctions dites primitives, on souhaite programmer d'autres constructeurs et d'autres fonctions plus élaborées, que l'on utilisera finalement pour vérifier une propriété ensembliste.

Remarque 1 : le fait d'être obligé de passer par ces 7 primitives impose de programmer dans un style qui n'est pas très efficace : par exemple pour calculer le cardinal d'un ensemble, on doit retirer 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 7 primitives, pour améliorer les performances.

Remarque 2 : Nous proposons ce jeu restreint de primitives, pour que le TP soit court, tout en illustrant comment l'utilisation d'un type abstrait permet la répartition des tâches : en aval, une personne peut réaliser les 7 primitives en C++ par exemple, tandis qu'en amont, une autre personne programmera ce TP. 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. Dans une bibliothèque réaliste, les fonctions d'inclusion, d'égalité, d'union... ci-dessous seraient aussi réalisées dans la classe.

▸ **Exercice 1. (Utilisateur du type `Ensemble`)**
Dans cet exercice, vous devez compléter le fichier `Ensemble-main.cpp` fourni. Vous ne devez pas modifier les autres fichiers. Un Makefile vous est aussi fourni, il permet de compiler `Ensemble-main.cpp` en tapant simplement la commande `make` dans le terminal.

1. Réaliser une fonction `cardinal` qui renvoie le cardinal (c'est-à-dire le nombre d'éléments) d'un ensemble passé en paramètre. Pour ceci, on fera une boucle qui retire les éléments un à un tant que l'ensemble n'est pas vide. Tester cette fonction. Pour les tests uniquement on pourra considérer que les objets contenus dans les ensembles sont des entiers.
2. Surcharger l'opérateur d'affichage pour les ensembles (on suppose qu'il a déjà été surchargé pour les objets). Tester cette fonction dans le `main`.
3. 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. Tester cette fonction.
4. Surcharger l'opérateur d'égalité pour les ensembles. Tester cette fonction.
5. Réaliser la fonction `unionEns` qui renvoie l'union de deux ensembles donnés. Tester cette fonction.
6. Réaliser la fonction `interEns` qui renvoie l'intersection de deux ensembles donnés. Tester cette fonction.
7. En supposant qu'il existe une fonction `Objet randomObj()` qui renvoie un objet aléatoire, réaliser une fonction `randomEns` qui crée un ensemble de n éléments, où n est un entier positif donné. Tester cette fonction dans le `main`.
8. **Principe d'inclusion-exclusion :**
   La propriété ensembliste suivante est appelée principe d'inclusion-exclusion : 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 cardinal (nombre d'éléments) de A. Utile par exemple en probabilité, on peut la généraliser à un nombre quelconque d'ensembles.

   Dans la fonction `inclsExcls`, 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). Indication : on pourra utiliser la fonction `rand()` de `C++`, qui ne prend pas de paramètre et renvoie un entier aléatoire (potentiellement très grand). Pour ceci, il faut tirer au hasard un nombre entre 0 et $12 - 8 = 4$ inclus et lui ajouter 8. Pour avoir un nombre entre 0 et 4 on peut prendre le reste de la division de `rand()` par 5. Des détails sur le tirage aléatoire vous ont été donnés à la fin du TP2.

   Le cas de test associé à cette fonction consiste à lancer plusieurs fois la fonction et vérifier qu'elle donne bien toujours `true`.

▸ **Exercice 2. (Mise en place pour l'implémentation du type `Ensemble`)**

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 entre 0 et une constante `MaxTaille`.

Les fichiers `Ensemble.hpp` et `Ensemble.cpp` sont en fait des liens vers les fichiers `EnsembleProf.hpp` et `EnsembleProf.cpp` qui contiennent l'implémentation de référence. Nous allons remplacer ces derniers par les fichiers `MonEnsemble.hpp` et `MonEnsemble.cpp` que vous devrez compléter dans la suite de ce TP afin d'obtenir votre propre implémentation du type abstrait `Ensemble`.

Parfois on voudra que `Ensemble.hpp` corresponde à `EnsembleProf.hpp` et parfois on voudra qu'il corresponde à `MonEnsemble.hpp` (cela vous permettra de comparer les résultats que vous obtenez avec votre implémentation à ceux qu'on est censé obtenir). Pour indiquer à quel fichier on veut que `Ensemble.hpp` corresponde, on utilise le *lien* suivant :

— `Ensemble.hpp` vers `EnsembleProf.hpp`

ce qui veux dire que si l'on essaye d'accéder au contenu du fichier `Ensemble.hpp`, on accédera en fait au fichier `EnsembleProf.hpp`. De même pour les `.cpp` correspondants. Pour faire ces liens, on a exécuté les commandes :

```
ln -s EnsembleProf.hpp Ensemble.hpp
ln -s EnsembleProf.cpp Ensemble.cpp
```

La commande «`ln -s`» agit comme une copie.

1. On va maintenant basculer sur votre implémentation à vous (`MonEnsemble.hpp`). Pour cela, supprimer les anciens liens et en faire de nouveaux avec les commandes suivantes :

   ```
   make clean
   rm -f Ensemble.hpp Ensemble.cpp
   ln -s MonEnsemble.hpp Ensemble.hpp
   ln -s MonEnsemble.cpp Ensemble.cpp
   ```

2. Faire `make` puis exécuter. Vous devez constater que le résultat n'est plus celui attendu, puisque vous n'avez pas encore écrit votre implémentation (il est même possible que vous ayez besoin de taper `Ctrl C` pour pouvoir interrompre le programme et reprendre la main sur le terminal). Le but de la suite du TP est de faire votre propre implémentation dans les fichiers fournis `MonEnsemble`. On retrouvera alors le résultat attendu en exécutant le programme.

Dans les exercices suivants, on demande de réaliser le type `Ensemble` avec un type concret particulier.

Vous devez de plus bien tester à chaque fois toutes les fonctions écrites, en complétant le fichier `Ensemble-test.cpp` fourni et en le compilant et exécutant à l'aide du `Makefile` fourni.

▸ **Exercice 3. (Implémentation par tableau de booléens)**

Dans cette première implémentation, on ne stocke que des ensembles d'entiers qui sont entre 0 et `MaxTaille` -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 `false` sinon. Réalisez cette première implémentation dans les fichiers `MonEnsemble.hpp` et `MonEnsemble.cpp`.

▸ **Exercice 4. (Implémentation par vecteur d'objets)**

1. Avant de passer à une nouvelle réalisation, il faut créer deux nouveaux fichiers : `MonEnsemble2.hpp` et `MonEnsemble2.cpp` puis refaire les liens depuis `Ensemble.hpp` et `Ensemble.cpp` comme précédement (4 commandes).
2. N'oubliez pas d'ajouter vos nouveaux fichiers avec `git add`.
3. Dans cette deuxième réalisation on utilise comme type concret une classe contenant un vecteur d'objets. Implémentez-là.

▸ **Exercice 5. (Implémentation par vecteur d'objets trié)**

1. Comme dans l'exercice précédent, créez de nouveaux fichiers, faites les liens et n'oubliez pas `git add`.
2. On utilise comme type concret une structure contenant un vecteur d'entiers **triés**. 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).
3. S'il vous reste du temps, vous pouvez optimiser les recherches grâce à la dichotomie. Voir `https://fr.wikipedia.org/wiki/Recherche_dichotomique`.

▸ **Exercice 6. (Bilan)** Comparer les différentes implémentations. Quels sont les avantages et inconvénients de chacune ?