TP Semaines 3 et 4
Notion d'agrégation

1 Les principaux mécanismes à travers un exemple

Le but de la séance est de savoir manipuler des classes dans lesquelles les données membres peuvent comprendre non seulement des types de base, mais aussi d'autres classes.

Pour cela, nous devons revenir sur le mécanisme d'instanciation d'une classe afin de le comprendre parfaitement et savoir, *dans tous les cas*, répondre aux questions suivantes :
- Quand et comment est appelé le constructeur ? Quel constructeur est appelé ?
- Quand et comment est appelé le destructeur ?

Vous savez actuellement répondre à ces questions dans le cas d'une classe *simple*, mais pas dans le cas d'une classe contenant des objets appartenant à d'autres classes. C'est donc le principal objectif de la séance !

Remarque importante : vous aurez très peu de code `C++` à écrire lors de la première partie de cette séance, ce n'est pas le but ! Nous vous fournissons volontairement la majorité du code pour vous permettre de faire de nombreux tests interactifs, et ainsi répondre par vous-même aux questions posées dans le sujet.
Compiler et exécuter le programme sans chercher à comprendre TOUS les affichages **ne vous apportera rien**.

L'exemple de base : les classes Point et Cercle

Recopiez chez vous le répertoire (et la totalité de son contenu) : `/net/exemples/AP2/TP03`.
Il contient un `Makefile`, les fichiers `mainpoint.cc` et `main.cc` utilisant deux classes dont les fichiers sont aussi dans le répertoire :
- la classe `Point`, que vous ne modifierez pas ; ces fichiers sont les mêmes qu'au TP précédent, pour une version "papier", reportez vous au document distribué en TD,
- une ébauche de la classe `Cercle` que vous allez tester et compléter. Les principaux fichiers sont en annexes de ce document.

Remarque : toutes les méthodes constructeur et destructeur de ces classes affichent un message à l'écran, ceci permettra de vérifier quand une méthode est appelée.

Exercice 13 : Classe `Point`.
Lisez attentivement le code de la classe `Point`, ainsi que le fichier `mainpoint.cc`.
Exécutez ce programme `mainpoint`, et assurez-vous que vous savez interpréter TOUS les affichages produits.

Exercice 14 : Classe `Cercle`.

1. Lisez attentivement le code de la classe `Cercle`. Vous remarquerez qu'elle possède une donnée membre de type `Point`, et vous porterez une attention particulière à la façon dont cette donnée membre est traitée (cf. commentaires dans le code).

2. Lisez attentivement le fichier `main.cc`. Exécutez le programme, et grâce aux affichages, reconstituez l'enchaînement des appels des méthodes et la *vie* de chaque objet, pas à pas sur une feuille de brouillon. En particulier, comment et quand est construit le point `my_centre` de chacun des cercles instanciés dans la fonction `main` ? Même question pour sa destruction.
3. La classe `Cercle` ne définit pas explicitement de constructeur par copie, donc un constructeur par copie est fourni par défaut.
   (a) Que remarquez-vous sur ce constructeur ? En particulier, comment est construit le point `my_centre` du cercle `c4` à la fin de la fonction `main` ?
   (b) Définissez explicitement un constructeur par copie dans la classe `Cercle`. Que remarquez-vous par rapport à l'exécution précédente ?

Constructeurs et la notation " :"

Nous venons de voir que, pour chaque constructeur défini explicitement dans la classe `Cercle`, son exécution fait appel automatiquement au constructeur par défaut de la classe `Point` pour instancier la donnée membre `my_centre`. Ainsi, si le point `my_centre` doit contenir d'autres valeurs que celles par défaut, il faut ensuite (dans le constructeur `Cercle` courant) modifier les valeurs de ce point. (cf. "constructeur rayon + x + y" et "constructeur rayon + point"). C'est un peu dommage...

Il est possible de spécifier à un constructeur de la classe `Cercle` quel constructeur de la classe `Point` utiliser pour instancier le point `my_centre` en donnant les paramètres d'appel de ce constructeur. Voici par exemple le "constructeur rayon + x + y" :

```
Cercle::Cercle( float r, float x, float y )
: my_centre( x, y )
// ici, appel Point::constructeur
{
  cout << "Cercle::Constructeur rayon + x + y " << endl;
  my_rayon = r;
  // plus besoin de modifier my_centre
}
```

De manière générale, à la suite du prototype du constructeur, et AVANT son corps délimité par `{ }`, on rajoute `: my_donnee1( params ), my_donnee2( params )...` (attention, les " :" font partie de la syntaxe !). Le type et l'ordre des `params` déterminent bien sûr le constructeur appelé ; ce sont soit des constantes, soit des paramètres du constructeur courant (comme pour l'exemple de `Cercle` donné ci-dessus).

Exercice 15 : Modifiez les constructeurs de la classe `Cercle` pour lesquels vous jugerez utile un tel fonctionnement. Vérifiez alors le bon enchaînement des appels à l'exécution du programme.
Le constructeur par copie n'échappe bien sûr pas à cette règle : faites en sorte qu'il fonctionne comme celui par défaut (cf. exercice précédent), c'est-à-dire qu'il fasse appel au constructeur par copie de la classe `Point`.

2 Pour aller plus loin

Nous vous proposons d'enrichir les classes Point et Cercle de la manière suivante : nous allons ajouter dans chaque classe deux données membres

- `static int nb_instance`, qui va compter le nombre d'instances de la classe.
- `int id`, qui va identifier l'instance et va être automatiquement initialiser lors de la construction de l'instance.

```
//Point.h
class Point {
  private :
    float my_abs, my_ord;
    static const float EPSILON;
    static int nb_instance;
    int id;
  public :
    ...
};

//Point.cc
...
const float Point::EPSILON=0.00000001; // calcul
int Point::nb_instance = 0;

Point::Point( float x, float y ) {
  nb_instance ++;
  id = nb_instance;
  // ou alors.. mais là on complique tout en rendant plus lisible!
  //    Point::nb_instance++;            // variable de classe
  //    (*this).id = Point::nb_instance; // affectation de cette variable
                                  // à une variable d'instance

  cout << "Point::Constructeur : " << x << " , " << y
       << " pour le point "<< id <<endl;
  my_abs = x;
  my_ord = y;
}
```

À vous de deviner l'intérêt d'un tel mécanisme.

Un Point ayant une "vie propre", nous pourrions définir la classe Cercle de la manière suivante :

```
class Cercle {
  private :
    Point *my_centre; // un pointeur sur un Point
    float my_rayon;
```

...

Implémenter.

3 Une application complète : la boîte englobante

Dans ce problème, on s'intéresse à définir et représenter une boîte englobant un certain nombre de points du plan. On se sert de deux classes : une classe `Point` qui représente tout point du plan, une classe `Boite` qui représente un ensemble de points et un rectangle (le rectangle minimal du plan pour lequel tous les points associés à cette boîte sont à l'intérieur). On considère que les points sont placés dans un repère orthonormé classique, i.e. que l'axe des x est orienté de la gauche vers la droite et que l'axe des y est orienté du bas vers le haut. La Figure 2 ci-dessous illustre le concept de boîte englobante d'un ensemble de points.

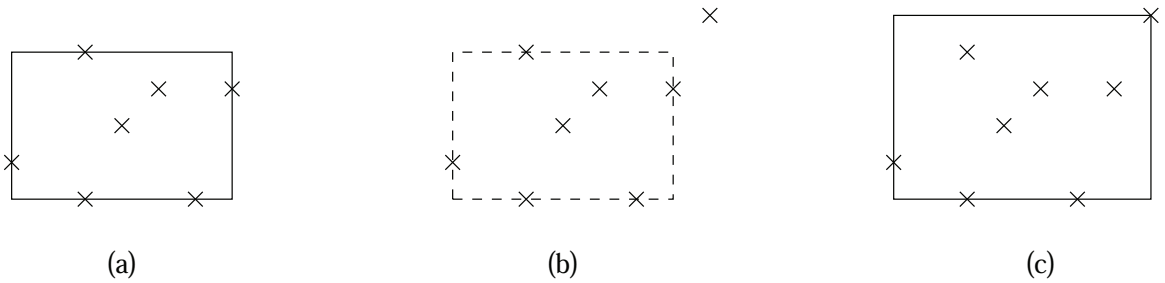

FIG. 2 – Boîte englobante : (a) boîte englobante d'un ensemble de points, (b) ajout d'un point, (c) nouvelle boîte englobante de ces points.

Dans le répertoire `/net/exemples/AP2/TP03/Boite_englobante`, vous avez à votre disposition un `Makefile`, la classe `Point` vue précédemment, et l'interface de la classe `Boite`. Vous allez écrire dans la suite les fichiers `Boite.cc` et `main.cc`.

Vous écrirez (**et testerez pas à pas ! !**) les fonctions de l'interface de la classe `Boite` en suivant les étapes ci-dessous.

Exercice 16 : Les constructeurs et le destructeur.
A noter que le constructeur par défaut (construction sans paramètre) de la classe `Boite` est interdit d'usage (car placé dans la section `private`) : une boîte englobante n'a de sens qu'à partir de un point (au moins). L'attribut `my_bas_gauche` permet de mémoriser le coin bas-gauche du rectangle englobant, l'attribut `my_haut_droite` permet de mémoriser le coin haut-droit de ce rectangle. L'attribut `my_nb_alloues` désigne le nombre de cellules allouées dynamiquement dans le tableau de points `my_points`. L'attribut `my_nb_points` désigne le nombre de points effectivement stocké dans ce tableau.

Ecrivez le constructeur qui construit une boite englobant un point : on suppose qu'on alloue un tableau de 5 points et que le point passé en paramètre est stocké en premier.

Puis écrivez le destructeur.

Exercice 17 : Deux méthodes simples.

1. Ecrivez la méthode `affiche` de la classe `Boite` qui affiche à l'écran la valeur de l'ensemble des attributs d'une boite.
2. La méthode `interieur` de la classe `Boite` doit renvoyer `true` si le point `p` passé en paramètre est à l'intérieur de la boîte, `false` sinon. Ecrivez cette méthode.

Exercice 18 : Pour ajouter un point.

1. Ecrivez la méthode `agrandir` de la classe `Boite`. Cette méthode est appelée lorsque l'on ajoute un point à la boîte et qu'il n'y a plus de place pour le mémoriser dans le tableau `my_points`. On considère que l'on double la taille du tableau à chaque fois.
2. Ecrivez maintenant la méthode `ajouterPoint`. L'objet `Boite` doit à la fois mémoriser le nouveau point **et** mettre à jour si nécessaire la boîte englobante.

Exercice 19 : Ecrivez la méthode `supprimerPoint`. Vous devez vérifier que le point appartient bien à l'ensemble des points mémorisés dans la boîte englobante. Puis, vous devez enlever ce point de ce tableau et recalculer **si nécessaire** la boîte englobante.

Exercice 20 : Rajoutez deux méthodes : le constructeur par copie et l'opérateur d'affectation de la classe `Boite`.