Semaine 5
Héritage (simple), Polymorphisme - Compléments de cours

Contenu du TD : notions d'héritage (simple), de polymorphisme, de transtypage, de classe abstraite...

Sources liées à ce TD :

- `AquaV1_a_distribuer.ps` : pour les étudiants

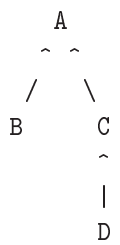
- `AquaV2_a_distribuer.ps` : pour les étudiants

Note aux enseignants : Un exemple nous servira de base pour cette séance : représentation d'un aquarium. Il y aura au final 3 classes principales : Element, Animal et Autre, Animal et Autre héritant de Element.
Deux versions successives de cet exemple ont été produites pour nous permettre d'introduire les différentes notions pédagogiquement (du moins je l'espère !). Deux fichiers postscript séparés ont été créés : `AquaV1_a_distribuer.ps` et `AquaV2_a_distribuer.ps`. Ils contiennent les 3 classes (seulement 2 pour la V1), un main.cc et l'exécution. Ils sont destinés à être distribués aux étudiants durant le TD.
Peu de "rappels" sont intégrés dans cette fiche de TD, pour plus de détails, se reporter aux transparents du cours.

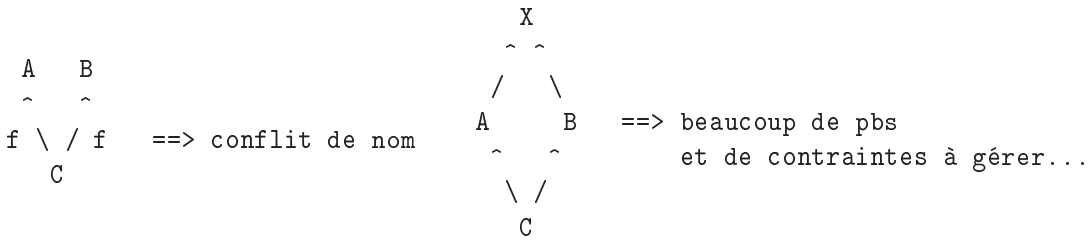
1 Héritage (simple) (AquaV1)

Petits rappels : Plutôt que de réimplémenter des fonctionnalités existantes déjà dans une classe A, une classe B peut intégrer des données et fonctions membres de la classe A. On dit que B *hérite* de A.

Bien sûr, plusieurs classes peuvent intervenir et former ainsi ce qu'on appelle une *hiérarchie d'héritage*. Lorsque chaque classe hérite d'une seule classe, on parle d'*héritage simple*, le graphe d'héritage est un arbre :



Lorsque chaque classe peut éventuellement hériter de plusieurs classes, on parle d'*héritage multiple*, le graphe d'héritage peut contenir des boucles et plusieurs problèmes peuvent survenir et compliquer la gestion :



D'ailleurs, certains langages n'autorisent que l'héritage simple (C#, Java). Nous ne traiterons ici que l'héritage simple !

Exercice 18 : Nous allons représenter les éléments d'un aquarium par une mini hiérarchie d'héritage. Dans un premier temps, nous aurons la classe `Element` qui sera la classe de base pouvant représenter tout élément (poissons, algues, roches, crabes, etc), et la classe `Animal` qui spécifiera la classe `Element` pour représenter les animaux de l'aquarium.
Écrivez une classe `Element` et une classe `Animal`, respectant les interfaces ci-dessous, regroupant ce qui a été vu en cours.

```
Element
-------
   my_x, my_y, my_z : entiers (coordonnées)
   ----------------
   Element(x,y,z)
   ~Element()
   mobile() : booléen
   deplacer(x,y,z) : void

Animal (hérite de Element)                          Element
------                                                 ^
   my_n : string (nom)                                 |
   -----------------                                   |
   Animal(n,x,y,z)                                   Animal
   ~Animal()
   mobile() : booléen
   deplacer(x,y,z) : void
```

Note pour les enseignants : guider les étudiants pour qu'ils écrivent les classes données en annexe (cf page web : `AquaV1_a_distribuer.ps`), sans les `virtual` qui seront commentés plus tard.

Écrivez ensuite un exemple de fonction `main` succinct pour observer comment fonctionnent les déclarations d'objets grâce aux constructeurs, ainsi que les appels de méthodes.

À la fin de l'exercice, distribuer la feuille `AquaV1_a_distribuer.ps` pour que les étudiants aient tous une version propre pour la suite.

Pendant le déroulement de cet exercice, on pourra faire des rappels sur :

- **Contrôle d'accès :** Rappelons et complétons la signification de `public`, `private` et `protected` pour la déclaration des membres d'une classe : un membre déclaré
  - `public` peut être utilisé n'importe où,
  - `private` ne peut être utilisé que par les fonctions membres de la classe au sein de laquelle il est déclaré,
  - `protected` ne peut être utilisé que par les fonctions membres de la classe au sein de laquelle il est déclaré, ainsi que par les fonctions membres des classes dérivées de cette classe.

  En ce qui concerne l'héritage, les 3 mêmes mots clé `public`, `private` et `protected` sont possibles. On se contentera ici de détailler l'héritage `public` :conserve les contrôles d'accès, membres `public` (resp. `protected`) restent membre `public` (resp. `protected`) de la classe dérivée. Les membres `private` de la classe que l'on dérive restent inaccessibles, même pour la classe dérivée.
- **Constructeurs et destructeurs :** ordres d'appels, appel explicite d'un constructeur de la classe mère avec la notation `:` (voir cela sur l'exécution proposée).
  Rappel : si B hérite de A :
  Ordre Constructeur : l'instruction `B obj;` provoque les choses suivantes :
  1. Allocation mémoire, autant d'octets que nécessaire compte-tenu des champs de B et de sa classe de base A.
  2. Appel du constructeur par défaut de A : A().
  3. Appel du constructeur par défaut de B : B().

  Ordre destructeur : l'appel des destructeurs se fait dans l'ordre inverse de l'appel des constructeurs. Lorsque `un_objet` est détruit, il se passe :
  1. Appel du destructeur de B.
  2. Appel du destructeur de A.
  3. La mémoire est rendue au système.
- **Redéfinition de méthodes :** la méthode `mobile` de Animal masque la méthode `mobile` de Element : cela s'appelle la *redéfinition* (*overriding*) de méthode. La redéfinition est particulièrement utile pour le polymorphisme. Sans le mot-clé `virtual` on obtient le fonctionnement suivant :

```
Animal poisson(....);
if (poisson.mobile()) ...;    // appel fonction membre Animal
Element epoisson(....);
if (epoisson.mobile()) ...;   // appel fonction membre de Element
```

2 Fonctions virtuelles et polymorphisme (AquaV1)

La redéfinition de la fonction `mobile()` telle que nous venons de la faire est permise en C++, MAIS ne fonctionne correctement que lorsque le type de l'objet sur lequel est appliquée cette fonction peut être clairement identifié par le compilateur lors de la création de l'exécutable. Le type étant clairement identifié, la fonction à appeler l'est aussi. C'est le cas pour l'exemple que nous venons de voir pour `poisson` et `epoisson`.

Or, il y a des cas où ce n'est pas possible de déterminer le type de l'objet à la compilation. Prenons l'exemple suivant.

Dans une hiérarchie d'héritage, un pointeur sur un objet du type de base peut désigner n'importe quel objet de la hiérarchie.

```
Animal poisson(....);
Element epoisson(....);
Element *pelement;

pelement = &epoisson;
if (pelement->mobile()) ...;  // utilise la fonction membre de Element

pelement = &poisson;
if (pelement->mobile()) ...;  // utilise la fonction membre de Element
                              // et non de Animal
```

A cause du *typage statique* du compilateur, c'est toujours la fonction `mobile` de Element que `pelement` atteint. Ce n'est évidemment pas ce qu'on veut, on voudrait que lorsque `pelement` pointe sur un objet Animal, `pelement->mobile()` appelle la fonction de Animal.

Pour cela, la fonction `mobile` dans Element doit être déclarée *virtuelle* (mot-clé `virtual` précède sa déclaration dans la classe). Ceci indique au compilateur qu'il doit laisser tomber le *typage statique* au profit du *typage dynamique* lors de l'appel de cette fonction. Le mot-clé `virtual` n'est pas nécessaire dans la classe Animal car le compilateur le rajoute automatiquement pour toutes les méthodes virtuelles redéfinies dans les classes dérivées.

Ainsi, on peut traiter des objets de classe dans une hiérarchie d'héritage comme des abstractions emboitables et compatibles : c'est ce qu'on appelle le *polymorphisme*.

Remarques : un constructeur ne peut pas être virtuel. Par contre, dès qu'une fonction dans une classe est virtuelle, le destructeur aussi doit être virtuel, sinon on s'expose à des pbs...

```
Element *pelement;
pelement = new Animal(....);
...
delete pelement;  // pb, appel destructeur de Element si destructeur non
                  // virtuel.
```

Reprendre AquaV1 et le commenter

- Le main reprend tout ce qu'on vient de voir.
- comparer les deux exécutions avec ou sans `virtual`.

Exemple classique d'utilisation du polymorphisme :
gérer un tableau de pointeurs d'objets de base dans lequel on peut mettre n'importe lesquels des objets de la hiérarchie d'héritage. Quand on parcourt tous les objets du tableau pour leur appliquer la "même" fonction (par exemple `pelement->mobile()`), la bonne fonction sera choisie à l'exécution.

Quand déclarer une fonction virtuelle ?

1. si on prévoit que la classe sera dérivée, plus précisément, si on veut avoir une interface abstraite commune à plusieurs classes dérivées,

2. si l'implémentation de la fonction est dépendante du type, sinon, héritage direct et pas besoin de la redéfinir...

3 Continuons d'enrichir l'exemple : Transtypage

Un pointeur de type `T` peut pointer vers tout objet de type `T'` où `T'` est une spécialisation de `T`. Il est parfois utile de retrouver le type réel d'un objet pointé, par exemple pour appeler des méthodes spécialisées qui ne sont pas définies dans le type de base.

Sur l'exemple de l'aquarium, on instancie parfois des Animaux. On voudrait alors, à partir d'un pointeur sur `Element` pointant vers un de ces animaux, obtenir un pointeur sur `Animal` sur cette même instance. On parle de *transtypage*. Or, on ne peut écrire directement :

```
Animal ours( "ours", 5, 10, 2 );
Element* ptr = &ours;
Animal* animal = ptr; // interdit
```

En effet, le compilateur ne peut pas autoriser une telle instruction car rien ne dit que l'élément pointé par `ptr` a été effectivement instancié comme un objet `Animal` (ou dérivé). Lorsque l'on souhaite transtyper un pointeur, il faut donc l'indiquer explicitement. Cela se fait avec l'opérateur `dynamic_cast`. Ainsi :

```
Animal ours( "ours", 5, 10, 2 );
Element* ptr = &ours;
Animal* animal = dynamic_cast<Animal*>( ptr ); // ok
if ( animal == 0 )
  cerr << "ptr ne pointait pas sur une instance de Animal." << endl;
else ...
```

L'opérateur `dynamic_cast` analyse le type réel de l'objet pointé et, si le transtypage est valide, retourne un pointeur du bon type, sinon il retourne le pointeur `0`. En quelque sorte, il permet de redescendre da manière sécurisée dans une hiérarchie de types.

Dans notre exemple, la méthode `deplacer` a peu de sens dans la classe `Element`. On peut donc l'enlever maintenant et n'appeler cette méthode que sur les objets transtypés.

```
Element* tbl[ MAX ];
...
for ( int i = 0; i < MAX; i++ ) {
  if ( tbl[ i ]->mobile() )
    dynamic_cast<Animal*>( tbl[ i ] )->deplacer( 1, 0, 0 );
}
```

4 Classes abstraites : introduction (AquaV2)

Certaines classes représentent des concepts pour lesquels des objets ne peuvent exister. Par exemple, la classe Element que nous venons de voir n'a de sens que comme classe de base pour des classes dérivées Animal et Autre. Vouloir instancier cette classe n'a en fait pas de sens. C'est la notion de "classe abstraite".

En C++, une classe est abstraite dès qu'une ou plusieurs fonctions membres sont *virtuelles pures*, cad que leur déclaration est précédée du mot clé `virtual` et suivie de l'expression `=0`. L'expression `=0` indique au compilateur qu'il devra chercher la définition de la fonction dans la ou les classes dérivées.

Distribuer Aqua2 et le commenter en parlant des méthodes virtuelles pures

```
class Element {      // classe abstraite
  protected:
     int my_x, my_y, my_z;  // coordonnées
  public:
     ...
     virtual ~Element();
     virtual bool mobile()=0;                // fonction virtuelle pure
};
```

Instancier une classe abstraite n'a aucun sens, donc le code suivant provoquera un erreur de compilation :

```
  Element e;     // erreur car Element est une classe abstraite
```

La classe Element n'est qu'une interface abstraite pour des classes dérivées spécifiant des comportement particuliers (Animal, Autre), ceci dans le but d'utiliser le polymorphisme.