POO C++

Sommaire

Notion d'héritage

L'**héritage** est un **concept fondamental de la programmation orientée objet**. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.

L'héritage permet **d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise**. Il permet donc **la spécialisation ou la dérivation de types**.

B **hérite** de A : "un B **est** un A avec des choses en plus". Toutes les instances de B sont aussi des instances de A.

On dit aussi que B dérive de A. A est une généralisation de B et B est une spécialisation de A.

Propriétés de l'héritage

L'héritage est **une relation entre classes** qui a les propriétés suivantes :
- si B hérite de A et si C hérite de B alors C hérite de A
- une classe ne peut hériter d'elle-même
- si A hérite de B, B n'hérite pas de A
- il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C
- le C++ permet à une classe C d'hériter des propriétés des classes A et B (héritage multiple)

Notion de visibilité pour l'héritage

*Rappels : Le C++ permet de préciser le **type d'accès des membres** (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets.*

Il faut maintenant tenir compte de la situation d'héritage :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **non par ceux d'une autre classe même dérivée**.
- **protégé** (*protected*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **par ceux d'une classe dérivée**.

Notion de redéfinition

En utilisant l'héritage, il est possible **d'ajouter des caractéristiques**, **d'utiliser les caractéristiques héritées** et de **redéfinir les méthodes héritées**.

Généralement, cette **redéfinition (*overriding*)** se fait par **surcharge** et permet de modifier le comportement hérité.

Exemple détaillé : des dames coquettes

On désire réaliser des programmes orientés objet en C++ qui **raconteront des histoires dans lesquelles un cowboy rencontre une dame coquette** :

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky
Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une
jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de coca-cola !
GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```

Les intervenants de nos histoires sont tous des humains. Notre **humain** est caractérisé par son **nom** et sa **boisson favorite**. La boisson favorite d'un humain est, **par défaut**, de l'eau.

UML	C++

Humain
string nom
string boissonFavorite
	`class Humain`
	`{`
	`  private:`
	`  string nom;`
	`  string boissonFavorite;`
	`};`

Un humain pourra **parler**. On aura donc une **méthode** `parle(texte)` qui affiche :

`(nom de l'humain) -- texte`

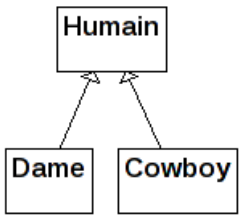
UML	C++

Humain
string nom
string boissonFavorite
Humain(const string nom = "", const string boissonFavorite = "eau")
void parle(const string texte)
	`class Humain`
	`{`
	`  private:`
	`  string nom;`
	`  string boissonFavorite;`
	`  public:`
	`  Humain(const string nom="", const string boissonFavorite="eau");`
	`  void parle(const string texte);`
	`};`
	
	`void Humain::parle(const string texte)`
	`{`
	`  cout << "(" << nom << ") -- " << texte << endl;`
	`}`

Un **humain** pourra également **se présenter** (il dit bonjour, son nom, et indique sa boisson favorite), et **boire** (il dira « Ah ! un bon verre de (sa boisson favorite) ! GLOUPS ! »). On veut aussi pouvoir connaître le **nom** d'un **humain** mais il n'est pas possible de le modifier.

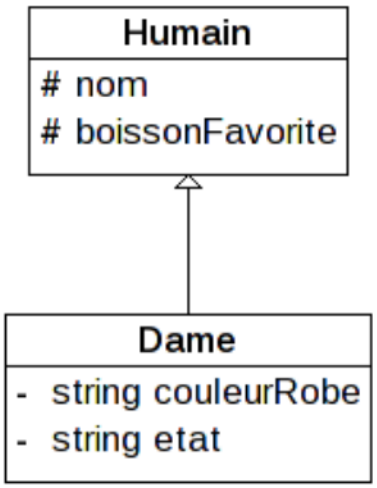
Les **dames** et les **cowboys** sont tous des **humains**. Ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces **deux classes d'humains**.

On va donc réaliser la modélisation suivante en utilisant l'héritage :



Une **dame** est caractérisée par la **couleur de sa robe** (une chaîne de caractères), et par son **état** (libre ou captive).

UML



C++

```
class Humain
{
  public:
      Humain(const string nom="", const
string boissonFavorite="eau");

  protected:
      string nom;
      string boissonFavorite;
};

// La classe Dame hérite de la classe Humain
class Dame : public Humain
{
   public:
      Dame(const string nom="", const string
boissonFavorite="lait", const string
couleurRobe="blanche");

   private:
      string couleurRobe;
      string etat;
};

// Le constructeur Dame appelle le constructeur
Humain puis intialise ses propres attributs
Dame::Dame(const string nom, const string
boissonFavorite, const string couleurRobe) :
Humain(nom, boissonFavorite),
couleurRobe(couleurRobe), etat("libre")
{ }
```

Il est nécessaire d'indiquer une visibilité `protected` aux membres `nom` et `boissonFavorite` pour permettre aux objets de la classe `Dame` d'accéder à ces membres.

Elle peut également **changer de robe** (tout en s'écriant « Regardez ma nouvelle robe (couleur de la robe) ! »). On désire aussi changer le mode de présentation des dames car une dame ne pourra s'empêcher de parler de la couleur de sa robe. Et quand on demande son **nom** a une **dame**, elle devra répondre « Miss (son nom) ».

Il faut donc **redéfinir les méthodes héritées** `getNom()` et `sePresente()` pour obtenir le comportement suivant :

```
Dame jenny("Jenny");

jenny.sePresente();
```

devra donner :

```
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
```


On déclare la classe `Dame` :

```
class Dame : public Humain
{
   public:
      // Constructeurs et Destructeur
      Dame(const string nom="", const string boissonFavorite="lait", const string
          couleurRobe="blanche");

      // Accesseurs
      string getNom() const; // surcharge
      string getEtat() const;

      // Services
      void sePresente() const; // surcharge
      void changeDeRobe(const string couleurRobe);

   private:
      string couleurRobe;
      string etat;
};
```

dame.h

On définit les méthodes de la classe `Dame` :

```
#include "dame.h"

// Constructeur
Dame::Dame(const string nom/*=""*/, const string boissonFavorite/*="lait"*/, const string
    couleurRobe/*="blanche"*/) : Humain(nom, boissonFavorite), couleurRobe(couleurRobe),
    etat("libre")
{
}

// Accesseurs
string Dame::getNom() const
{
   return "Miss " + nom;
}

string Dame::getEtat() const
{
   return etat;
}

// Services
void Dame::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " et j'ai une jolie
       robe " << couleurRobe << endl;
}

void Dame::changeDeRobe(const string couleurRobe)
{
   this->couleurRobe = couleurRobe;
```



```
   cout << "(" << nom << ") -- " << "Regardez ma nouvelle robe " << couleurRobe << " !" <<
      endl;
}
```

dame.cpp

Un **cowboy** est un **humain** qui est caractérisé par sa popularité (0 pour commencer) et un adjectif le caractérisant ("vaillant" par défaut). On désire aussi changer le mode de présentation des cowboys. Un cowboy dira ce que les autres disent de lui (son adjectif).

De même, on peut donner une boisson par défaut à chaque sous-classe d'humain : du lait pour les dames et du whisky pour les cowboys ...

Notion de redéfinition

Il ne faut pas mélanger la redéfinition et la surdéfinition :
- Une **surdéfinition (ou surcharge)** permet **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente**.
- Une **redéfinition (*overriding*)** permet **de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer**. Elle doit avoir une signature rigoureusement identique à la méthode parente.

Un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire ou de la compléter.

Notion de polymorphisme

Le **polymorphisme** représente **la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation**.

On voit donc apparaître ici le concept de **polymorphisme** : **choisir en fonction des besoins quelle méthode appeler et ce au cours même de l'exécution**.

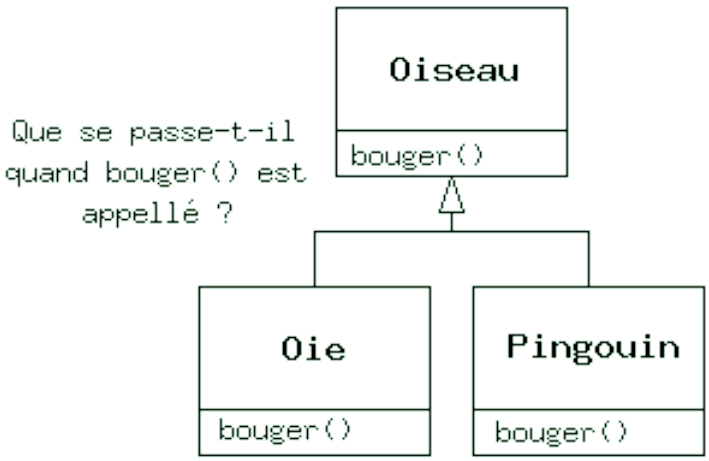
Le **polymorphisme** est implémenté **en C++** avec **les fonctions `virtual` et l'héritage**.

Notion de fonctions virtuelles

Pour créer une fonction membre `virtual`, il suffit de faire précéder la déclaration de la fonction du mot-clef `virtual`. Seule, la déclaration nécessite ce mot-clef, pas la définition. Si une fonction est déclarée `virtual` dans la classe de base, elle est `virtual` dans toutes les classes dérivées.

La redéfinition d'une fonction virtuelle dans une classe dérivée est généralement appelée **redéfinition (*overriding*)**.

Exemple : Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'`Oiseau`, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ? Car la méthode `bouger()` de la classe `Oiseau` a été déclarée `virtual`. Puis les classes `Oie` et `Pingouin` l'ont redéfinie pour obtenir le bon comportement.



On commence à posséder de nombreux types d'humain. On va introduire une fonction qui leur permettra de se présenter :

```
void presentezVous(const Humain &humain)
{
   humain.sePresente();
}
```

presentezVous()

Ce qui permettra à chaque humain de se présenter correctement :

```
Cowboy lucky("Lucky Luke", "coca-cola");
Dame jenny("Jenny");

// 1. les présentations des personnages de l'histoire
presentezVous(lucky);
presentezVous(jenny);
```

Les présentations

Il n'y a pas d'erreurs à la compilation puisque les **cowboys**, les **dames**, les **barmans** et même les **brigands** sont tous des **humains**.

Par contre, on n'obtient pas ce que l'on désire à l'exécution :

```
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Jenny et j'aime le lait
```

En effet, c'est la méthode `sePresente()` de la classe `Humain` qui est appelée et non celle des classes `Cowboy`, `Dame`, `Barman` et `Brigand`.

Pour corriger cela, il suffit de déclarer la méthode `sePresente()` comme **virtuelle** (**`virtual`**) dans la classe `Humain` :

```
class Humain
{
   ...
   virtual void sePresente() const;
```



```
   ...
};
```

humain.h

Puis, chaque classe héritée de `Humain` pourra redéfinir la méthode `sePresente()` :

```
class Dame : public Humain
{
   ...
   void sePresente() const;
   ...
};

void Dame::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " et j'ai une jolie
      robe " << couleurRobe << endl;
}
```

dame.h

On obtient maintenant un **comportement polymorphe** : la "bonne méthode" `sePresente()` est appelée en fonction du type de l'objet à l'exécution

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
```

