TP POO C++ n°6

Sommaire

Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++.
On désire réaliser un programme C++ permettant d'écrire facilement des histoires de Western.
Dans nos histoires, nous aurons des brigands, des cowboys, des shérifs, des barmen et des dames en détresses ... (à partir d'une idée de Laurent Provot)

Western C++

Objectifs

On désire réaliser un programme orienté objet en C++ qui **racontera une histoire dans laquelle on découvre un barman**.

```
(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
(Joe) -- Bonjour, je suis Joe le méchant et j'aime le tord-
boyaux.
(Robert) -- Bonjour, je suis Robert le barman du Saloon Robert
et j'aime le bière mon gars.
(Robert) -- Tiens Lucky Luke, un verre de coca-cola mon gars.
(Robert) -- Tiens Jenny, un verre de lait mon gars.
(Robert) -- Tiens Joe, un verre de tord-boyaux mon gars.
```

On ajoutera ensuite à notre histoire des **shérifs**.

La programmation orientée objet en action

Rappels

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

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**.

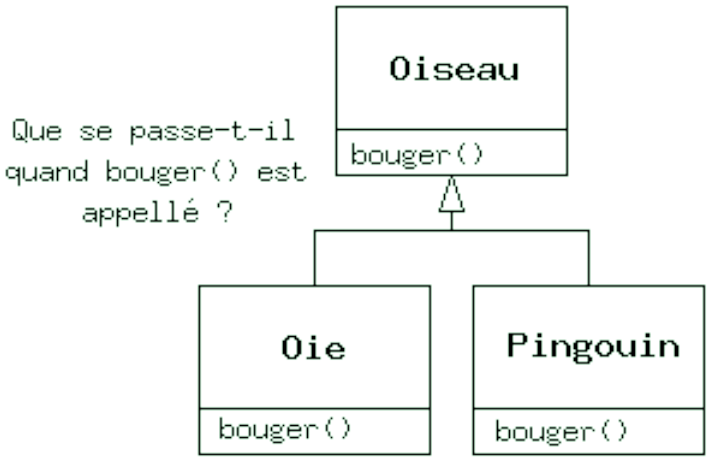
Le **polymorphisme** est implémenté **en C++** avec **les fonctions `virtual` et l'héritage**.

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.

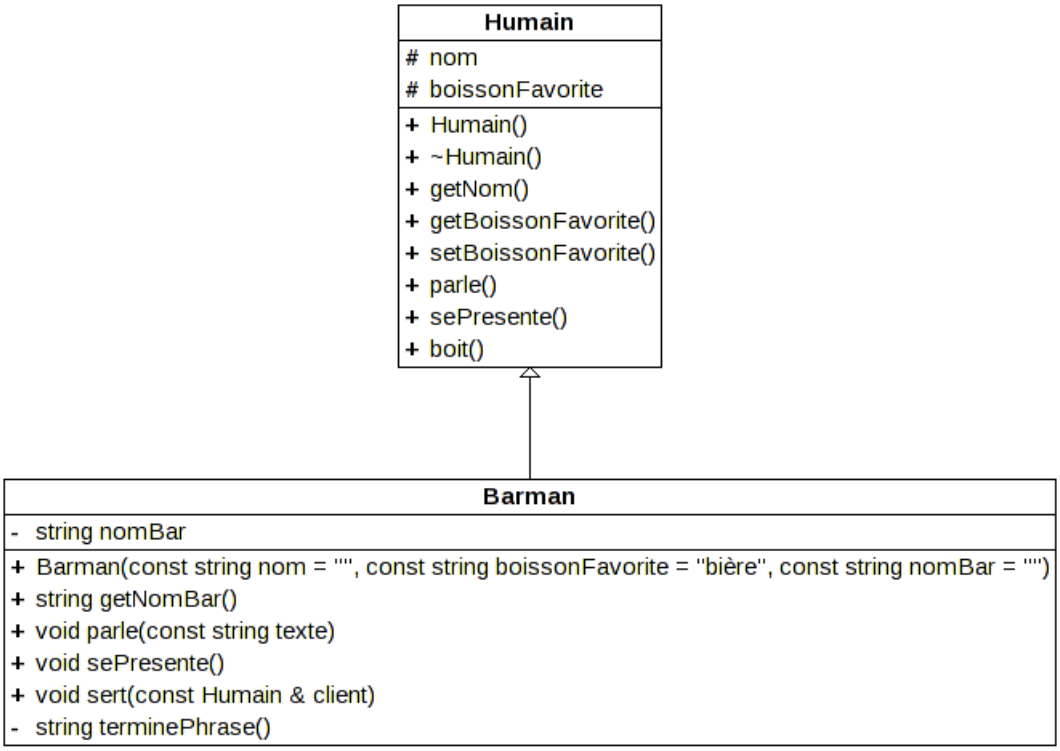


Un verre, patron !

Exemple détaillé : les barmans

Un **barman** est un **humain** dont la **boisson favorite** est la bière par défaut. Il peut **servir** n'importe quel **humain**, en lui donnant un verre de sa **boisson favorite** (les barmen ont un sixième sens pour cela).

Il est caractérisé par le **nom de son bar**. Par défaut, il s'agira du bar « Saloon (nom du barman) ». Quand un barman **se présente**, il n'oublie pas de mentionner le nom de son bar. Quand un barman **parle**, il **termine toutes ses phrases** par « mon gars. ».



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");
Brigand joe("Joe");
Barman robert("Robert");

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



```
presentezVous(joe);
presentezVous(robert);
```

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
(Joe) -- Bonjour, je suis Joe et j'aime le tord-boyaux
(Robert) -- Bonjour, je suis Robert et j'aime le bière
```

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 Barman : public Humain
{
   ...
   void sePresente() const;
   ...
};

void Barman::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " le barman du " <<
      getNomBar() << " et j'aime le " << getBoissonFavorite() << terminePhrase() << endl;
}

string Barman::terminePhrase() const
{
   return " mon gars.";
}
```

barman.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
(Joe) -- Bonjour, je suis Joe le méchant et j'aime le tord-boyaux.
(Robert) -- Bonjour, je suis Robert le barman du Saloon Robert et j'aime le bière
```


Question 1. Coder la classe `Barman` et écrire une histoire où le barman Robert sert succesivement tous les personnages existants (lucky, jenny et joe).

```
(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
(Joe) -- Bonjour, je suis Joe le méchant et j'aime le tord-boyaux.
(Robert) -- Bonjour, je suis Robert le barman du Saloon Robert et j'aime le bière mon gars.
(Robert) -- Tiens Lucky Luke, un verre de coca-cola mon gars.
(Robert) -- Tiens Jenny, un verre de lait mon gars.
(Robert) -- Tiens Joe, un verre de tord-boyaux mon gars.
```

I'm the law !

La classe Sherif

On ajoute maintenant à notre histoire des shérifs. Un shérif est un cowboy qui peut coffrer des brigands (en criant « Au nom de la loi, je vous arrête ! ». Il peut également rechercher un brigand. Un commentaire indique alors qu'il placarde une affiche dans toute la ville, et il dit, par exemple, « OYEZ OYEZ BRAVE GENS !! 200 $ à qui arrêtera Bob le brigand mort ou vif !! ». Naturellement le contenu du commentaire dépendra du brigand en question.

Tout le monde s'accorde pour dire que les shérifs sont honnêtes. Un shérif est caractérisé par le nombre de brigands qu'il a coffrés, ce qu'il ne manquera pas de préciser lorsqu'il se présente.

Il refuse de se faire appeler autrement que par Shérif (son nom).

Question 2. Coder la classe `Sherif` et écrire une histoire où le shérif Clint recherche puis coffre Joe le brigand.

```
(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
(Joe) -- Bonjour, je suis Joe le méchant et j'aime le tord-boyaux.
(Robert) -- Bonjour, je suis Robert le barman du Saloon Robert et j'aime le bière mon gars.
(Clint) -- Bonjour, je suis Shérif Clint et j'aime le eau.
** OYEZ OYEZ BRAVE GENS ! ! 100$ à qui arrêtera Joe mort ou vif ! !
(Joe) -- Damned, je suis fait ! Clint, tu m'as eu !
```

