TP Western C++ n°2

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)

Histoire n°2 : des cowboys et des dames ...

Objectifs

On désire réaliser un programme orienté objet en C++ qui **racontera une histoire dans laquelle 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 !
```

La programmation orientée objet en action

Rappels

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

Une classe **déclare propriétés communes (attributs et méthodes)** à un ensemble d'objets. Elle apparat comme un **type** ou un *modèle* à partir duquel il sera possible de créer des objets.

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

*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

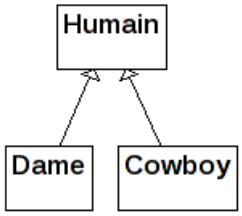
En utilisant l'héritage, il est possible **d'ajouter des caractéristiques**, **d'utiliser les caractéristiques héritées** et de **redéfinir les méthodes héritées**.

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

Exemple détaillé : des dames coquettes

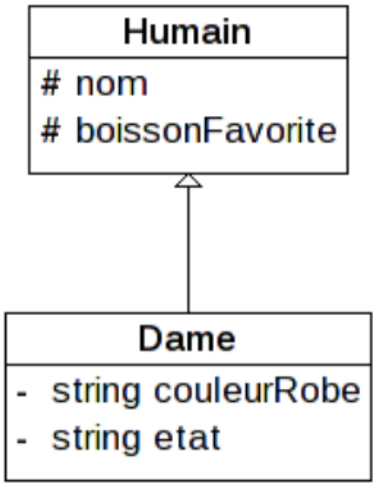
Les **dames** et les **cowboys** sont tous des **humains**. Ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces **deux classes d'humains**.

On va donc réaliser la modélisation suivante :



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 hérités** `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

Travail demandé

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 veut donner une boisson par défaut à chaque sous-classe d'humain : du lait pour les dames et du whisky pour les cowboys.

Question 1. Écrire les classes `Cowboy` et `Dame` afin d'assurer l'exécution du programme de test ci-dessous.

```
#include <iostream>

using namespace std;

#include "cowboy.h"
#include "dame.h"

/* TP Western C++ n°2 : des cowboys et des dames */

int main()
{
   Cowboy lucky("Lucky Luke");
   Dame jenny("Jenny");

   // 1. La rencontre ...
   lucky.sePresente();
   jenny.sePresente();

   // 2. Allons boire un coup ...
   jenny.changeDeRobe("verte");
   lucky.boit();
   jenny.boit();

   return 0;
}
```

histoire-2.cpp


```
$ ./histoire-2
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky Luke et j'aime le le whisky
(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 !
$
```

