Master 1 Informatique
Université de Lyon 2 - ICOM 2023/2024

Module Programmation de spécialité *C++*

Travaux Pratiques 4

Programmation Objet

Objectif: L'objectif de ce TP est de comprendre la création et l'utilisation des objets dans le cadre d'une programmation orientée objet.

Concept de cours :

Notion d'objet : Un objet est une structure qui possède ses propres variables appelées `attributs` et ses propres `fonctions` appelées méthodes.

Définition d'une classe : en C++, on peut créer ses propres objets en définissant une `classe`. Cela permet de mieux organiser son code. On peut aussi utiliser des classes déjà codées. Par exemple les vecteurs que nous avons utilisés pour réaliser la calculatrice du TP2 utilisait un (`vector<double>`) qui sont des objets C++ dont on se sert très souvent sans avoir à re coder la classe.

Exemple concret : ma première classe
Nous voulons coder la classe `Personnage` qui a deux attributs :

- un `nom` ;
- un `nombre de points de vie`.

Ensuite, nous voulons une methode `getStat()` qui affiche les caractéristiques du personnage.
A noter : une classe contient toujours une méthode qui permet de construire l'objet : le constructeur.
Pour réaliser cette classe, il est nécessaire de créer deux fichiers :

- Fichier `Personnage.h` : contient ce qui définit la classe : les attributs et les prototypes des méthodes.

Personnage.h :

```
#include <string>
class Personnage
{
   std::string nom; // Attribut
   int vie;         // Attribut
public:
   Personnage(std::string nom, int vie);
};
```

- Fichier `Personnage.cpp` : contient essentiellement le "code" des méthodes :

Personnage.cpp :

```
#include "Personnage.h"
#include <iostream>
// Constructeur
Personnage::Personnage(std::string nomPerso, int viePerso):nom(
    nomPerso),vie(viePerso){};
```

La création d'une classe s'effectue ensuite de la façon suivante :

```
int main(){
    Personnage joueur("Tom",20);


}
```

Exercice 4.1 *Première classe*
Dans cet exercice, il s'agit d'appliquer directement les définitions précédentes afin de comprendre la création et l'utilisation d'objets.

1. Initiez un nouveau projet en suivant les indications suivantes :
   — Utilisez *Visual Studio*.
   — Choisissez un projet de base de type **Application console C++**.
   — Une fois le projet créé, initialisez un dépôt git local et vous aurez à commiter chaque question des exercices.

2. Afin de tester les concepts de classe décrits précédemment, vous devez rajouter deux nouveaux fichiers au projet : un fichier que vous nommerez `Personnage.h` qui représentera la déclaration de la classe `Personnage` et le fichier `Personnage.cpp` qui représentera le code des méthodes. Vous pouvez utiliser l'assistant de classe proposé dans *Visual Studio* qui est visible à partir de l'explorateur de solutions.

3. En recopiant les codes précédents, testez la création de classe en créant plusieurs joueurs.

Maintenant que vous savez comment définir une classe, il s'agit d'être capable de rajouter une méthode. Dans notre exemple, nous allons rajouter le méthode `stat` qui renverra un objet de type `string` contenant les statistiques du joueur.

4. Rajoutez la déclaration de la nouvelle méthode `stat()` dans le fichier *header* `Personnage.h` puis rajoutez ensuite le code de la fonction dans le fichier `Personnage.cpp`.

5. Dans le fichier `main.cpp`, utilisez la méthode `stat()` sur les différents objets créés afin d'afficher les caractéristiques des joueurs.

6. Toujours dans la fonction `main()`, tentez de changer l'attribut associé au nombre de point de vie d'un des joueur. Pour cela, vous pouvez tenter de le modifier en utilisant directement l'attribut de l'objet à partir de l'opérateur point. Le changement a t'il fonctionné ?

En C++, il est possible d'utiliser des *opérateurs* qui se définissent comme des fonctions mais préfixé par le mot clé `operator` devant le symbole de l'opérateur.
7. Dans la classe personnage, rajoutez un opérateur binaire << qui prendra en paramètre un autre joueur et qui prendra les points de vie du joueur passé en paramètre et les donnera au joueur appellant l'opérateur <<.
8. Même question mais en rajoutant un opérateur de type ++ qui préfixer à l'objet fera gagné 10 points de vie.
9. Rajoutez un opérateur << permettant d'afficher directement dans le flux ostream les statistiques du joueur.

Concept du cours (suite) :

Encapsulation : chaque attribut et méthode d'une classe peut avoir son propre droit d'accès. Il en existe 3 différents :

- **Public :** l'attribut ou la méthode peut être utilisé depuis l'extérieur de l'objet.
- **Private :** l'attribut ou la méthode ne peut être utilisé que depuis de l'objet.
- **Protected** : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur sauf depuis ses classes héritées.

En pratique, on met toujours les attributs en private (ou protected) et les méthodes en public. C'est le principe d'encapsulation.

Exercice 4.2 *Encapsulation et destruction d'objets*

L'objet de cet exercice est d'illustrer cette notion d'encapsulation et doit être effectué dans le même projet que l'exercice précédent.

1. Afin qu'un utilisateur puisse quand même changer la vie d'un joueur, déclarez une nouvelle fonction `void changeVie(int vie)` dans le fichier `Personnage.h` et implémentez la code de la fonction dans le fichier `Personnage.cpp`.
2. Dans le `main`, utilisez la méthode précédente et mettez en évidence que la méthode a bien fonctionné sur vos deux objets.
3. L'encapsulation est aussi utile pour controler les valeurs définies à certains attributs. Faites en sorte qu'il ne soit pas possible de donner une valeur négative à l'attribue `vie`. En cas de tentative d'attribution, donner un message d'avertissement (utiliser la sortie `wcout` à la place de `cout`).
4. Afin de tester l'appel du *destructeur* d'un objet, rajoutez une déclaration du destructeur qui s'écrit en suivant la même logique que le constructeur mais en rajoutant le symbole ~ devant le nom de la classe.

Exercice 4.3 *Compte Bancaire*

En vous aidant des exercices précédents, créez un programme qui permette de faire des dépôts et retraits sur un compte bancaire ainsi que de connaître son solde. Le compte devra présenter les méthodes suivantes :

1. `deposerArgent(double a)` : déposera une certaine somme d'argent sur le compte.
2. `retraitArgent(double a)` : retirera une certaine somme d'argent sur le compte. Si le compte n'a pas assez d'argent, la méthode renverra false et true sinon.
3. `consulteSolde()` : pour consulter le solde du compte. Si le compte est négatif on souhaiterai avoir un affichage en rouge.

Notions de cours (suite et fin) :

Héritage : On appelle `classe héritée` une classe qui est un "sous ensemble" de la `classe parente`. **Exemple :** on souhaite créer la classe `Magicien`. Un Magicien est un Personnage qui a un nombre de point de vie et un nom. Mais il possède un attribut en plus : la `mana`. L'héritage permet de créer cette classe sans avoir à réécrire tout le code de la classe `Personnage`. L'ensemble des méthodes de la classe `Personnage` pourra s'appliquer sur les objets `Magicien` mais l'inverse n'est pas vrai ! Pour définir une classe qui hérite d'une classe personnage, la syntaxe du fichier *.h* est la suivante :

Magicien.h

```
#include "Personnage.h"

class Magicien : public Personnage{
   public:
      Magicien(std::string nom, int vie, int mana);
      int get_mana();
   protected:
      int mana;
};
```

Et le fichier du code associé est le suivant :

Magicien.cpp

```
#include "Personnage.h"
#include "Magicien.h"

Magicien::Magicien(std::string nomMage, int vieMage, int manaMage):
    Personnage(nomMage, vieMage),mana(manaMage){};

int Magicien::get_mana(){
    return mana;
}
```

Exercice 4.4 *Analyse de code et polymorphisme*

1. Recopiez les morceaux de code d'illustration de l'exemple encadré ci-dessus..

2. D'après ces exemples précédents, donner la syntaxe générale d'une déclaration de classe pour définir qu'une classe A hérite d'une autre classe B.

3. Essayez de donner une explication sur le fonctionnement du constructeur. Les explications sont à mettre en commentaire dans votre code.

4. L'héritage permet d'appliquer un mécanisme dit de surcharge de méthode qui permet de redéfinir une méthode déjà existante afin de la spécialiser. Cette surcharge peut être faite simplement en re définissant la méthode et en éventuellement appellant la méthode père. Testez ce mécanisme en re-définissant la méthode `stats()` qui fera appel à la fonction `stats()` du personnage puis affichera le valeur de Mana.

5. Testez le bon fonctionnement de la méthode dans la classe main.

6. De la même manière que précédement, surcharger la méthode `changeVie` pour le magicien et faire afficher un message dans la console différenciant la méthode de classe Personnage.
7. Dans votre fichier principal, rajoutez une fonction permettant de simuler un combat entre deux joueurs. Le changement de point sera pris au hazard entre -50 et 50 et utilisera la méthode `changeVie()`.
8. Testez la fonction en utilisant deux personnages et afficher statistiques des personnages.
9. Même question mais en faisant combattre un personnage et un magicien. L'appel de la méthode a t'elle réussie ? Quelle méthode `changeVie` a été appelée par votre fonction ?
10. Afin d'améliorer le comportement espéré de la question précédente, testez d'utiliser le mot clé *virtual* dans la classe Personnage.

Exercice 4.5 *Exercice 6 suite : Compte Bancaire*

1. Créez la classe LivretA, héritée de la classe CompteBancaire.

2. Vérifiez l'utilisation des méthodes de la classe parent par un objet de type LivretA. N'oubliez pas de définir un attribut propre à la classe livret A : le plafond.