TP 4 : création de classes

Exercice 4a : Créer le programme *main.cxx* qui va comporter la déclaration complète de la classe *Shape* ainsi que la fonction *main()* qui va permettre de l'utiliser. La classe *Shape* décrit une forme en 2D et comporte les champs et méthodes suivants :

- *Width* : la largeur de la forme, dans une unité quelconque,
- *Height* : la hauteur de la forme,
- *string getName()* : renvoie le nom de la classe,
- *double getBox()* : renvoie la surface du rectangle horizontal englobant la forme (surface d'une image minimale qui contiendrait la forme).

Déclarez cette classe à l'aide du mot clé *struct*, qu'il est inutile de répéter lors de la déclaration d'une instance de classe. Dans la fonction *main()* déclarez 2 instances de cette classe *sh1* et *sh2*, en initialisant les attributs *Width* et *Height* de la *sh2* à 2, puis affichez les dimensions des 2 formes et testez les autres méthodes. A l'intérieur des méthodes, utilisez systématiquement le mot clé *this* pour accéder aux membres de la classe.
Modifiez le code de la fonction *main()* afin de doubler la largeur de *sh2*.
Sortie attendue :

```
scinfr066:TP4 Wilfrid$ ./a.out
Shape: 2.19405e-314x0 = 0
Shape: 2x2 = 4
Shape: 4x4 = 8
```

Exercice 4b : Un principe fondamental de la POO est l'encapsulation, qui permet de protéger l'accès aux membres d'une classe par des entités extérieures, comme par exemple ce que l'on fait lorsque l'on modifie le membre *width* de la classe *Shape* depuis la fonction *main()*, qui n'est pas une méthode de *Shape*.
Changez le mot clé *struct* par le mot clé *class* dans la déclaration de *Shape*. Désormais les membres de *Shape* sont par défaut privés et chaque tentative de les utiliser depuis la fonction *main()* génère une erreur de compilation. Ajouter le qualifier *public* au début du corps de la déclaration de classe et vous constatez que l'on peut à nouveau compiler.
Respecter le principe d'encapsulation a pour objectif de protéger le contenu de la classe mais aussi de rendre son implémentation indépendante de son utilisation. La règle consistera donc à rendre privées toutes les variables internes et à rendre publiques toutes les méthodes qui implémentent les services rendus par la classe aux entités externes. Modifiez le programme en conséquence.
Remarque : on place généralement la section publique en premier et la section privée en dernier, pour des raisons de lisibilité (on s'intéresse en principe d'abord aux services rendus par une classe et non à son implémentation).
Vous constatez qu'il n'est plus possible d'initialiser *Width* et *Height* lors de la déclaration de *sh2* car *Shape* n'est plus un agrégat. Les conditions pour qu'une classe soit un agrégat sont :

- pas d'héritage de classe,
- absence de constructeur déclaré,
- pas de méthode virtuelle,
- tous les membres non statiques sont publics.

On voit que le fait de rendre privés *Width* et *Height* a invalidé la dernière condition. Supprimer l'initialisation lors de l'instanciation et la remplacer par une initialisation lors de la déclaration « à la C ». Supprimer l'instanciation de *sh2* pour ne conserver que *sh1*.
Sortie attendue :

```
scinfr066:POO Wilfrid$ ./a.out
Shape: 1
```

Exercice 4c : Afin de pouvoir récupérer et modifier les dimensions de la forme, créez des accesseurs *void setWidth(double w)* et *double getWidth()* ainsi que l'équivalent pour la hauteur. Ecrivez une méthode *void Print()* qui écrit la ligne qui était écrite par la fonction *main()*.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
   Shape sh;

   sh.Print();
   sh.setWidth(sh.getWidth()*2);
   sh.Print();
   return 0;
}
```

Et la sortie attendue celle-là :

```
scinfr066:POO Wilfrid$ ./a.out
Shape: 1x1 = 1
Shape: 2x1 = 2
```

Exercice 4d : On constate tout de suite qu'écrire des accesseurs pour tous les membres qui en nécessitent va être fastidieux et allonger le code de façon importante. Ecrivez une macro fonction *#define SetGetMacro(name, type)* qui va générer le code nécessaire pour déclarer et implémenter les accesseurs à la variable interne de nom *name* et de type *type*. Concrètement l'écriture suivante :

```
class MyClass
{
   SetGetMacro(Width, double);
}
```

Devra être convertie par le pré-compilateur de la façon suivante :

```
class MyClass
{
public:
   double getWidth()
   {
      return this->Width;
   }
   void setWidth(double w)
   {
      this->Width = w;
   }
private:
   double Width;
}
```

N'oubliez pas la compilation avec l'option -E afin de contrôler la façon dont le pré-compilateur va interpréter votre macro fonction.
Ici les 2 accesseurs sont publics mais il est courant qu'une variable interne ne puisse être que consultée et non modifiée de l'extérieur, ce qui conduit à rendre le getter public mais le setter privé. Ecrivez une macro fonction *#define GetMacro(name, type)*, équivalente à la précédente mais où le setter sera privé.
Remarque : dans ce cas on pourrait se passer de setter et modifier la variable avec une simple affectation puiqu'elle est interne. Il est toutefois préférable de le faire via un setter car cela évitera de devoir modifier le code partout où cela est nécessaire si d'aventure on décide par exemple de changer le nom de la variable où la structure de données à laquelle elle appartient.
Sortie attendue :

```
scinfr066:TP4 Wilfrid$ ./a.out
Shape: 0x0 = 0
Shape: 0x0 = 0
```

On remarque que les dimensions sont initialisées à 0. C'est normal puisque le fait de passer par des macro fonctions pour déclarer les variables internes a supprimé l'initialisation de celles-ci. On pourrait bien sur modifier la macro-fonction en conséquence mais on va se contenter de rajouter un appel à *setWidth()* et à *setHeight()* dans le main().

Exercice 4e : Appliquez le qualifier *const* à la méthode *getBox()* car elle n'est pas censée modifier l'état de l'objet. Vérifiez que vous ne pouvez pas y ajouter une instruction qui modifie l'état de l'objet. Faites de même pour la méthode *Print()*, que se passe-t-il ? Ajoutez les qualifier *const* partout où cela est nécessaire afin que le programme compile.
Le fait de devoir qualifier ainsi les méthodes peut vous sembler rébarbatif voire peu utile, dans la mesure où c'est vous-même qui écrivez le code des méthodes. Gardez cependant à l'esprit que ces qualifiers portent une sémantique, qui va s'imposer aux classes dérivées, qui seront peut être écrites par d'autres ou par vous-même à un moment où la contrainte sémantique ne vous sautera pas aux yeux. En imposant ces qualifiers, vous garantissez que la sémantique du modèle sera préservée par son implémentation.

Exercice 4f : Lors de la création d'une classe, un constructeur est créé par défaut, qui ne prend aucun paramètre et … ne fait rien. Le fait de déclarer ce constructeur va remplacer l'implémentation par défaut. La création d'une instance de classe est le bon moment pour acquérir les ressources nécessaires au bon fonctionnement de celle-ci et pour définir son état initial en initialisant ses différentes variables internes. Ceci est fait dans le constructeur. Il peut y avoir plusieurs constructeurs, qui se distinguent par le nombre et les types des paramètres formels qu'ils admettent, le choix du constructeur qui sera utilisé étant déterminé par la liste des paramètres effectifs qui sont transmis. Surchargez le constructeur par défaut de la classe *Shape* afin d'initialiser *Width* et *Height* à 1. Ajouter un constructeur qui prend en paramètres la largeur et la hauteur de la forme.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
   Shape sh1, sh2(2, 3);

   sh1.Print();
   sh2.Print();
   return 0;
}
```

Sortie attendue :

```
scinfr066:TP4 Wilfrid$ ./a.out
Shape: 1x1 = 1
Shape: 2x3 = 6
```

Exercice 4g : Modifiez la classe afin de mémoriser dans un tableau *dimensions* les dimensions de la forme, à savoir, dans cet ordre, les coordonnées en haut à gauche (l'origine de l'écran est son coin supérieur gauche), la largeur et la hauteur. Le tableau est alloué dans le constructeur et détruit dans le destructeur. L'origine par défaut de la forme est (0,0). Ajoutez un constructeur qui prend les 4 dimensions en paramètres. Répéter les mêmes instructions d'initialisation des variables locales dans les différents constructeurs est non seulement fastidieux mais également introduit une redondance toujours source d'erreurs. Utilisez la possibilité d'attribuer des valeurs par défaut aux paramètres des constructeurs afin de réduire au maximum ces redondances. Vous remarquez un petit problème. Maintenant créez une fonction privée *Init()* afin de résoudre ce problème.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
```

```
  Shape sh1, sh2(2, 3), sh3(3, 5, 2, 2);

  sh1.Print();
  sh2.Print();
  sh3.Print();
  return 0;
}
```

Sortie attendue :

```
macbook-pro-3:TP4 Wilfrid$ ./a.out
Shape: 0x0  1x1 = 1
Shape: 0x0  2x3 = 6
Shape: 3x5  2x2 = 4
```

Exercice 4h : Depuis la norme C++2011 il est possible qu'un constructeur appelle un autre constructeur. Ce mécanisme est appelé délégation car un constructeur va déléguer tout ou partie de son travail à un autre constructeur et il fonctionne pour toute méthode en C++. Toutefois ne confondez pas la délégation et le fait d'appeler une méthode depuis le corps d'une autre méthode, ce qui ne produira pas le même effet pour un constructeur. Supprimez la fonction *Init()* et trouvez la solution au problème posé grâce à cette nouvelle fonctionnalité. Si besoin compilez avec l'option *-std=c++11* afin de demander au compilateur de s'aligner sur cette norme, ce n'est pas forcément le cas par défaut.

Exercice 4i : Modifiez la méthode getBox() afin qu'elle retourne le vecteur à 4 double plutôt que la surface de la forme et calculez cette surface dans la fonction *main()*. Avant l'affichage multipliez la largeur de la forme par 2 et affichez-la. Cette instruction violant le principe d'encapsulation de l'orienté objet, faites en sorte qu'elle soit non compilable.