TP 5 : héritage et polymorphisme

Exercice 5a : Reprenez le code de l'exercice 4i et ajoutez au début de chaque constructeur et du destructeur une ligne de code qui affiche le nom de la classe, le nom de la méthode ainsi que ses paramètres (voir sortie attendue ci-dessous). Créez un type structuré *t_point* permettant de représenter un point dans le plan. Puis créez la classe *ImplicitShape*, héritée de *Shape*, qui va représenter des formes données par leur équation mathématique. On va se limiter au cercle pour simplifier les choses et mémoriser son centre de type *t_point* et son rayon.
Surchargez tous les constructeurs et destructeur de la classe *Shape* ainsi que la méthode *getName()* mais pas la méthode *Print()*. Observez bien la sortie et notamment quelles méthodes de quelle classe sont invoquées ainsi que l'ordre dans lequel elles le sont, afin de bien comprendre ce qui se passe.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
  ImplicitShape sh1;
  cout << " \n";
  ImplicitShape sh2(2, 3);
  cout << " \n";
  ImplicitShape sh3(3, 5, 2, 2);
  cout << " \n";
  sh1.Print();
  cout << " \n";
  sh2.Print();
  cout << " \n";
  sh3.Print();
  cout << " \n";
  return 0;
}
```

Sortie attendue :

```
macbook-pro-3:TP5 Wilfrid$ ./a.out
Shape::Shape(0,0,1,1)
Shape::Shape()
ImplicitShape::ImplicitShape()

Shape::Shape(0,0,2,3)
Shape::Shape(2,3)
ImplicitShape::ImplicitShape(2,3)

Shape::Shape(3,5,2,2)
ImplicitShape::ImplicitShape(3,5,2,2)

Shape: 0x0  1x1

Shape: 0x0  2x3

Shape: 3x5  2x2

ImplicitShape::~ImplicitShape()
Shape::~Shape()
ImplicitShape::~ImplicitShape()
Shape::~Shape()
ImplicitShape::~ImplicitShape()
Shape::~Shape()
```

Exercice 5b : Vous avez remarqué que la méthode *Print()* qui s'exécute est celle de la classe *Shape*, ce qui est normal puisqu'elle n'a pas été surchargée. Cependant nous avons pris la peine de redéfinir la méthode *getName()* dans la classe *ImplicitShape* mais c'est la version de la classe *Shape* qui est appelée par la méthode *Print()*. C'est le comportement par défaut. Afin ne pas avoir à redéfinir la méthode *Print()* lors de la spécialisation, puisqu'elle est inchangée, mais qu'elle appelle la méthode spécialisée, déclarez *getName()* virtuelle en précédant sa déclaration du qualifier *virtual*. Ceci rend la sélection de la méthode dynamique en l'associant à l'instance qui l'invoque plutôt qu'au type de la variable. Dynamique doit ici être compris comme "at run time" plutôt que "at compile time". En effet le compilateur ne peut se baser que sur le type de la variable pour déterminer quelle méthode doit être sélectionnée. C++ offre la possibilité de questionner le type de l'instance au moment de l'exécution – via l'opérateur typeid – et ainsi c'est la version correspondant à la classe de l'instance qui sera invoquée. Vérifiez-le en déclarant une référence de type *Shape* sur l'instance *sh1*, qui est de type *ImplicitShape*. Ce mécanisme est appelé polymorphisme et en C++ il doit être activé explicitement par utilisation du qualifier *virtual*. Une classe qui comporte des méthodes virtuelles est aussi dite polymorphique.

Exercice 5c : Il est même possible de déclarer une méthode virtuelle pure, ce qui signifie qu'elle n'admet pas de définition dans la classe générique. Par exemple il est impossible de calculer le rectangle englobant une forme 2D qui est inconnue. La définition de la forme figurant dans la classe spécialisée, en l'absence de spécialisation la fonction *getBox()* ne devrait pas pouvoir s'exécuter. Nous allons donc la déclarer virtuelle pure, en remplaçant sa définition par *=0*. Vérifiez que vous ne pouvez pas compiler le programme. C'est normal puisqu'il n'existe pas de définition de la méthode *getBox()*. Par conséquent une classe qui comporte des méthodes pures virtuelles ne pourra pas être instanciée, il sera indispensable de la spécialiser et d'instancier une spécialisation. On dit d'une telle classe qu'elle est abstraite et elle ne peut donc être utilisée que comme classe de base pour définir des classes spécialisées.
Reprenez la définition de la méthode *getBox()* et ajoutez-la à la classe *ImplicitShape*. Vous remarquez que le programme ne compile toujours pas. Une nouvelle erreur est apparue car vous tentez d'accéder, depuis une méthode de la classe spécialisée, à un membre privé de la classe générique. Il existe un qualifier qui permet qu'un membre soit inaccessible par une classe externe mais accessible par une classe dérivée, il s'agit de *protected*. Ainsi *dimensions* reste protégé mais accessible par les classes dérivées, la classe générique étant finalement une partie de la classe spécialisée. Vérifiez que désormais le programme compile bien.

Exercice 5d : Complétez la définition de la classe *ImplicitShape* en ajoutant un constructeur qui prend en paramètres le centre et le rayon du cercle, sous la forme de 3 *double*. Vérifiez que *getBox()* renvoie les bonnes valeurs. Créez maintenant la classe *PointSet* qui va représenter des formes données par une liste de points dans l'espace (polygone). On mémorise alors la liste ordonnée des sommets que l'on définit comme un *vector* de *t_point*. Cette classe admet un constructeur qui prend en paramètre une liste de points 2D, sous forme d'un *vector* de *t_point*.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
  ImplicitShape sh1(8, 8, 5);
  vector<t_point> pts;
  pts.push_back({2, 4});
  pts.push_back({5, 7});
  pts.push_back({1, 6});
  PointSet sh2(pts);
  sh1.Print();
  sh2.Print();
  return 0;
}
```

Sortie attendue :

```
macbook-pro-3:TP5 Wilfrid$ ./a.out
Shape::Shape(0,0,1,1)
Shape::Shape()
ImplicitShape::ImplicitShape(8,8,5)
Shape::Shape(0,0,1,1)
Shape::Shape()
PointSet::PointSet(vector<3 t_point>)
ImplicitShape: 3x3  10x10
PointSet: 1x4  4x3
PointSet::~PointSet()
Shape::~Shape()
ImplicitShape::~ImplicitShape()
Shape::~Shape()
```

Exercice 5e : Surchargez la méthode *Print()* de la classe *PointSet* afin qu'elle appelle la méthode *Print()* de la classe générique puis qu'elle affiche la liste des points de la forme. Modifiez la fonction *main()* afin de créer une nouvelle instance de *PointSet* par assignation. Vérifiez que les 2 instances sont bien équivalentes. Vous remarquez qu'une erreur est apparue à la fin de l'exécution du programme. A quoi est-elle due ?

La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
  ImplicitShape sh1(8, 8, 5);
  vector<t_point> pts;
  pts.push_back({2, 4});
  pts.push_back({5, 7});
  pts.push_back({1, 6});
  PointSet sh2(pts);
  sh1.Print();
  cout << " \n";
  sh2.Print();
  cout << " \n";
  PointSet sh3;
  cout << " \n";
  sh3 = sh2;
  cout << " \n";
  sh3.Print();
  cout << " \n";
  return 0;
}
```

Sortie attendue :

```
macbook-pro-3:TP5 Wilfrid$ ./a.out
Shape::Shape(0,0,1,1)
Shape::Shape()
ImplicitShape::ImplicitShape(8,8,5)
Shape::Shape(0,0,1,1)
Shape::Shape()
PointSet::PointSet(vector<3 t_point>)
ImplicitShape: 3x3  10x10

PointSet: 1x4  4x3
        (2, 4) (5, 7) (1, 6)

Shape::Shape(0,0,1,1)
Shape::Shape()
PointSet::PointSet()
```

```
PointSet: 1x4  4x3
        (2, 2) (5, 5) (1, 1)

PointSet::~PointSet()
Shape::~Shape()
PointSet::~PointSet()
Shape::~Shape()
a.out(42052,0x7fffa61ca380) malloc: *** error for object 0x7fa51740280: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Exercice 5f : Surchargez l'opérateur d'assignation de la classe *Shape* afin de résoudre le problème ci-dessus. Ajoutez une ligne d'impression dans l'opérateur afin de pouvoir détecter son invocation sur la sortie. Vous remarquez que *sh2* et *sh3* sont de type *PointSet* alors que nous n'avons pas redéfini l'opérateur d'assignation de la classe *PointSet* mais que celui de la classe *Shape* est néanmoins invoqué. Modifiez la méthode *Shape::Print()* afin qu'elle affiche la valeur du membre dimensions, autrement dit l'adresse qu'il contient.
Dans la fonction *main()* remplacez les 2 lignes suivantes :

```
PointSet sh3;
sh3 = sh2;
```

par celle-ci :

```
PointSet sh3 = sh2;
```

Que se passe-t-il ? Essayez de comprendre pourquoi à nouveau cette erreur à la fin.
Est-il nécessaire de surcharger l'opérateur d'assignation de la classe *PointSet* ? Recherchez sur internet la réponse à la question (je parle de rechercher dans la documentation officielle du C++, pas dans les discussions de comptoir ou pseudo cours en ligne !).
L'initialisation d'une instance consistant à fabriquer un clone d'une autre instance peut se faire par 2 opérations différentes :

- l'initialisation par assignation, que nous venons de voir et qui utilise l'opérateur d'assignation,
- l'initialisation par copie, qui repose sur un constructeur prenant en paramètre une autre instance et qui est l'objet de l'exercice suivant.

Exercice 5g : Créez un constructeur par copie pour chacune des 3 classes et vérifiez que vous obtenez bien une copie conforme qui est identique à celle obtenue par assignation. Quelle est la différence de comportement au niveau de l'invocation des méthodes de la classe générique entre l'initialisation par assignation et l'initialisation par copie ? Prenez le temps de réfléchir afin de comprendre la justification de cette différence.
Note : une déclaration de classe est dite en **forme canonique** ou **de Coplien** si elle comporte les méthodes suivantes : constructeur par défaut ou sans argument, construction par copie, opérateur d'assignation et destructeur. C'est une recommandation à laquelle il est **important** de se conformer car certaines classes de la librairie C++ standard ont quelquefois besoin que ces différentes méthodes soient définies. Si elles ne le sont pas, il sera impossible de compiler car par exemple le compilateur ne trouvera aucun constructeur de votre classe avec la signature exigée par la librairie.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
  ImplicitShape sh1(8, 8, 5);
  vector<t_point> pts;
  pts.push_back({2, 4});
  pts.push_back({5, 7});
```

```
 pts.push_back({1, 6});
 PointSet sh2(pts);
 sh1.Print();
 cout << " \n";
 sh2.Print();
 cout << " \n";
 PointSet sh3;
 cout << " \n";
 sh3 = sh2;
 cout << " \n";
 sh3.Print();
 cout << " \n";
 PointSet sh4(sh2);
 cout << " \n";
 sh4.Print();
 cout << " \n";
 return 0;
}
```

Exercice 5h : Ajoutez à la classe *Shape* un tableau *sceneBox* qui devra contenir les coordonnées des coins supérieur gauche et inférieur droit de la scène, autrement dit le plus petit rectangle contenant l'ensemble des formes. Afin que cette structure soit unique mais modifiable par chaque instance, utilisez le qualifier *static*. L'allocation mémoire du tableau *sceneBox* se fait lors de la première instanciation et sa libération lors de la destruction de la dernière instance. Chaque fois qu'une instance modifie ses coordonnées, par exemple lors de son instanciation, elle doit mettre à jour la taille de la scène si besoin. Créez une méthode *Shape::ComputeExtents()* qui réalise ce travail. Cette méthode doit donc pouvoir accéder au membre *dimensions* de chaque instance. Créez un *vector* statique *Shape::instances* afin de mémoriser un pointeur sur chaque instance créée et modifiez les constructeurs et destructeurs afin de mettre à jour cette structure lorsque cela est nécessaire. Enfin créez une méthode *Shape::PrintSceneBox()* qui imprime la taille de la scène.
La fonction *main()* doit être celle-ci :

```
int
main(int argc, char *argv[])
{
 ImplicitShape sh1(8, 8, 5);
 sh1.Print();
 sh1.PrintSceneBox();
 cout << " \n";
 vector<t_point> pts;
 pts.push_back({2, 8});
 pts.push_back({5, 16});
 pts.push_back({1, 11});
 PointSet sh2(pts);
 sh2.Print();
 sh2.PrintSceneBox();
 cout << " \n";
 pts[1].x = 15;
 pts[1].y = 2;
 PointSet sh3(pts);
 sh3.Print();
 sh3.PrintSceneBox();
 cout << " \n";
 return 0;
}
```

Sortie attendue :

```
Shape::Shape(0,0,1,1)
Shape::Shape()
ImplicitShape::ImplicitShape(8,8,5)
ImplicitShape: 3x3  10x10
Shape: scene box is (3,3) x (13,13)

Shape::Shape(0,0,1,1)
Shape::Shape()
PointSet::PointSet(vector<3 t_point>)
PointSet: 1x8  4x8
        (2, 8) (5, 16) (1, 11)
Shape: scene box is (1,3) x (13,16)

Shape::Shape(0,0,1,1)
Shape::Shape()
PointSet::PointSet(vector<3 t_point>)
PointSet: 1x2  14x9
        (2, 8) (15, 2) (1, 11)
Shape: scene box is (1,2) x (15,16)

PointSet::~PointSet()
Shape::~Shape()
PointSet::~PointSet()
Shape::~Shape()
ImplicitShape::~ImplicitShape()
Shape::~Shape()
```