TD/TP 8

Héritage, polymorphisme

Objectifs, méthodes

Ce 1er TD/TP d'application de l'héritage et du polymorphisme n'est programmé que sur une seule séance de 1H30. Nous travaillerons avec des petits « robots » virtuels qui avancent en mode console. La méthode proposée est de vous donner le code utilisateur des classes (code client) : **à vous d'écrire les classes qui permettent à ce code appelant de fonctionner**. Ceci est à rapprocher de la méthode « *test driven development* ».

Pour aborder confortablement les 1H30 de ce TD/TP il est indispensable d'avoir bien suivi et probablement relu le cours 8 sur l'héritage et le polymorphisme en C++. ***Le résultat du travail peut donner lieu à un dépôt campus, en monôme ou en binôme, si votre chargé de TP le demande, selon les modalités qu'il précisera.*** *Une estimation du volume de travail : environ 150 lignes de code à produire (Code only). Lisez le paragraphe 12 même si vous n'avez pas fini le reste.*

Les concepts du cours couverts par ce TP seront (pas nécessairement dans cet ordre)

- **`héritage simple`** d'une classe dérivée d'une classe de base
- **`hiérarchie d'héritage`** à plusieurs niveaux
- **`ajout de méthode`** dans une classe dérivée
- **`redéfinition de méthode`** dans une classe dérivée
- **`ajout d'attribut et de méthode`** dans une classe dérivée
- **`constructeurs et héritage`** appel au constructeur de la classe de base
- **`appels aux méthodes`** de classe dérivée et de classe de base depuis les méthodes
- **`méthodes virtuelles`** pour avoir du polymorphisme
- **`destructeur virtuel`** indispensable pour delete polymorphiquement
- **`conteneur polymorphe`** contenant un mix de (pointeurs sur) objets de types dérivés

1. Présentation de la famille Robot, diagramme de classes

La famille Robot est constituée de 6 catégories. Dans chaque catégorie les « robots » ont un mode de locomotion et/ou une représentation de leur trajet qui change.

étapes ↓ position →

Robot	RobotTeleporteur	RobotRapide	RobotTraceur	RobotCarburant	RobotEscargot
A A A A A A A A	B B B B B B B B	C C C C C C C C	D .D ..D ...D ....D .....D .....D .....D	E -E --E ---E ----E -----E -----E -----E	@ @ _@ _@ __@ __@ ___@ ___@

Voir page suivante pour un description précise de chaque catégorie.

➔ **Robot** : chaque Robot est caractérisé par un **aspect** qui est un seul caractère (`char` quelconque) et par une **position** entière. À sa création on précise son aspect (son caractère), et il part forcément à la position 0. Il peut **avancer** selon une certaine distance, et on peut demander à le voir à sa position actuelle à la console avec une commande **montrer**.

➔ **RobotTeleporteur** : chaque RobotTeleporteur a les mêmes caractéristiques qu'un Robot mais en plus il dispose d'une commande **teleporter** qui l'amène directement à une certaine position (reçue en paramètre).

➔ **RobotRapide** : chaque RobotRapide a les mêmes caractéristiques qu'un Robot mais quand on lui demande d'**avancer** il avance 2 fois plus qu'un simple Robot (à paramètre de distance équivalent).

➔ **RobotTraceur** : chaque RobotTraceur a les mêmes caractéristiques qu'un Robot avec en plus une trace laissée derrière lui, visible quand on regarde la position du RobotTraceur avec **montrer**. La caractère de **trace** est indiqué à la création d'un RobotTraceur (en plus de son aspect), il est donc paramétrable (ce n'est pas forcément le caractère `'.'`)

➔ **RobotCarburant** : chaque RobotCarburant a les mêmes caractéristiques qu'un RobotTraceur avec en plus un rayon d'action limité par son **carburant** initial (indiqué à la création, en plus de son aspect et de son caractère de trace). Quand on demande d'**avancer** d'une distance d, il perd d en carburant. Quand le carburant arrive à 0 il n'avance plus.

➔ **RobotEscargot** : chaque RobotEscargot a les mêmes caractéristiques qu'un RobotTraceur mais quand on lui demande d'**avancer** il avance 2 fois moins qu'un simple RobotTraceur (à paramètre de distance équivalent). Si on lui demande d'avancer de 1 plusieurs fois de suite, il avancera en fait de 1 une fois sur deux seulement. Un RobotEscargot est toujours créé avec un aspect `'@'` et un caractère traceur `'_'` (il n'est pas paramétrable à sa création).

Faites un diagramme de classes détaillé (noms de classes, attributs, méthodes, relations entre classes) de la famille Robot au complet. Pour le dépôt prévoyez un format électronique lisible (idéalement pdf) et nommez ce diagramme familleRobot (par exemple familleRobot.pdf). Mettez ce fichier dans le répertoire de projet que vous zipperez puis déposerez.

2. Classe Robot

Créez un nouveau projet et mettez en place la classe de base Robot de telle sorte que le code client suivant donne bien le résultat prévu pour Robot (page 1 en bas).

```
void testRobot()
{
    Robot x{'A'};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1); // Si on mettait 2 ici on irait 2x plus vite !
    }
}
```

Rappel : pour refaire l'indentation, dans CodeBlocks utiliser Plugins -> Source code formatter (Astyle).
Astuce pour répéter n fois un caractère `'c'` dans un affichage (sans faire de boucle) :
`std::cout << std::string(n, 'c') << ...`

3. Classe RobotTeleporteur

Complétez le projet pour mettre en place la classe RobotTeleporteur de telle sorte que le code client suivant donne bien le résultat prévu pour RobotTeleporteur (page 1 en bas). Ne pas utiliser l'héritage serait un hors-sujet (ce qui nous intéresse ici est l'héritage, pas juste le résultat).

```
void testRobotTeleporteur()
{
    RobotTeleporteur x{'B'};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1);
        if (i==3)
            x.teleporter(8);
    }
}
```

4. Classe RobotRapide

Complétez le projet pour mettre en place la classe RobotRapide de telle sorte que le code client suivant donne bien le résultat prévu pour RobotRapide (page 1 en bas). Ne pas utiliser l'héritage serait un hors-sujet (ce qui nous intéresse ici est l'héritage, pas juste le résultat).

```
void testRobotRapide()
{
    RobotRapide x{'C'};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1);
    }
}
```

5. Classe RobotTraceur

Complétez le projet pour mettre en place la classe RobotTraceur de telle sorte que le code client suivant donne bien le résultat prévu pour RobotTraceur (page 1 en bas). Ne pas utiliser l'héritage serait un hors-sujet (ce qui nous intéresse ici est l'héritage, pas juste le résultat). Noter que le caractère de trace est donné en **paramètre** du constructeur. Si on précise un autre caractère que `'.'` on obtiendra une trace avec cet autre caractère.

```
void testRobotTraceur()
{
    RobotTraceur x{'D', '.'};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1);
    }
}
```

6. Classe RobotCarburant

Complétez le projet pour mettre en place la classe RobotCarburant de telle sorte que le code client suivant donne bien le résultat prévu pour RobotCarburant (page 1 en bas). Ne pas utiliser l'héritage serait un hors-sujet (ce qui nous intéresse ici est l'héritage, pas juste le résultat). Héritez bien de la bonne classe ! Noter que le carburant initial est donné en dernier paramètre au constructeur. Ici sur ce cas particulier avec 5 au départ le RobotCarburant avance de 5 avant de ne plus pouvoir avancer.

```
void testRobotCarburant()
{
    RobotCarburant x{'E', '-', 5};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1);
    }
}
```

7. Classe RobotEscargot

Complétez le projet pour mettre en place la classe RobotEscargot de telle sorte que le code client suivant donne bien le résultat prévu pour RobotEscargot (page 1 en bas). Ne pas utiliser l'héritage serait un hors-sujet (ce qui nous intéresse ici est l'héritage, pas juste le résultat). Héritez bien de la bonne classe !

```
void testRobotEscargot()
{
    RobotEscargot x{};

    for (int i=0; i<8; ++i)
    {
        x.montrer();
        x.avancer(1);
    }
}
```

8. Résumé des classes qui marchent comme prévu

Complétez le main avec les appels à toutes les fonction de test « qui passent ». Commentez les appels aux tests qui ne passent pas (si vous n'avez pas réussi à faire toutes les classes). Le code déposé doit compiler, et le résultat de l'exécution doit correspondre à tout ou partie de ce qui est demandé.

```
int main()
{
    std::cout << "========= Tests directs =========\n";

    testRobot();
    testRobotTeleporteur();
    testRobotRapide();
    testRobotTraceur();
    testRobotCarburant();
    testRobotEscargot();

    ...
```

9. Fonction polymorphe

Modifiez votre hiérarchie de classes de telle sorte que vous puissiez utiliser le polymorphisme (il peut être nécessaire d'ajouter des précisions à la classe de base...). Ajoutez un « module » polymorphe.cpp et polymophe.h et développez y la fonction ***testRobotPolymorphe*** *telle que le code client suivant fasse* ***la même chose que la séquence de tests directs page précédente*** *à l'exception du RobotTeleporteur qui ne se téléportera pas.*

```
// ... suite du main ...
std::cout << "========= Tests polymorphisme =========\n";

Robot a{'A'};
RobotTeleporteur b{'B'};
RobotRapide c{'C'};
RobotTraceur d{'D', '.'};
RobotCarburant e{'E', '-', 5};
RobotEscargot f{};

testRobotPolymorphe( a );
testRobotPolymorphe( b );  // Ici même résultat pour B que pour A
testRobotPolymorphe( c );
testRobotPolymorphe( d );
testRobotPolymorphe( e );
testRobotPolymorphe( f );
```

Ne pas utiliser le polymorphisme serait un hors-sujet (ce qui nous intéresse ici est le polymorphisme, pas juste le résultat). Ce qui veut dire qu'on ne demande pas 6 fonctions surchargées ni des bricolages. Il y a bien une seule fonction (fonction = sous-programme on ne fait plus la distinction). Cette unique fonction testRobotPolymorphe **factorise** ce qui était répété dans les fonctions de test préalables. Le RobotTeleporteur nécessitant un appel que les autres types de Robot ne peuvent pas comprendre on le traitera comme les autre, on n'essaiera pas de savoir si on a affaire à un RobotTeleporteur ou pas, on n'essaiera pas d'appeler la méthode teleporter.

10. Conteneur polymorphe

Compléter le module polymorphe (polymorphe.cpp ...) avec le code suivant :

```
void testConteneurPolymorphe()
{
    std::list<Robot*> bots;

    remplirConteneurPolymorphe(bots);
    utiliserConteneurPolymorphe(bots);
    viderConteneurPolymorphe(bots);
}
```

Compléter le main avec le code suivant :

```
    std::cout << "========= Conteneur polymorphe =========\n";
    testConteneurPolymorphe();
```

Coder les 3 fonctions :
remplirConteneurPolymorphe met dedans 6 Robot de chacun des types (paramètres précédents)
utiliserConteneurPolymorphe fait avancer « simultanément » les 6 robots, ***voir page suivante***
viderConteneurPolymorphe libère les robots du conteneur

```
========= Conteneur polymorphe =========
====================
A
B
C
D
E
@
====================
 A
 B
  C
.D
-E
@
====================
  A
  B
    C
..D
--E
_@
====================
   A
   B
      C
...D
---E
_@
====================
    A
    B
        C
....D
----E
__@
====================
     A
     B
          C
.....D
-----E
__@
====================
      A
      B
            C
......D
-----E
___@
====================
       A
       B
              C
.......D
-----E
___@
```

11. Conteneur polymorphe avec Downcasting pour teleporter

Slide 66 du cours 8 vous trouverez un exemple de code permettant de « downcaster » càd retrouver un type dérivé spécifique à partir d'une référence/pointeur sur type de base. *Utiliser cette technique pour appeler la méthode teleporter spécifiquement pour le RobotTeleporteur.*

```
========= Conteneur polymorphe =========
====================
A
B
C
D
E
@
====================
 A
 B
  C
.D
-E
@
====================
  A
  B
    C
..D
--E
_@
====================
   A
   B
      C
...D
---E
_@
====================
    A
        B
        C
....D
----E
__@
====================
     A
         B
          C
.....D
-----E
__@
etc...
```

12. Packager, déposer

Vérifiez que tout compile bien. Neutralisez (commenter) les codes/appels qui ne marchent pas ou pas correctement ou supprimez les. Pensez au diagramme de classes demandé (exo 1). Zipper. Vérifiez que vous avez bien mis tous les sources dans le package ! Déposer.

Note aux utilisateurs de macOS / Linux / IDE autres que CodeBlocks : dans la mesure du possible essayez de nous livrer vos dépôts avec un projet CodeBlocks opérationnel (.cbp) parce que l'évaluation des projets avec des sources « en vrac » nous demande un temps supplémentaire (refaire un projet CodeBlocks...)

Idéalement, le main de test complet est :

```
int main()
{

    std::cout << "========= Tests directs =========\n";

    testRobot();
    testRobotTeleporteur();
    testRobotRapide();
    testRobotTraceur();
    testRobotCarburant();
    testRobotEscargot();


    std::cout << "========= Tests polymorphisme =========\n";

    Robot a{'A'};
    RobotTeleporteur b{'B'};
    RobotRapide c{'C'};
    RobotTraceur d{'D', '.'};
    RobotCarburant e{'E', '-', 5};
    RobotEscargot f{};

    testRobotPolymorphe( a );
    testRobotPolymorphe( b );
    testRobotPolymorphe( c );
    testRobotPolymorphe( d );
    testRobotPolymorphe( e );
    testRobotPolymorphe( f );

    std::cout << "========= Conteneur polymorphe =========\n";

    testConteneurPolymorphe();

    return 0;
}
```