Langage C++

- CIR2 -

ISEN ALL IS DIGITAL! NANTES | yncréa

TP9 : Polymorphisme

Ayoub KARINE (ayoub.karine@isen-ouest.yncrea.fr)

Ce TP reprend la hiérarchie de classes démarrée dans le TP n°8. Il a pour objectif d'illustrer comment le polymorphisme d'héritage permet de factoriser du code et d'écrire des traitements génériques.

1 - Constructeur de copie

1. Ajouter des constructeurs de recopie dans toutes vos classes.

Contraintes :

- Chaque constructeur doit initialiser uniquement les membres présents dans sa classe.
- Les constructeurs doivent afficher leur prototype et l'adresse de l'instance manipulée

2. Essayer d'anticiper ce qui est affiché par la fonction de test suivante :

```
void gasVehicleCopyTest(){
   GasVehicle original(Point(25,3), "XY-358-PQ", 60, 6.8, 95);
   GasVehicle copy = original;
   original.setIdentifier("XXX");
   original.affichage();
   copy.affichage();
}
```

3. Vérifier en exécutant cette fonction dans votre IDE. Le comportement de la fonction est-il conforme à vos attentes ? Si ce n'est pas le cas, faire les corrections nécessaires.

2 - Upcast et destruction

1. Essayer d'anticiper ce qui est affiché par la fonction de test suivante :

```
void dieselVehicleDestructionTest() {
   DieselVehicle* dieselVehicle = new DieselVehicle(Point(2, 10), "HD-888-ZY",
                                                    40, 5.5, true);
   Vehicule* vehicle = dieselVehicle; // upcast
   delete vehicle;
}
```

2. Si vous n'avez qu'un affichage de destructeur, quelque chose ne va pas dans votre code. Faites les corrections nécessaires.

3 - Modèle de consommation spécifique

On considère que les véhicules à essence possèdent le modèle de consommation suivant :

- la consommation de référence correspond à un indice d'octane de 95%
- chaque point d'indice d'octane en plus réduit la consommation de carburant de 1%

Par exemple, si un véhicule à essence a une consommation de 5L/100km avec un indice d'octane de 95%, cette consommation sera de 4,85L/100km avec un indice 98% (4,85 correspondant à 97% de 5).

1. Modifiez votre code pour implémenter cette spécificité.
   Contrainte : il est interdit de surcharger la méthode moveTo().
2. **Test sans polymorphisme** : Exécuter la fonction de test suivante :

```
void octaneRatingConsumptionTests(){
   GasVehicle gasVehicle95(Point(0,0), "XY-358-PQ", 30, 5, 95);
   cout << "Traveled distance : " << gasVehicle95.moveTo(0, 100) << endl;
gasVehicle95.affichage();
   GasVehicle gasVehicle98(Point(0,0), "HD-888-ZY", 30, 5, 98);
   cout << "Traveled distance : " << gasVehicle98.moveTo(0, 100) << endl;
gasVehicle98.affichage();
}
```

Et vérifiez que l'affichage 1 est de la forme :
[XY-358-PQ] position : (0,100), consumption: 5, fuel left: 25, octane rating: 95
et que l'affichage 2 est de la forme :
[HD-888-ZY] position : (0,100), consumption: 4.85, fuel left: 25.15, octane rating: 98

3. **Test avec polymorphisme** :

Soit la fonction de test suivante :

```
void octaneRatingConsumptionTestsWithUpcast(){
   GasVehicle gasVehicle95(Point(0,0), "XY-358-PQ", 30, 5, 95);
   GasVehicle gasVehicle98(Point(0,0), "HD-888-ZY", 30, 5, 98);
   vector<Vehicule*> vehicles = { &gasVehicle95, &gasVehicle98 };
   octaneRatingConsumptionTestsWithUpcast(vehicles);
}
```

a. Implémenter la fonction :

```
void octaneRatingConsumptionTestsWithUpcast(const vector<Vehicle*>& vehicles)
```

Pour que `octaneRatingConsumptionTestsWithUpcast()` ait exactement le même comportement que `octaneRatingConsumptionTests()`