TP n° 12

Transport de véhicules

On désire modéliser l'activité d'un ferry qui transporte différentes catégories de véhicules et leurs passagers ; le mode de calcul du tarif de transport d'un véhicule dépend de ses caractéristiques propres, le tarif de transport d'une personne est fixé à 15 €.

1 Modélisation des véhicules

On veut que toutes les catégories de véhicules fournissent obligatoirement les services suivants :

1. consultation du nombre de personnes transportées ;
2. consultation de la longueur (entier) ;
3. calcul du tarif de transport (réel) ;
4. affichage (compact) de ses caractéristiques (opérateur ≪).

La définition de la classe abstraite Vehicule vous est donnée ; il vous reste à programmer son implémentation.

Question 1 : Justifiez les choix faits dans la classe Vehicule : méthodes virtuelles pures ou non, méthodes non virtuelles ; expliquez comment rendre polymorphe l'opérateur d'affichage.

1.1 Caractéristiques de quelques véhicules

Important : Pensez à programmer les différentes classes de véhicules présentées ci-dessous en donnant la possibilité d'étendre la hiérarchie.

1.1.1 Auto

— longueur égale à 2 unités ; nombre de personnes quelconque ;
— tarif = 350 € s'il s'agit d'un véhicule tout-terrain, 100 € sinon, à quoi s'ajoute le tarif de transport des passagers.

Programmez la classe Auto.

1.1.2 Bus

— longueur quelconque ; nombre de personnes quelconque ;
— tarif = 200 € + 50 € par unité de longueur, à quoi s'ajoute le tarif de transport des passagers.

Programmez la classe Bus.

2 Modélisation du ferry

Voici les caractéristiques d'un ferry :

— un ferry a une *capacité* limitée, aussi bien en unités de longueur qu'en nombre de passagers transportés ;
— on doit pouvoir placer dans un ferry donné un véhicule de n'importe quelle catégorie.

Les véhicules transportés dans un ferry seront placés dans l'un des conteneurs de la bibliothèque standard vector, deque ou list ;

Question 2 : quel doit être le type des éléments de ce conteneur ? Expliquez.

On veut doter la classe Ferry des méthodes suivantes :

— méthode qui *ajoute* un véhicule dans un ferry si c'est possible ; cette méthode renvoie un booléen pour indiquer si l'ajout a pu se faire.
— méthode qui calcule le *tarif* total de transport des véhicules présents dans le ferry ;
— opérateur ≪ qui *affiche* le contenu du ferry : détail de chaque véhicule, longueur disponible, nombre de places disponibles, tarif total des véhicules transportés.

1. La définition (partielle) de la classe Ferry vous est donnée ; complétez-la et programmez son implémentation en veillant à permettre aisément le remplacement d'un conteneur (vector, deque ou list) par un autre ; en particulier, définissez le type du conteneur (typedef).
   Il n'est pas utile de programmer le corps du destructeur avant d'aborder la partie 3.
2. Utilisez le programme fourni client.cc pour tester le fonctionnement d'un ferry.
3. Changez le conteneur utilisé pour les véhicules (ceci doit entraîner la modification d'au plus deux lignes), recompilez et testez.

3 Clonage polymorphe

Problème :

— le programme client, qui crée les instances de véhicules et en est propriétaire ne peut les détruire car il n'a aucun moyen d'accéder aux véhicules placés dans le ferry.
— Le ferry, qui « stocke » des véhicules, n'a pas le droit de les détruire, car il n'est pas leur propriétaire (le programme client pourrait vouloir se servir des véhicules après la destruction du ferry).

Solution :

— première possibilité : le programme client conserve les véhicules dans une structure qui lui est propre, c'est-à-dire un deuxième conteneur, ce qui lui permet de détruire les instance de véhicules en fin d'utilisation.
— deuxième possibilité (complémentaire de la première) : la méthode ajouter va placer dans le ferry une *copie* du véhicule paramètre, dont le ferry sera *propriétaire* et qu'il devra donc se charger de détruire .

C'est cette deuxième possibilité qu'il vous est demandé de programmer ici.

La méthode naturelle pour obtenir une copie d'une instance est d'utiliser le *constructeur de copie* ; or le constructeur de copie n'est pas polymorphe : il n'est donc pas possible de l'utiliser pour obtenir une copie d'une instance d'une classe fille désignée par un pointeur ou une référence sur une classe mère (abstraite ou concrète).

Il faut donc munir la hiérarchie de véhicules d'une *méthode polymorphe de clonage*, qui sera implémentée dans chaque classe fille (à l'aide du *constructeur de copie* de la classe correspondante).

Question 3 : Expliquez votre solution.

Faites les modifications nécessaires dans la classe Ferry, la hiérarchie de véhicules et le programme client pour que le ferry gère des copies des véhicules qui y sont ajoutés.

Avant de passer à la suite, vérifiez la gestion mémoire de votre programme. (valgrind)

4 Trier les véhicules

4.1 Problème

Ajoutez dans la classe ferry une méthode trier qui trie le contenu du ferry avec l'algorithme sort de la bibliothèque standard ; celui-ci compare les éléments avec l'opérateur <. Comme indiqué dans la documentation de la STL (disponible par exemple à l'adresse : http://www.sgi.com/tech/stl/), cet algorithme ne s'applique qu'à des itérateurs à accès direct ; le conteneur list n'est donc plus adapté ici.

```
template <class RandomAccessIterator>
void std::sort(RandomAccessIterator first, RandomAccessIterator last);
```

NB : l'algorithme sort est défini dans le fichier algorithm de la bibliothèque standard.

Le résultat du tri n'est (probablement) pas celui auquel on pouvait s'attendre.

Question 4 : D'après vous, selon quel critère s'effectue le tri ? Expliquez.

4.2 Solution

Pour résoudre ce problème, il faut passer un paramètre supplémentaire à l'algorithme sort : une instance de comparateur ; ce comparateur sera appelé par l'algorithme sort chaque fois qu'il a besoin de comparer deux éléments du conteneur. Voici la signature de cette deuxième version de l'algorithme sort :

```
template <class RandomAccessIterator, class Comparator>
void std::sort( RandomAccessIterator first,
                RandomAccessIterator last,
                Comparator compare);
```

4.2.1 Objet fonctionnel

Un *objet fonctionnel* est une instance d'une classe qui implémente l'opérateur « *appel de fonction* », c'est-à-dire l'opérateur operator(), et se comporte comme une fonction : appel avec un nom, des () et des paramètres ;

exemple : soit maFonction une instance d'une telle classe ; l'expression maFonction(param1, param2) est traduite par le compilateur en maFonction.operator()(param1, param2).

Il suffit donc de programmer le corps de operator() pour lui donner la signification souhaitée.

4.2.2 Comparateur

Un *comparateur* est un objet fonctionnel dont l'opérateur operator() est programmé pour lui donner le comportement d'un opérateur de comparaison. Dans ce cas, la signature de cet opérateur devient par exemple :

```
bool operator() (const T & param1, const T & param2) const;
```

ou

```
bool operator() (T param1, T param2) const;
```

remarque : T est le type des éléments à comparer ; dans notre cas, c'est celui des éléments du conteneur.

exemple : soit monComparateur une instance d'une telle classe ; on peut écrire l'expression :
`if (monComparateur(param1, param2)) { ...} else { ... }`

4.3 Premier tri

4.3.1 par longueur croissante

Programmez la classe ComparerLongueurVehicules dont l'opérateur operator() compare deux éléments du conteneur et renvoie vrai si la longueur du premier est inférieure à celle du second.

Modifiez la méthode trier de la classe ferry pour qu'elle effectue le tri à l'aide une instance de ce comparateur puis testez.

4.3.2 choix du sens de tri

On veut donner au client le choix du sens de tri : ordre croissant ou décroissant ; pour cela, il suffit de :

- définir un attribut booléen dans la classe ComparerLongueurVehicules ;
- initialiser cet attribut dans le constructeur ; vous donnerez au paramètre du constructeur une valeur par défaut judicieuse ;
- modifier le corps de l'opérateur pour changer le sens de la comparaison en fonction du booléen ;
- passer un booléen en paramètre à la méthode de tri, lui aussi avec une valeur par défaut...

Programmez cette modification et triez le ferry, par ordre croissant puis par ordre décroissant.

4.4 Trier selon plusieurs critères de tri

On veut avoir la possibilité de trier les véhicules du ferry selon différents critères :
- par longueur croissante ou décroissante
- par nombre de passagers croissant ou décroissant
- par tarif croissant ou décroissant
- etc...

4.4.1 idée 1 : une méthode de tri par critère de tri

Une première idée consiste à créer une classe de comparateur ainsi qu'une méthode de tri dans la classe ferry pour chaque nouveau critère de tri. Cette solution fonctionne, mais n'est pas très souple ; en particulier, le client ne peut pas décider de trier les véhicules du ferry avec d'autres critère que ceux définis par les méthodes de tri de la classe ferry.

4.4.2 ideée 2 : paramétrer la méthode de tri par un comparateur

Pour donner un peu plus de souplesse, on propose le fonctionnement suivant : on ne définit dans la classe Ferry qu'une seule méthode trier, qui sera *paramétrée par un comparateur* ; c'est le client qui se charge de programmer les comparateurs qui l'intéressent et de passer à la méthode de tri le comparateur qui correspond au critère de tri de son choix.

Question 5 : quel doit être le type du paramètre de la méthode trier ? Proposez une organisation des classes de comparateurs qui permette d'appeler cette méthode de tri avec différents comparateurs.

1. Programmez les classes qui permettent de trier les véhicules selon au moins deux des critères ci-dessus (en conservant la possibilité de choisir le sens de comparaison).
2. Modifiez la méthode trier ainsi que le programme client afin de trier les véhicules selon (au moins) deux critères différents.

Problème : soit le compilateur refuse de compiler, soit il compile, mais l'exécution ne donne pas les résultats escomptés.

Que se passe-t-il ? Tout simplement, le paramètre *comparateur* de l'algorithme sort est passé **par valeur**, ce qui est incompatible avec la liaison dynamique rendue nécessaire par l'organisation des classes de comparateurs définie ci-dessus.

4.4.3 ébauche de solution

Utiliser la généricité et paramétrer la classe ferry avec le type du comparateur :

```
/// @param TComparateur : type du comparateur
template <class TComparateur>
class Ferry {
...
public:
  /** trier le ferry selon l'ordre défini par le comparateur
      @param comparateur : le comparateur
   */
    void trier(TComparateur comparateur = TComparateur());
...
};
```

Du coup, le paramètre de la méthode trier est du *même type* que le paramètre de généricité de la classe ferry et lors de l'instanciation d'une classe de ferry, par exemple ainsi :

```
Ferry<ComparerLongueurVehicules> jules;
```

le type du comparateur de trier sera celui du comparateur effectivement utilisé ; il n'y a plus besoin de liaison dynamique et il n'y a plus de problème de troncature.

Problème : ceci ne permet pas de trier le ferry avec différents comparateurs ; en effet, le comparateur est déterminé lors de l'instanciation de la classe ferry et il n'est pas possible de passer un autre comparateur en paramètre de trier.

4.4.4 autre proposition

Utiliser la généricité mais paramétrer uniquement la *méthode de tri* avec le type du comparateur :

```
class Ferry {
public:
  /** trier le ferry selon l'ordre défini par le comparateur
      @param TComparateur : type du comparateur
      @param comparateur : le comparateur
   */
  template <class TComparateur>
    void trier(TComparateur comparateur = TComparateur());
};
```

Programmez cette version et testez avec différents critères de tri ; au passage, notez la valeur par défaut du paramètre qui fait appel au constructeur sans paramètre de la classe du comparateur : ceci permet l'appel de la méthode de tri sans fournir le comparateur...

4.5 un patron de conception

En regardant les différents comparateurs, on s'aperçoit qu'on a deux ou trois classes concrètes quasiment identiques :

- attribut booléen pour mémoriser le sens du tri, initialisé par un constructeur
- opérateur () dont l'algorithme (très simple) est le même pour toutes les classes ; seul change le critère de comparaison.

Une première amélioration consiste à déplacer l'attribut booléen dans la classe mère, soit en accès protégé, soit en accès privé avec accesseurs protégés.

Une deuxième amélioration consiste à programmer la logique de l'opérateur () dans la classe mère, sous forme d'une méthode *non virtuelle* (donc qui ne fait pas l'objet de liaison dynamique) et faire appel à une *méthode virtuelle pure* (compare) pour comparer les caractéristiques des véhicules ; les classes filles concrètes n'ont plus qu'à implémenter cette méthode compare qui compare deux véhicules sans se soucier du sens de tri.

Cette façon de concevoir l'opérateur () suit un patron de conception nommé *patron de méthode* : celui-ci consiste à définir la structure d'un algorithme dans une classe de base (éventuellement abstraite), à l'aide d'opérations dont certaines peuvent être abstraites et qui devront être implémentées par les classe concrètes dérivées.

Modifiez l'organisation de vos classes pour programmer l'opérateur () selon un *patron de méthode*.

5 Autres véhicules

5.1 Ambulance

Une ambulance est une auto, dont le tarif de transport est toujours nul, qu'il s'agisse d'un véhicule tout terrain ou non.

Programmez la classe **Ambulance** et complétez le programme client de test..

Remarque : cet ajout ne doit normalement entrainer aucune modification des autres classes.

5.2 Cycle

Un cycle ne peut transporter qu'une personne ; sa longueur est d'une unité et son tarif de 20 € à quoi s'ajoute le tarif passager.

Programmez la classe **Cycle** ; même remarque.