MOOC Intro POO C++
Tutoriels semaine 5 : polymorphisme

Les tutoriels sont des exercices qui reprennent des exemples similaires à ceux du cours et dont le corrigé est donné progressivement au fur et à mesure de la donnée de l'exercice lui-même.

Ils sont conseillés comme un premier exercice sur un sujet que l'étudiant ne pense pas encore assez maîtriser pour aborder par lui-même un exercice «classique».

Les solutions sont fournies au fur et à mesure sur les pages paires.

> Cet exercice correspond à l'exercice «*pas à pas*» page 145 de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Introduction

Le but de cet exercice est d'illustrer la notion de polymorphisme en utilisant une collection hétérogène de véhicules.

Dans le fichier `aeroport.cc`, commencez par définir des classes `Vehicule`, `Avion` et `Voiture`. Les avions et les voitures *sont des* véhicules. Dotez chacune de ces classes d'une méthode permettant de les afficher (message quelconque au choix).

On souhaite ensuite créer une classe `Aeroport` ayant à gérer un ensemble de véhicules constitué à la fois de voitures et d'avions.

Sans polymorphisme, du fait que l'on manipule deux types d'objets différents, on est obligé de créer deux tableau différents ; ceci est parfaitement valable en soit, mais pour les besoins de l'exercices supposons ici que l'on souhaiterait plutôt les gérer ensembles comme une collection de véhicules, sans distinction à ce stade (collection).

Définissez la classe `Aeroport` dans cet esprit. Dotez-la de trois méthodes :

- `affiche_vehicules(ostream&)` permettant d'afficher tous les véhicules de l'aéroport ;
- `ajouter_vehicules(...)` permettant d'ajouter un véhicule à l'aéroport ;
- `vider_vehicules()` supprimant tous les véhicules de l'aéroport

[Essayez de le faire par vous même avant de regarder la solution qui suit]

(solution page suivante)

Solution :

On pourrait imaginer :

```
class Aeroport {
public:
 void affiche_vehicules(ostream&);
 void ajouter_vehicules(Vehicule const&);
 void vider_vehicules();
protected:
  vector<Vehicule> vehicules;
};
```

mais cela ne permettrait pas aux voitures et avions de s'afficher en tant que tel, bref, en deux mots, d'utiliser le polymorphisme.

Il fautpour cela utiliser des **références** ou des **pointeurs** sur les objets plutôt que les objets eux-mêmes. Comme nous ne pouvons pas mettre de référence dans un `vector`, nous devons donc utiliser ici un **tableau dynamique de pointeurs**.

Notre classe `Aeroport` s'écrit alors :

```
class Aeroport {
public:
 void affiche_vehicules(ostream&) const;
 void ajouter_vehicules(Vehicule*);
 void vider_vehicules();
protected:
  vector<Vehicule*> vehicule;
};

void Aeroport::affiche_vehicule(ostream& sortie) const {
  for (auto const& vehicule : vehicules) {
      vehicule->affiche(sortie);
  }
}

void Aeroport::ajouter_vehicule(Vehicule* v) {
   vehicules.push_back(v);
}

void Aeroport::vider_vehicules() {
  vehicules.clear();
}
```

Pour que la résolution dynamique des liens puisse être mise en œuvre, il faut aussi que les méthodes que l'on invoque sur les objets de type `Vehicule` soient **virtuelles.**

De cette façon, si `vehicules[i]` est un (pointeur sur un) avion `vehicules[i]->afficher()` fera appel à la méthode d'affichage de la classe `Avion` et non la méthode de `Vehicule`.

Notre classe `Vehicule` doit donc être :

```
class Vehicule {
public:
  Vehicule(// par exemple...
           string marque, unsigned int date, double prix
          );
  virtual void affiche(ostream&) const;
  virtual ~Vehicule() {}

protected:
 // par exemple....
  string       marque       ;
  unsigned int date_achat   ;
  double       prix_achat   ;
  double       prix_courant ;
};
```

Voilà, notre classe `Aeroport` est maintenant utilisable.

Pour que le codage en soit complètement satisfaisant, il faudrait cependant pouvoir offrir le moyen d'éventuellement pouvoir libérer la mémoire des objets mis dans la collection (au cas où la fonction qui les a créés et ajouté à la collection souhaite les supprimer).
(En tout rigueur il faudrait aussi pouvoir en supprimer 1 élément précis et fournir une méthode de copie profonde au cas où).

Ajoutez une méthode `supprimer_vehicules()` qui effectue ce nettoyage mémoire.

Vous pouvez alors tester avec le `main()` suivant :

```
int main() {
   Aeroport gva;
   gva.ajouter_vehicule(new
       Voiture("Peugeot", 1998, 147325.79, 2.5, 5, 180.0, 12000));
   gva.ajouter_vehicule(new
       Voiture("Porsche",  1985, 250000.00, 6.5, 2, 280.0, 81320));
   gva.ajouter_vehicule(new
       Avion("Cessna",     1972, 1230673.90, HELICES,   250));
   gva.ajouter_vehicule(new
       Avion("Nain Connu", 1992, 4321098.00, REACTION, 1300));
   gva.ajouter_vehicule(new
       Voiture("Fiat",    2001,   7327.30, 1.6, 3,  65.0,  3000));

  gva.affiche_vehicules(cout);

  // pour être propre, le main() demande (à gva) de libérer
  // la mémoire qu'il (main) a alloué (et c'est à lui (main) de le faire
  // pas au destructeur de gva qui n'a pas alloué cette mémoire)
  gva.supprimer_vehicules();

  return 0;
}
```

[NOTE : Comme dans toute collection hétérogène construite par pointeurs, il faut faire attention à la gestion de la mémoire.
Nous avons volontairement écarté ici l'aspect allocation dynamique (qui n'est pas nécessaire mais souvent utile) ici, et laissez le soin à la fonction `main()` de s'en charger de façon simple, mais si cette allocation dynamique devait être géré au niveau de la classe `Aeroport`, il faudrait bien entendu le faire proprement, avec constructeur de copie, copie profonde, libération par le destructeur, surcharge de l'opérateur d'affectation (=), etc. **]**

Vous pouvez trouver ici un code complet possible :

(solution page suivante)

Solution :

```
#include <algorithm> // pour max()
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// ---------------------------------------------
// ------------ La classe Vehicule -----------
class Vehicule {
public:
  Vehicule(string marque, unsigned int date, double prix);
  virtual void calculePrix();
  virtual void affiche(ostream&) const;
  virtual ~Vehicule() {}

protected:
  string       marque       ;
  unsigned int date_achat   ;
  double       prix_achat   ;
  double       prix_courant ;
};

Vehicule::Vehicule(string marque, unsigned int date, double prix)
 :  marque(marque), date_achat(date), prix_achat(prix), prix_courant(prix)
{}

void Vehicule::affiche(ostream& affichage) const {
  affichage << "marque : " << marque
            << ", date d'achat : " << date_achat
            << ", prix d'achat : " << prix_achat
            << ", prix actuel : "  << prix_courant
            << endl;
}

void Vehicule::calculePrix() {
  double decote((2003 - date_achat) * .01);
  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}

// ---------------------------------------------
// ------------ La classe Voiture  -----------
class Voiture : public Vehicule {
public:
  Voiture(string marque, unsigned int date, double prix,
          double cylindree, unsigned int portes, double cv, double km);
  void calculePrix();
  void affiche(ostream&) const;

protected:
  double cylindree       ;
  unsigned int nb_portes ;
  double puissance       ;
  double kilometrage     ;
};

Voiture::Voiture(string marque, unsigned int date, double prix,
```

```
   double cylindree, unsigned int portes, double cv,
   double km)
 : Vehicule(marque, date, prix)
 , cylindree(cylindree), nb_portes(portes), puissance(cv),  kilometrage(km)
{}

void Voiture::affiche(ostream& affichage) const {
  affichage << " ---- Voiture ----" << endl;
  Vehicule::affiche(affichage);
  affichage << cylindree << " litres, "
            << nb_portes << " portes, "
            << puissance << " CV, "
            << kilometrage << " km." << endl;
}

void Voiture::calculePrix() {
  double decote((2003 - date_achat) * .02);
  decote += 0.05 * kilometrage / 10000.0;
  if (marque == "Fiat" or marque == "Renault")
    decote += 0.1;
  else if (marque == "Ferrari" or marque == "Porsche")
    decote -= 0.2;

  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}


// -------------------------------------------
// ----------- La classe Avion    ----------
enum Type_Avion { HELICES, REACTION };
class Avion : public Vehicule {
public:
  Avion(string marque, unsigned int date, double prix,
        Type_Avion moteur, unsigned int heures);
  void calculePrix();
  void affiche(ostream&) const;

protected:
  Type_Avion   moteur     ;
  unsigned int heures_vol ;
};

Avion::Avion(string marque, unsigned int date, double prix,
      Type_Avion moteur, unsigned int heures)
 : Vehicule(marque, date, prix)
 , moteur(moteur), heures_vol(heures)
{}

void Avion::affiche(ostream& affichage) const {
  affichage << " ---- Avion à ";
  if (moteur == HELICES) affichage << "hélices";
  else                   affichage << "réaction";
  affichage << " ----" << endl;
  Vehicule::affiche(affichage);
  affichage << heures_vol << " heures de vol." << endl;
}
```

```
void Avion::calculePrix() {
  double decote;
  if (moteur == HELICES)
    decote = 0.1 * heures_vol / 100.0;
  else
    decote = 0.1 * heures_vol / 1000.0;

  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}


// -----------------------------------------
// ------------ La classe Aeroport   ------------

typedef vector<Vehicule*> Vehicules;

class Aeroport {
public:
  void affiche_vehicules(ostream&) const;
  void ajouter_vehicule(Vehicule* v) { vehicules.push_back(v); }
  void vider_vehicules() { vehicules.clear(); }
  void supprimer_vehicules();

protected:
  Vehicules vehicules;
};

void Aeroport::affiche_vehicules(ostream& out) const {
  for (auto vehicule :  vehicules) {
    vehicule->calculePrix();
    vehicule->affiche(out);
  }
}

void Aeroport::supprimer_vehicules(){
  for (auto vehicule :  vehicules) delete vehicule;
  vider_vehicules();
}


// ========================================================================
// un petit main pour tester tout ca
int main() {
   Aeroport gva;
   gva.ajouter_vehicule(new
       Voiture("Peugeot", 1998, 147325.79, 2.5, 5, 180.0, 12000));
   gva.ajouter_vehicule(new
       Voiture("Porsche", 1985, 250000.00, 6.5, 2, 280.0, 81320));
   gva.ajouter_vehicule(new
       Avion("Cessna",     1972, 1230673.90, HELICES,   250));
   gva.ajouter_vehicule(new
       Avion("Nain Connu", 1992, 4321098.00, REACTION, 1300));
   gva.ajouter_vehicule(new
       Voiture("Fiat",    2001,   7327.30, 1.6, 3,  65.0,  3000));

  gva.affiche_vehicules(cout);
```

```
  // pour être propre, le main() demande (à gva) de libérer
  // la mémoire qu'il (main) a alloué (et c'est à lui (main) de le faire
  // pas au destructeur de gva qui n'a pas alloué cette mémoire)
  gva.supprimer_vehicules();

  return 0;
}
```
