TP POO C++ n°3

Sommaire

Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++.

Rappels

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Concrètement, un **objet** est une **structure de données** (**ses attributs** c.-à-d. des variables) qui définit son **état** et une **interface** (**ses méthodes** c.-à-d. des fonctions) qui définit son **comportement**.

Une **classe déclare des propriétés communes** à un ensemble d'objets.

Un **objet** est créé à partir d'un **modèle** appelé **classe**. Chaque objet créé à partir de cette classe est une **instance** de la classe en question.

Travail demandé

Nous allons successivement découvrir les notions suivantes :
– l'utilisation des membres données statiques et des fonctions statiques
– la surcharge d'opérateurs de flux << et >>

On vous fournit un programme `testPoint.cpp` où vous devez décommenter progressivement les parties de code source correspondant aux questions posées.

Membres données statiques

Un membre donnée déclaré avec l'attribut `static` est **partagé par tous les objets de la même classe**. Il existe même lorsque aucun objet de cette classe n'a été créé.

Un membre donnée statique doit être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

En général, son initialisation se fait dans le fichier `.cpp` de définition de la classe.

On va utiliser un membre statique `nbPoints` pour **compter le nombre d'objets `Point` créés à un instant donné**.

On **déclare** un membre donnée statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   ...
};
```

Point.h

Il faut maintenant `initialiser` ce membre statique et `compter/décompter` le nombre d'objets créés et détruits :

```
int Point::nbPoints = 0; // initialisation d'un membre statique

Point::Point() // Constructeur
{
   x = 0.;
   y = 0.;
   nbPoints++; // un objet Point de plus
}

...

Point::~Point() // Destructeur
{
   nbPoints--; // un objet Point de moins
}
```

Point.cpp

Tous les constructeurs de la classe `Point` doivent incrémenter le membre statique `nbPoints` !

Fonctions membres statiques

Lorsqu'une fonction membre a **une action indépendante d'un quelconque objet de sa classe**, on peut la déclarer avec l'attribut `static`.

Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Les fonctions membre statiques :
- ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.
- peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

On va utiliser une fonction membre statique `compte()` pour **connaître le nombre d'objets `Point` existants à un instant donné**.

On **déclare** une fonction membre statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   public:
      ...
      static int compte(); // je suis une méthode statique

};
```

Point.h

Il faut maintenant **définir** cette méthode statique :

```
// je retourne le nombre d'objets Point existants à un instant donné
int Point::compte()
{
   return nbPoints;
}
```

Point.cpp

Question 1. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter les membres statiques `nbPoints` et `compte()`. Décommenter les parties de code de cette question dans le fichier `testPoint.cpp` et tester. Pensez à bien compter tous les objets créés !

```
Point p0, p1(4, 0.0), p2(2.5, 2.5);
Point *pp = new Point(1,1);

cout << "Il y a " << Point::compte() << " points\n"; // Affiche : Il y a 4 points

delete pp;

cout << "Il y a " << Point::compte() << " points\n"; // Affiche : Il y a 3 points
```

Les services, qui permettent de calculer la **distance** entre 2 points et le **milieu** de 2 points, ont une action indépendante et peuvent être déclarés comme statique.

Question 2. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter les méthodes statiques `distance()` et `milieu()`. Décommenter les parties de code de cette question dans le fichier `testPoint.cpp` et tester. Vérifier que vous obtenez ceci.

```
Point p1(4, 0.0), p2(2.5, 2.5);

double distance = Point::distance(p1, p2);
cout << "La distance entre p1 et p2 est de " << distance << endl;

Point pointMilieu = Point::milieu(p1, p2);
cout << "Le point milieu entre p1 et p2 est "; pointMilieu.affiche();
```

Ce qui donne :

```
La distance entre p1 et p2 est de 2.91548
Le point milieu entre p1 et p2 est <3.25,1.25>
```

La surcharge des opérateurs de flux << et >>

Rappels : Un **flot** est un **canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information**. Ce canal est associé à un périphérique ou à un fichier.

Un **flot d'entrée** est un objet de type `istream` tandis qu'un **flot de sortie** est un objet de type `ostream`.

Le flot `cout` est un flot de sortie prédéfini connecté à la sortie standard `stdout`. De même, le flot `cin` est un flot d'entrée prédéfini connecté à l'entrée standard `stdin`.

On surchargera les opérateurs de flux `<<` et `>>` pour une classe quelconque, sous forme de **fonctions amies**, en utilisant ces « canevas » :

```
ostream & operator << (ostream & sortie, const type_classe & objet1)
{
   // Envoi sur le flot sortie des membres de objet en utilisant
   // les possibilités classiques de << pour les types de base
   // c'est-à-dire des instructions de la forme :
   // sortie << ..... ;

   return sortie ;
}

istream & operator >> (istream & entree, type_de_base & objet)
{
   // Lecture des informations correspondant aux différents membres de objet
   // en utilisant les possibilités classiques de >> pour les types de base
   // c'est-à-dire des instructions de la forme :
   // entree >> ..... ;

   return entree ;
}
```

Question 3. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter la surcharge de l'opérateurs de flux de sortie `<<` pour qu'il affiche un objet `Point` de la manière suivante : `<x,y>`. Décommenter les parties de code de cette question dans le fichier `testPoint.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
Point p0, p1(4, 0.0), p2(2.5, 2.5);

cout << "P0 = " << p0 << endl;
cout << "P1 = " << p1 << endl;
cout << "P2 = " << p2 << endl;
```

Ce qui donnera :

```
P0 = <0,0>
P1 = <4,0>
P2 = <2.5,2.5>
```

Question 4. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter la surcharge de l'opérateurs de flux d'entrée `>>` pour qu'il réalise la saisie d'un objet `Point` de la manière suivante : `<x,y>`. Décommenter les parties de code de cette question dans le fichier `testPoint.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
cout << "Entrez un point : ";
cin >> p0;

if (! cin)
{
   cout << "ERREUR de lecture !\n";
   return -1;
}

cout << "P0 = " << p0 << endl;
```


Ce qui donnera :

```
Entrez un point : <2,6>
P0 = <2,6>

Entrez un point : <s,k>
ERREUR de lecture !
```

La surcharge des opérateurs arithmétiques

Question 5. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter la surcharge des opérateurs utilisés dans le code source fourni. Décommenter les parties de code de cette question dans le fichier `testPoint.cpp` et tester. Vérifier que vous obtenez les résultats ci-dessous.

```
cout << "P0 * 2 = " << p0 * 2 << endl;
cout << "2 * P0 = " << 2 * p0 << endl;
cout << "-P0 = " << -p0 << endl;
cout << "P0 + P1 = ";
cout << (p0 + p1) << endl;
cout << "P0 * P1 = ";
cout << (p0 * p1) << endl;
```

Ce qui doit donner avec `P0 = <1,1>` :

```
P0 * 2 = <2,2>
2 * P0 = <2,2>
-P0 = <-1,-1>
P0 + P1 = <5,1>
P0 * P1 = 4
```

