TP POO C++ n°0

Sommaire

Les TP d'acquisition des fondamentaux visent à construire un socle de connaissances de base, à appréhender un concept, des notions et des modèles qui sont fondamentaux. Ce sont des étapes indispensables pour aborder d'autres apprentissages. Les TP sont conduits de manière fortement guidée pour vous placer le plus souvent dans une situation de découverte et d'apprentissage.

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

Initiation à la programmation orientée objet

Notion d'objets

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**.

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.

Notion de classe

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

Une classe représentera donc une **catégorie d'objets**.

Elle apparaît comme un **type** ou un *moule* à partir duquel il sera possible de créer des objets.

Notion de visibilité

Le C++ permet de préciser le **type d'accès des membres** (**attributs** et **méthodes**) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.

Notion d'encapsulation

L'encapsulation est l'idée de **protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler**.

En respectant ce principe, toutes les variables (attributs) d'une classe seront donc privées.

L'objet est ainsi vu de l'extérieur comme une "boîte noire" possédant certaines propriétés et ayant un comportement spécifié.

Exemple détaillé : une classe Point

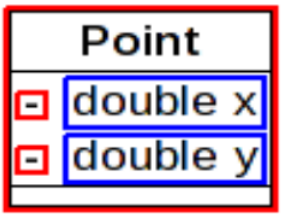
Nous allons successivement découvrir les notions suivantes :
- la modélisation et la déclaration d'une classe
- la construction d'objets d'une classe et la définition de ses fonctions membres
- les services rendus par une classe
- l'accès contrôlé aux membres d'une classe

Modélisation d'une classe

On veut manipuler des **points**. Un **point** est défini par **son abscisse** (`x`) et **son ordonnée** (`y`). L'abscisse et l'ordonnée d'un point sont des **réels** (`double`).

On en sait suffisamment pour modéliser une **classe `Point`** :

UML	C++

	`class Point`
	`{`
	`  private:`
	`  double x;`
	`  double y;`
	`};`

Le code C++ ci-dessus correspond à la déclaration de la classe `Point`. Elle se placera donc dans un fichier en-tête (*header*) `Point.h`.

Construction d'objets

Pour **créer des objets à partir de cette classe**, il faudra ... **un constructeur** :
- Un constructeur est chargé d'**initialiser un objet de la classe**.
- Il est appelé **automatiquement au moment de la création** de l'objet.
- Un constructeur est une **méthode qui porte toujours le même nom que la classe**.
- Il existe quelques contraintes :
  - Il peut avoir des paramètres, et des valeurs par défaut.
  - Il peut y avoir plusieurs constructeurs pour une même classe.
  - Il n'a jamais de type de retour.

On le **déclare** de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point

   public:
      Point(double x, double y); // je suis le constructeur de la classe Point
};
```

Point.h

Il faut maintenant **définir** ce constructeur afin qu'il **initialise tous les attributs de l'objet au moment de sa création** :

```
// Je suis le constructeur de la classe Point
Point::Point(double x, double y)
{
   // je dois initialiser TOUS les attributs de la classe
   this->x = x; // on affecte l'argument x à l'attribut x
   this->y = y; // on affecte l'argument y à l'attribut y
}
```

Point.cpp

Le code C++ ci-dessus correspond à la définition du constructeur la classe `Point`. Elle se placera donc dans un fichier C++ `Point.cpp`. On doit faire précéder chaque méthode de `Point::` pour préciser au compilateur que ce sont des membres de la classe `Point`. Le mot clé "`this`" permet de désigner l'adresse de l'objet sur laquelle la fonction membre a été appelée.

On pourra alors **créer nos propres points** :

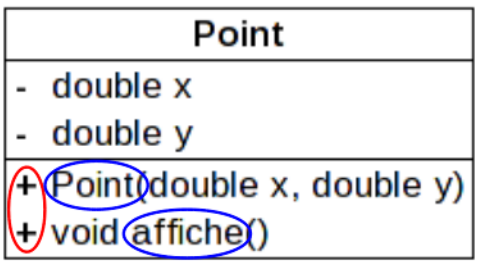
UML	C++
p1 Point x = 1 y = 2	`Point p1(1, 2);`
p2:Point x = 4 y = 0.	`Point p2(4, 0.);`

Les objets `p1` et `p1` sont des instances de la classe `Point`. Un objet possède sa propre existence et un état qui lui est spécifique (c.-à-d. les valeurs de ses attributs).

Les services rendus par une classe

Un point pourra s'**afficher**. On aura donc une **méthode** `affiche()` qui produira un affichage de ce type : `<x,y>`

UML	C++

	`class Point`
	`{`
	`  private:`
	`  double x, y;`
	
	`  public:`
	`  Point(double x, double y);`
	
	`  void affiche();`
	`};`
	
	`// Je suis le constructeur de la classe Point`
	`Point::Point(double x, double y)`
	`{`
	`  this->x = x;`
	`  this->y = y;`
	`}`
	
	`// Je suis la méthode publique affiche()`
	`void Point::affiche()`
	`{`
	`  cout << "<" << x << "," << y << ">" << endl;`
	`}`

A l'intérieur de la méthode `affiche()` de la classe `Point`, on peut accéder directement à l'abscisse du point en utilisant la donnée membre `x`. De la même manière, on pourra accéder directement à l'ordonnée du point en utilisant la donnée membre `y`.

On utilisera cette méthode dès que l'on voudra **afficher** les coordonnées d'un point :

```
cout << "P1 = ";
p1.affiche();

cout << "P2 = ";
p2.affiche();
```

Ce qui donnera à l'exécution :

```
P1 = <1,2>
P2 = <4,0>
```

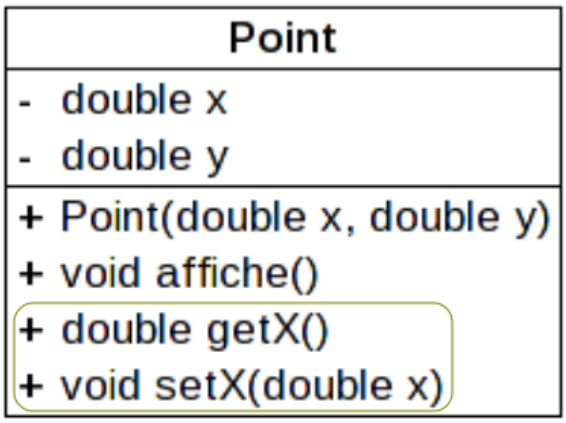
Une méthode publique est un service rendu à l'utilisateur de l'objet.

Les accès contrôlés aux membres d'une classe

Toutes les variables de la classe `Point` étant **privées par respect du principe d'encapsulation**, on veut néanmoins pouvoir connaître son abscisse et pouvoir modifier cette dernière.

Il faut donc créer deux **méthodes publiques** pour **accéder** à l'**attribut** `x` :

UML



C++

```cpp
class Point
{
    private:
        double x, y;

    public:
        Point(double x, double y);
        void affiche();
        double getX() const;
        void setX(double x);

};

// L'accesseur get du membre x
double Point::getX() const
{
    return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
    this->x = x;
}
```

La méthode `getX()` est déclarée constante (`const`). Une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet. Il est conseillé de qualifier `const` toute fonction qui peut l'être car cela garantit qu'on ne pourra appliquer des méthodes constantes que sur un objet constant.

La méthode publique `getX()` est un accesseur (*get*) et `setX()` est un manipulateur (*set*) de l'attribut `x`. L'utilisateur de cet objet ne pourra pas lire ou modifier directement une propriété sans passer par un accesseur ou un manipulateur.

On utilisera ces méthodes pour accéder en lecture ou écriture aux membres privés d'un point :

```
// on peut modifier le membre x de P1
p1.setX(5);
// on peut accéder au membre x de P1
cout << "L'abscisse de P1 est " << p1.getX() << endl;

// on peut accéder au membre x de P2
cout << "L'abscisse de P2 est " << p2.getX() << endl;
```

Ce qui donnera à l'exécution :

```
L'abscisse de P1 est 5
L'abscisse de P2 est 4
```

Travail demandé

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

```
#include <iostream>
#include <iomanip>

using namespace std;

#include "Point.h"

int main()
{
   Point p0(1,2);
   Point p1(4, 0.);

   cout.precision(2);

   cout << "p0 = ";
   p0.affiche();
   cout << "p1 = ";
   p1.affiche();

   p0.setX(5);
   cout << "L'abscisse de p0 est " << p0.getX() << endl;
   cout << "L'abscisse de p1 est " << p1.getX() << endl;
   cout << endl;

   /* Question 1 */
   /*
   p1.setY(6);
   cout << "L'ordonnée de p0 est " << p0.getY() << endl;
   cout << "L'ordonnée de p1 est " << p1.getY() << endl;
   cout << endl;
   */

   /* Question 2 */
```



```
/*
Point p2;
cout << "p2 = ";
p2.affiche();
cout << endl;
*/

/* Question 3 */
/*
Point *p3; // je suis pointeur sur un objet de type Point
p3 = new Point(2,2); // j'alloue dynamiquement un objet de type Point
cout << "p3 = ";
p3->affiche(); // Comme pointC est une adresse, je dois utiliser l'opérateur -> pour
    accéder aux membres de cet objet
//p3->setY(0); // je modifie la valeur de l'attribut y de p3
cout << "p3 = ";
(*p3).affiche(); // cette écriture est possible : je pointe l'objet puis j'appelle sa
    méthode affiche()
delete p3; // ne pas oublier de libérer la mémoire allouée pour cet objet
cout << endl;
*/

/* Question 4 */
/*
Point tableauDe10Points[10]; // typiquement : les cases d'un tableau de Point
int i;

cout << "Un tableau de 10 Point : " << endl;
for(i = 0; i < 10; i++)
{
   cout << "P" << i << " = "; tableauDe10Points[i].affiche();
}
cout << endl;
*/

/* Question 5 */
/*
const Point p5(7, 8);

cout << "p5 = ";
p5.affiche();
cout << "L'abscisse de p5 est " << p5.getX() << endl;
cout << "L'ordonnée de p5 est " << p5.getY() << endl;
//p5.setX(5); // essayez quand même !
*/

/* Question 6 */
/*
double distance = p5.distance(p2);

cout << "p5 = ";
p5.affiche();
cout << "p2 = ";
```



```
   p2.affiche();
   cout << "La distance entre p5 et p2 est de " << distance << endl;

   Point pointMilieu = p5.milieu(p2);
   cout << "Le point milieu entre p5 et p2 est "; pointMilieu.affiche();
   */

   return 0;
}
```

testPoint.cpp

À la fin du TP, vous devez obtenir l'exécution suivante :

```
$ ./testPoint
p0 = <1,2>
p1 = <4,0>
L'abscisse de p0 est 5
L'abscisse de p1 est 4

L'ordonnée de p0 est 2
L'ordonnée de p1 est 6

p2 = <0,0>

p3 = <2,2>
p3 = <2,2>

Un tableau de 10 Point :
P0 = <0,0>
P1 = <0,0>
P2 = <0,0>
P3 = <0,0>
P4 = <0,0>
P5 = <0,0>
P6 = <0,0>
P7 = <0,0>
P8 = <0,0>
P9 = <0,0>

p5 = <7,8>
L'abscisse de p5 est 7
L'ordonnée de p5 est 8

p5 = <7,8>
p2 = <0,0>
La distance entre p5 et p2 est de 11
Le point milieu entre p5 et p2 est <3.5,4>
```


Accesseurs de l'ordonnée d'un point

Question 1. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter les accesseurs `getY()` et `setY()` de l'ordonnée d'un point.

Constructeur par défaut

Si vous essayer de créer un objet sans lui fournir une abscisse `x` et une ordonnée `y`, vous obtiendrez le message d'erreur suivant :

```
ligne x: erreur: no matching function for call to Point::Point()'
```

Ce type de constructeur se nomme un **constructeur par défaut**. Son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable.

Le constructeur par défaut de la classe `Point` sera :

```
Point::Point() // Sans aucun paramètre !
{
   x = 0;
   y = 0;
}
```

Question 2. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter le constructeur par défaut de la classe `Point`.

Allocation dynamique d'objet

Pour allouer dynamiquement un objet en C++, on utilisera l'opérateur `new`. Celui-ci renvoyant une adresse où est créé l'objet en question, il faudra un pointeur pour la conserver. Manipuler ce pointeur, reviendra à manipuler l'objet alloué dynamiquement.

Pour libérer la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

Question 3. Vérifier le fonctionnement de l'allocation dynamique de l'objet `Point p3` réalisée dans le programme `testPoint.cpp`.

Un tableau d'objets

Il est possible de conserver et de manipuler des objets `Point` dans un tableau.

Question 4. Tester le fonctionnement de la déclaration d'un tableau de 10 objets `Point` réalisée dans le programme `testPoint.cpp`. Quel constructeur est appelé pour la création des objets `Point` ? Combien de fois ?

Un objet Point constant

Les règles suivantes s'appliquent aux objets constants :
– On déclare un objet constant avec le modificateur `const`
– On ne peut appliquer que des méthodes constantes sur un objet constant
– Un objet passé en paramètre sous forme de référence constante est considéré comme constant

Question 5. Pourquoi obtenez-vous une erreur à la compilation lorsque vous appellez la méthode `affiche()` sur l'objet constant `p5` ? Proposez une correction de cette erreur dans la classe `Point`.

Rendre des services

On doit pouvoir calculer la **distance** entre 2 points et le **milieu** de 2 points.

Question 6. Compléter les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter les méthodes `distance()` et `milieu()`.

L'objet `Point` passé en **argument** des méthodes `distance()` et `milieu()` le sera :
– en **référence** ce qui assure de bonnes performances
– et la référence sera **constante**, ce qui garantit que seules des méthodes constantes (ne pouvant pas modifier les attributs) seront appelables sur l'objet passé en argument

