TP 1 POO C++: Définition des objets et des classes

IMT Atlantique – TAF Robin/Ascii

Programmation en langage C/C++

Objectifs :
— Modélisation et déclaration d'une classe
— Construction d'objets d'une classe et la définition de ses fonctions membres
— Accès aux membres d'une classe

Exercice1 – Une classe `Point`

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`).

La classe `Point` devra disposer de **trois constructeurs** :
- un constructeur par défaut qui initialisera x et y à zéro ;
- un constructeur qui recevra en paramètres l'abscisse et l'ordonnée du point à initialiser ;
- un constructeur de copie.

Tous les attributs de la classe `Point` seront **privés**. On veut néanmoins pouvoir connaître son abscisse et son ordonnée, et éventuellement les modifier. Il faudra donc créer les **accesseurs** et **mutateurs** correspondants. Nous souhaitons également disposer d'une méthode permettant d'**afficher** les coordonnées du point sous la forme $< x, y >$. On doit aussi pouvoir calculer la distance entre 2 points et le milieu de 2 points.

Téléchargez le fichier `classe-point.zip` à partir de Moodle et complétez progressivement les fichiers `Point.cpp` et `Point.h`. Un programme de test `testPoint.cpp` permettant de tester les différentes questions est également fourni. Vous devez dé-commenter progressivement les parties de code source correspondant aux questions posées.

Constructeurs et destructeur de classe

Q1 Complétez les fichiers `Point.cpp` et `Point.h` fournis afin d'implémenter le constructeur par défaut de la classe `Point`, qui affichera le message "Je suis le constructeur par défaut de la classe Point !" et l'adresse mémoire de l'objet en hexadécimal.

Testez le fonctionnement du constructeur dans le programme `testPoint.cpp`. Vous devez obtenir l'exécution suivante :

```
Q1 :
Je suis le constructeur par défaut de la classe Point !
Point : 0x7ffebbeaa980
```

Q2 Pour compter le nombre d'instances actives de la classe `Point`, il faut créer une variable statique compteur, initialisée à 0, incrémentée à la construction de chaque nouvelle instance et décrémentée lorsqu'une instance disparaît.

— Complétez les fichiers `Point.cpp` et `Point.h` afin de compter le nombre d'instances actives de la classe.

Q3 On souhaite surcharger le constructeur par défaut pour initialiser des objets `Point` en fonction de leur abscisse et leur ordonnée. Vous devez, pour cela, utiliser la **liste d'initialisation** [1] pour définir le constructeur de la classe `Point`.

— Complétez les fichiers `Point.cpp` et `Point.h` afin d'implémenter un constructeur qui recevra en paramètres l'abscisse et l'ordonnée d'un point et qui affichera le message "Je suis le constructeur de la classe Point !"

— Testez le fonctionnement du constructeur dans le programme `testPoint.cpp`. Vous devez obtenir l'exécution suivante :

```
Q3 :
Je suis le constructeur de la classe Point !
P1(2.0,3.0): 0x7ffdad92daf0
Je suis le constructeur de la classe Point !
P2(2.0,1.0): 0x7ffdad92db00
```

Q4 Le **constructeur de copie** est appelé dans :

- la création d'un objet à partir d'un autre objet pris comme modèle ;
- le passage en paramètre d'un objet par valeur à une fonction ou une méthode ;
- le retour d'une fonction ou une méthode renvoyant un objet.

La forme habituelle d'un constructeur de copie est la suivante :

```
Class T
{
  public :
     T(const  T&)
};
```

— Complétez les fichiers `Point.cpp` et `Point.h` afin d'implémenter le constructeur de copie, qui affichera le message "Je suis le constructeur de copie de la classe Point !".

— Testez le fonctionnement du constructeur dans le programme `testPoint.cpp`. Vous devez obtenir l'exécution suivante :

```
Q4 :
Je suis le constructeur de copie de la classe Point !
P3(2.0, 3.0) : 0x7ffcdd683fa0
```

Q5 Le destructeur est une méthode membre appelée automatiquement lorsqu'un objet est détruit.

— Complétez les fichiers `Point.cpp` et `Point.h` afin d'implémenter le destructeur de la classe `Point` qui affichera le message "Je suis le destructeur de la classe Point !" et qui décrémentera la valeur du compteur lorsqu'une instance disparaît.

— Testez le fonctionnement du destructeur dans le programme `testPoint.cpp`. Vous devez obtenir l'exécution suivante :

1. La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre.

```
Q5 : cpt = 3
Je suis le constructeur de la classe Point !
P4(1,1): 0x7fffe2749b50
cpt : 4
Je suis le constructeur de copie de la classe Point !
P5(1,1): 0x7fffe2749b60
cpt : 5
Je suis le destructeur de la classe Point !
Je suis le destructeur de la classe Point !
cpt: 3
```

Q6 Complétez les fichiers `Point.cpp` et Point.h fournis afin d'implémenter les accesseurs et mutateurs des attributs x et y. Testez leur fonctionnement dans programme `testPoint.cpp`. Vous devez obtenir l'exécution suivante :

```
Q6 :
L'abscisse de p0 est 0
L'abscisse de p1 est 2
L'ordonnée de p0 est 0
L'ordonnée de p1 est 3
L'abscisse de p0 est maintenant 1.5
L'ordonnée de p0 est maintenant 3.5
```

Allocation dynamique d'objet

On veut pouvoir créer dynamiquement, par `new`, des objets de cette classe. Celui-ci renvoyant une adresse où est créé l'objet en question, il faudra un pointeur pour la conserver.

Q7 Créez dynamiquement une nouvelle instance $p4$ de la classe `Point`. Affichez les coordonnées de $p4$ à l'aide des accesseurs et mutateurs. Vous devez obtenir l'exécution suivante :

```
Q7 : allocation dynamique
Je suis le constructeur de la classe Point !
P4: 0x55a2e07db280
L'abscisse de p4 est 1
L'ordonnée de p4 est 0
Je suis le destructeur de la classe Point !
```

Q8 On souhaite créer un tableau de 10 objets `Point`. Testez 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 ?

Manipulation d'objets constants

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 ;
— On déclare une méthode [2] constante en ajoutant le modificateur `const` ;
— Un objet passé en paramètre sous forme de référence constante est considéré comme constant.

2. Une méthode constante est une méthode qui ne modifie aucun des attributs de l'objet.

Q9 Créez un objet constant $p4$ dans le programme `testPoint.cpp`. Appelez ensuite les méthodes `get()` et `set()` sur l'objet constant $p4$. Que constatez-vous ? Proposez une solution pour corriger cette erreur dans la classe `Point`.

Implémentation des autres méthodes de la Classe `Point`

Q10 Complétez les fichiers `Point.cpp` et `Point.h` afin d'implémenter

(a) la méthode `afficher()`

(b) la méthode `distance()`

(c) la méthode `milieu()`

```
Q10 : test services de la classe Point
p0 = <1.5,3.5>
p1 = <2,3>
p2 = <2,1>
calcul de la distance entre 2 points
La distance entre p1 et p2 est de 2
Le point milieu entre p1 et p2 est <2,2>
```

Exercice 2 – Classe `Stack` (A faire en autonomie)

Une **pile (« stack » en anglais)** est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (Last In, First Out)**. Toutes les opérations définies sur les piles s'appliquent à une seule extrémité, appelée **sommet**. En particulier l'ajout et la suppression d'éléments en sommet de pile suivent le modèle LIFO.

La notion de pile est une notion importante, et tout particulièrement en informatique (ex : la pile d'évaluation d'un programme qui reflété les différents appels de fonctions).

Cinq opérations de base sont définies sur le type `Stack` :

— *empiler* (ou **push**) : ajoute un élément en sommet de pile ;

— *déplier* (ou **pop**) : supprime et retourne l'élément en sommet de pile ;

— *sommet* (ou **peek**) : retourne l'élément en sommet de pile sans le dépiler ;

— *est-vide ?* (ou **isEmpty**) : renvoie « vrai » si la pile est vide, sinon « faux » ;

— *taille* (ou **get_size**) : renvoie le nombre d'éléments présents dans la pile.

Réalisation d'une pile au moyen d'un tableau

Il existe classiquement deux façons de représenter une pile, soit par un tableau, soit par une liste chaînée. Dans cet exercice nous utiliserons la première représentation. Le but est de coder une classe `stackArray` réalisant la pile au moyen d'un tableau de taille fixe. Les données membres **privées** associées à cette classe seront :

- une taille maximale `tailleMax` pour le tableau stockant les éléments de la pile. La taille sera fixée à l'initialisation donc `tailleMax` sera déclarée comme donnée membre **constante**.
- un entier `head` donnant l'indice du tableau correspondant au sommet de pile (initialisé à -1)
- un pointeur `data` désignant un tableau d'entiers, alloué dynamiquement avec (**new**) pour mémoriser le contenu de la pile (initialisé à 0).

Vous associerez à votre classe un **constructeur** prenant en paramètre la taille maximale associée à la pile et permettant d'initialiser toutes les données membres de la classe, ainsi qu'un **destructeur** pour libérer la mémoire allouée pour le tableau d'entiers `data`. Puis vous la doterez de l'ensemble des méthodes définies précédemment.

```
Class stackArray
{
  private :
    static const int default_size = 50; // constante de classe
    const int tailleMax;
          int head;
          int* data;
  public :
    // constructeur par défaut
    stackArray (int size = default_size) : // A compléter
};
```

Par ailleurs, pour traiter proprement les situations de fonctionnement anormal de la pile (dépliement d'une pile vide, débordement, etc), il faudrait utiliser les **exceptions** (cf. section Gestion des exceptions). Pour l'instant, contentez-vous de signaler ces situations par un message d'erreur (utilisez le flux de sortie prédéfini `cerr` pour la gestion des erreurs et arrêtez l'exécution du programme au moyen de l'instruction `exit (1)`.

Dans un programme `testPile.cpp`, créer une pile et testez toutes les méthodes de classe `stackArray`.

Gestion des exceptions

En partant de la classe `stackArray`, on veut que les méthodes *dépiler* et *sommet* retournent une erreur si la pile est vide et que la méthode *empiler* retourne une erreur si la pile est pleine.

Implémentez, en utilisant les exceptions, les mécanismes permettant de récupérer les erreurs de type `out_of_range` de la classe `exception`.