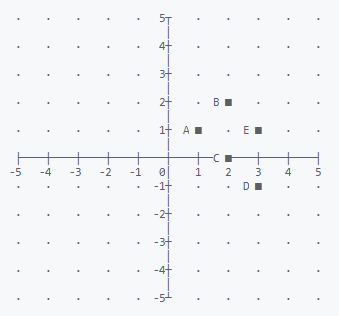
|  |  |  |  |
| --- | --- | --- | --- |
| **BTS2 Devoir #2** | **POO et C++ (4h)** | | **17/12/2020** |
| **Nom :** | | **Prénom :**  /20 | |

# Présentation du système

Ce devoir est divisé en deux parties : la première sur papier et la seconde sur ordinateur.

Le programme mis en place dans le cadre de ce devoir doit permettre la manipulation de points, segments et polygones dans un espace en deux dimensions, la détection de forme particulière et le calcul de propriétés simples : longueurs, périmètres, aires…

# Point



Les points sont représentés par des objets de type « Point ». Ils possèdent trois propriétés :

* « name » : une chaîne de caractères correspondant au nom du point
* « x » : la coordonnée entière du point sur l’axe des abscisses
* « y » : la coordonnée entière du point sur l’axe des ordonnées

Une propriété « count » commune à tous les points permet de connaître le nombre de point instanciés dans le système.

On ne peut créer des objets « Point » qu’en précisant leur nom et leurs coordonnées (pas de valeur par défaut).

Point a("A", 1, 1);

Les propriétés « nom », « x » et « y » ne sont pas modifiables après la création.

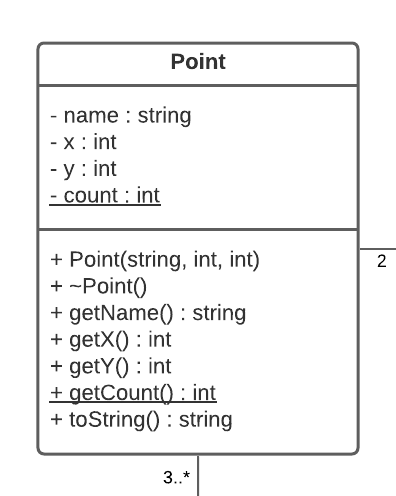
Une méthode « toString() » accessible à l’extérieur de l’objet permet de récupérer une chaîne de caractère donnant les valeurs des propriétés de l’objet avec le formatage suivant :

cout << a.toString();

> A (1, 1)

**Q1 (3pts).** En respectant l’encapsulation et en prévoyant les accesseurs nécessaires, proposer un diagramme de classe UML pour la classe Point.

*(Nb. Les attributs et méthodes statiques sont représentés avec un soulignement.)*



**Q2 (2pts)**. Proposer la déclaration en C++ de la classe Point correspondant à votre UML.

*(Nb. Concernant les méthodes, la déclaration se limite à leur prototype.*

class Point {

    private :

        string name = "";

        int x, y;

        static int count;

    public :

        Point(string, int, int);

        ~Point();

        string getName();

        int getX();

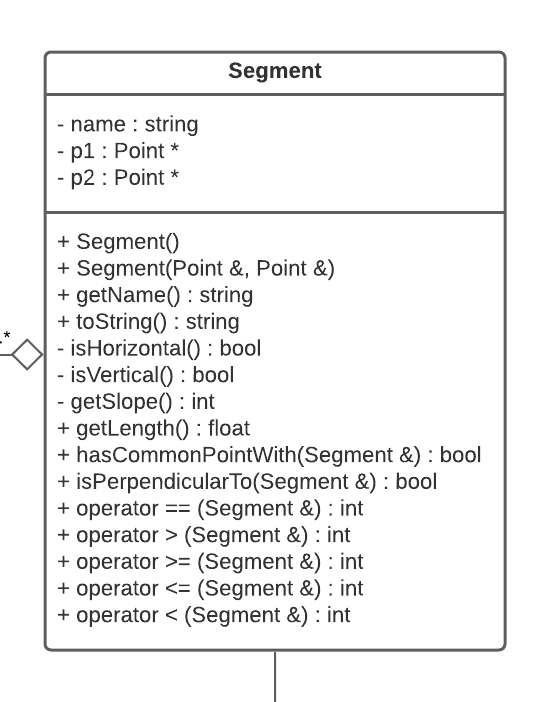
        int getY();

        static int getCount();

        string toString();

};

# Segment

Les segments relient deux points par des lignes droites. Ils sont représentés par des objets « Segment » qui ont les propriétés suivantes :

* « name » : le nom du segment créé à partir du nom des deux points le constituant (ex : un segment reliant un point « A » à un point « B » sera nommé « AB »)
* « p1 » : un pointeur vers le premier point relié par le segment
* « p2 » : un pointeur vers le second point relié par le segment

Le diagramme UML de la classe Segment est présenté ci-contre.

**Q3 (1pt)**. Le second constructeur de la classe a le prototype suivant :

Segment(Point &, Point &)

A quoi sert le symbole « & » ?

Le & permet de passer les arguments par référence. Cela veut dire qu’il n’y a pas de recopie des objets passés en arguments dans la fonction et que ce sont les objets du programme appelant qui sont passés et donc modifiables depuis la fonction.

**Q4 (1pt)**. En se basant sur le type de p1 et p2, qu’adviendra-t-il de l’attribut « count » des points à la création d’un segment ?

Comme il n’y a pas de recopie des objets Point, l’attribut count ne sera pas incrémenté.

**Q5 (1pt)**. Ce constructeur valorise p1 et p2 à partir de ses arguments et génère le nom du segment. Proposer une définition de ce constructeur en C++.

Segment::Segment(Point & \_p1, Point & \_p2) {

    name = \_p1.getName() + \_p2.getName();

    p1 = &\_p1;

    p2 = &\_p2;

}

**Q6 (1pt)**. La méthode « isHorizontal() » renvoie « vrai » quand le segment répond à la définition suivante : un segment est horizontal quand les abscisses des points le constituant sont égales. Proposer une définition de cette méthode en C++.

bool Segment::isHorizontal() {

    return p1->getY() == p2->getY();

}

**Q7 (1pt)**. Proposer une définition de la méthode « isVertical() » en C++.

bool Segment::isVertical() {

    return p1->getX() == p2->getX();

}

Les méthodes suivantes seront étudiées dans la seconde partie du devoir. Voici leur description :

* « getName() » : accesseur de « name ».
* « toString() » : renvoie une chaîne de caractère contenant la concaténation entre accolade du résultat de la méthode « toString() » de p1 et p2.
* « getSlope() » : renvoie la pente du segment.
* « getLength() » : renvoie la longueur du segment.
* « hasCommonPointWith() » renvoie « vrai » si le segment a un point commun avec le segment passé en argument.
* « isPerpendicularTo() » renvoie « vrai » si le segment est perpendiculaire au segment passé en argument.

**Q8 (1pt)**. Expliquer ce que sont et à quoi servent les méthodes commençant par « operator ».

Ce sont des surdéfinitions d’opérateurs. Elles permettent de définir le comportement des opérateurs avec la classe Segment.

**Q9 (1pt)**. Un segment est égal à un autre s’ils relient les mêmes points. Proposer une définition de la méthode correspondant à la ligne « operator == (Segment &) : int » en C++.

int Segment::operator == (Segment & \_s) {

    return p1 == \_s.p1 && p2 == \_s.p2;

}

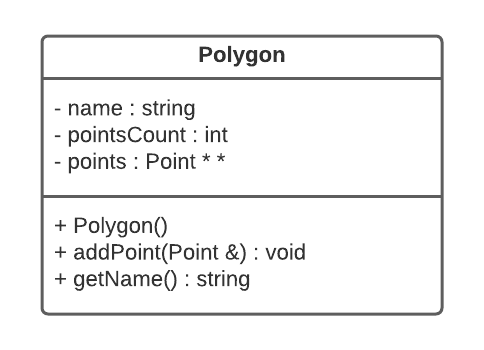
**Q10 (1pt)**. Un segment est plus grand qu’un autre si sa longueur est strictement plus grande que celle de l’autre. Proposer une définition de la méthode correspondant à la ligne « operator > (Segment &) : int » en C++.

int Segment::operator > (Segment & \_s) {

    return getLength() > \_s.getLength();

}

# Polygone

Un polygone relie au moins 3 points par des segments. Ils sont représentés par des objets Polygon qui ont les propriétés suivantes :

* « name » : le nom du polygone créé à partir du nom des points le constituant dans l’ordre de déclaration (ex : un polygone reliant les points « A », « B », « E » et « C » sera nommé « ABEC »).
* « pointsCount » : le nombre de points constituant le polygone.
* « points » : un tableau de pointeurs sur les points constituant le polygone.

Le diagramme UML de la classe Polygon est présenté ci-contre.

**Q11 (1pt)**. En se basant sur le type des attributs, quelle méthode supplémentaire serait-il souhaitable de mettre en place pour une gestion optimale de la mémoire ? Expliquer pourquoi.

Il faut mettre en place un destructeur pour libérer la mémoire qui va être allouée dynamiquement au tableau de pointeurs « points ».

La méthode « addPoint() » permet d’ajouter un point au polygone. Elle incrémente « pointsCount », augmente la taille de « points » et y ajoute l’adresse de l’objet Point fourni en argument.

**Q12 (2pts)**. Le fragment de programme suivant entraînera une anomalie, expliquer laquelle, proposer un mécanisme pour l’éviter et expliquer son fonctionnement.

...

Point a("A", 1, 1), b("B", 2, 2), c("C", 2, 0);

Polygon \* triangle1;

\*(triangle1).addPoint(a);

\*(triangle1).addPoint(b);

\*(triangle1).addPoint(c);

Polygon triangle2 = \*triangle1;

delete triangle1;

...

*(Nb. On pourra utiliser un schéma.)*

L’instruction Polygon triangle2 = \*triangle1; fait appel au constructeur par recopie, or il n’est pas défini dans la classe « Polygon », c’est donc le constructeur par recopie standard qui va être appelé. Le constructeur par recopie standard ne fait que recopier les valeurs des attributs de l’objet, même pour les pointeurs. L’attribut « points » de « triangle2 » pointe donc vers la même adresse (le même espace mémoire) que celui de « \*triangle1 ». L’instruction delete triangle1; détruisant l’objet pointé par « triangle 1 », l’espace mémoire pointé par « triangle1->points » est réalloué et « triangle2->points » pointent donc dans une zone mémoire qui n’est plus contrôlée.

Il faut donc mettre en place un constructeur par recopie dans la classe Polygon pour allouer correctement l’espace pour l’attribut points.

**Q13 (1pt)**. Le fragment de programme suivant entraînera la même anomalie, proposer un mécanisme pour l’éviter et expliquer son fonctionnement.

...

Point a("A", 1, 1), b("B", 2, 2), c("C", 2, 0);

Polygon \* triangle1, triangle2;

\*(triangle1).addPoint(a);

\*(triangle1).addPoint(b);

\*(triangle1).addPoint(c);

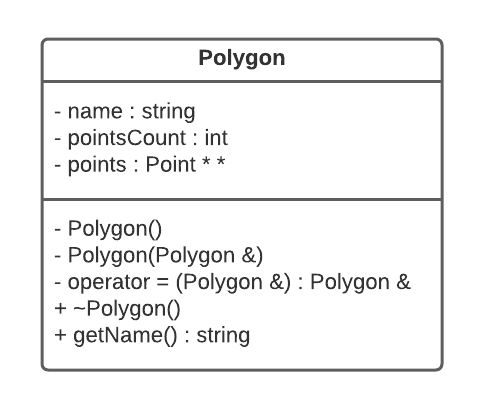
triangle2 = \*triangle1;

delete triangle1;

...

Il faut surdéfinir l’opérateur d’affectation (=) et allouer correctement l’espace pour l’attribut points.

On souhaite finalement qu’il ne soit pas possible d’instancier directement des objets « Polygon ». Les constructeurs de la classe sont donc déclarés en privé et la méthode « addPoint() » est supprimée.

Le diagramme UML de la classe « Polygon » devient le diagramme ci-contre.

L’instanciation d’objet « Polygon » ne doit plus être possible que par l’instanciation d’objet héritant de cette classe.

Les classes « Triangle » et « Quadrilateral » sont créées dans ce but.

La classe « Triangle » permet de créer des polygones à partir de 3 points passés en arguments de son constructeur.

La classe « Quadrilateral » permet de créer des polygones à partir de 4 points passés en arguments de son constructeur.

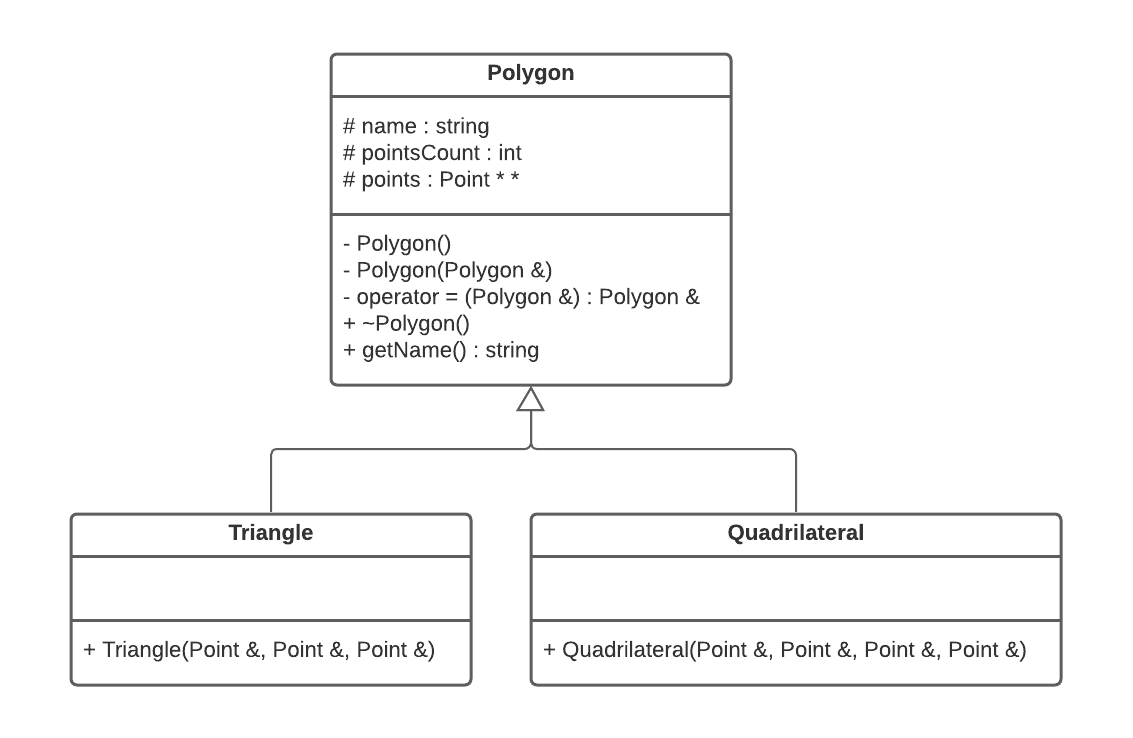
**Q14 (1pt)**. Ces deux classes doivent avoir accès aux attributs de la classe « Polygon », quelles modifications doivent être apportées à cette dernière pour que cela soit possible ?

Il y a deux solutions :

1. soit passer les attributs en accès protected
2. soit créer des accesseurs public ou protected pour les attributs

**Q15 (2pt)**. Proposer le diagramme de classe UML correspondant à votre réponse à la question précédente et incluant les classes « Polygon », « Triangle » et « Quadrilateral ».

Version si on passe les attributs de Polygon en protected (1) :



Version si on crée des accesseurs public :

