Cours de C++, 1ère année, HE-Arc

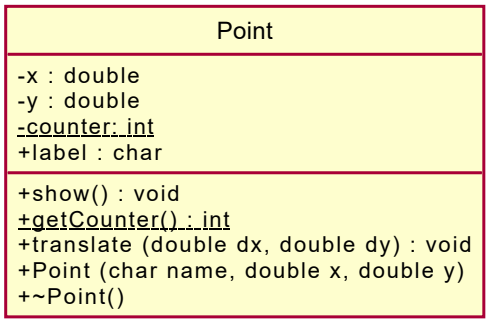
Serie 2.2 : Des classes plus classes...

Exercice 1 : CLASSE Point améliorée

Reprendre la classe `Point` de la série 2.1 et la compléter avec les éléments suivants :

- Une variable membre *statique* privée `counter` initialisée à 0 qui sera incrémentée dans tous les constructeurs et décrémentée dans le destructeur de la classe
- Une méthode *statique* publique `getCounter()` qui permettra au programme `main` d'afficher la valeur de `counter`
- Un destructeur `~Point()`

Son diagramme de classe UML sera donc :



- Compléter le programme pour créer des objets `Point` et afficher la valeur du compteur. Vérifier en particulier qu'à la fin du programme il n'y ait plus d'objets `Point` en mémoire grâce au compteur (encapsuler `main` dans une paire de { } supplémentaire). Par exemple :

```
int main()
{
    Point *ptrPoint = nullptr;
    {
        Point p1('A');
        Point p2('B', 3, 4);

        // 1. Afficher le nombre de points avec getCounter() (doit afficher 2)

        // 2. Créer dynamiquement un point et mettre son adresse dans le pointeur

        // 3. Afficher le nombre de points (doit afficher 3)

    }   // fin du bloc --> p1 et p2 sont détruits ici

        // 4. Afficher le nombre de points (doit afficher 1)

    delete ptrPoint;    // destruction du point alloué dynamiquement
    ptrPoint = nullptr;
```

```
        // 5. Afficher le nombre de points (doit afficher 0)

    cout << "\n\nPlease hit ENTER to continue... ";
    cin.get();
    return 0;
}
```

Exercice 2 : CLASSE Rectangle (composition de points)

Une ***composition*** est une relation très forte. Quand l´objet est détruit, les éléments qui le composent doivent aussi être détruits (si je détruis ma maison, les pièces qui la composaient sont détruites).

En s'appuyant sur la classe `Point`, réaliser la classe `Rectangle`.

Un *rectangle* sera composé de 2 *points* : son coin supérieur gauche `cornerUL` et son coin inférieur droit `cornerBR`.

La classe possèdera les méthodes :

- `contains(const Point& p)` renvoie `true` si le point `p` est à l'intérieur du rectangle
- `getPerimeter()` *retourne le périmètre du rectangle*
- `show()` *affiche les informations du rectangle (coordonnées des coins et périmètre)*
- `translate(int, int)` *translate les deux points du rectangle*

ainsi que 2 constructeurs :

- `Rectangle(int xUL, int yUL, int xBR, int yBR)`
- `Rectangle(const Point& cornerUL, const Point& cornerBR)`

A faire :

- **Donner le diagramme UML de la classe `Rectangle`**
- **Implémenter la classe `Rectangle`** en s'appuyant sur la classe `Point`
  - Ne pas oublier le mot clé `const` pour les méthodes constantes (ex. `void show() const`).
- **Ecrire un programme** pour tester la classe `Rectangle`. Le programme construira un objet rectangle avec chacun des constructeurs, les affichera avec `show()`, utilisera ses méthodes, puis modifiera les arguments de constructions donnés en paramètre aux constructeurs et les affichera à nouveau.

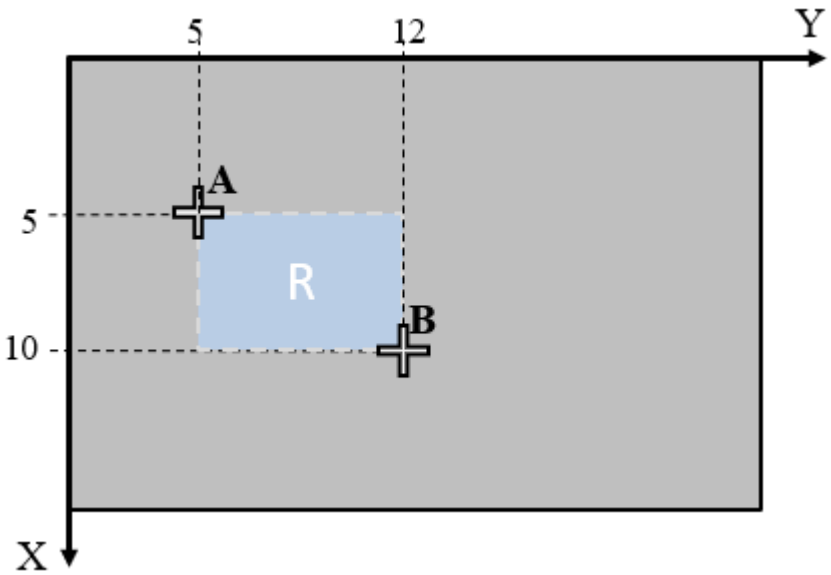
Question : comment donner accès aux coordonnées des points aux fonctions de `Rectangle` ?

Question : est-il possible de contruire un rectangle à partir d'un autre ? Essayer et afficher le nouveau rectangle pour voir si cela fonctionne bien.

Exemple d´extrait de programme :

```
  Point pt1('A', 5., 5.), pt2('B', 10., 12.), ptX('X', 7., 8.);
  Rectangle R(pt1, pt2);
  R.show();
  cout << "Périmètre : " << R.getPerimeter() << endl; // 24
  cout << "is ptX contained in the area of rectangle R ? " <<  R.Contains(ptX) <<
endl;
  pt1.translate(3,5);
  R.show();
  cout << "Périmètre : " << R.getPerimeter() << endl; // ??
```

```
    // construire un rectangle R2 à partir de R et l'afficher
```



Exercice 3: CLASSE RectangleAssoc (association de points)

Une ***association*** est une relation moins forte. Quand l'objet est détruit, les éléments qui y étaient associés continuent à exister (si une entreprise disparait, on ne liquide pas ses employés !)

En s'appuyant sur la classe `Point`, réaliser la classe `RectangleAssoc`.

Un *rectangle* sera associé à 2 points par les pointeurs : `ptrULCorner` et `ptrBRCorner`.

La classe possèdera les méthodes : `show()` , `getPerimeter()`, et `translate(int, int)`

La classe `RectangleAssoc` offrira un constructeur :

- `RectangleAssoc(Point* cornerUL, Point* cornerBR)`

A faire :

- **Implémenter la classe** `RectangleAssoc` en s'appuyant sur la classe `Point`
- **Ecrire un programme** qui
  - Crée deux points
  - Associe les deux points pour construire le rectangle `R` de classe `RectangleAssoc`
  - Utilise `translate(dx, dy)`sur un des points
  - Affiche le rectangle et les points avec `show()`; qu'observez-vous ?
  - Crée un second *rectangle* copie du rectangle original avec la déclaration `RectangleAssoc copyR(Ra1);`
  - `translate` un des deux rectangles puis affiche les deux rectangles; qu'observez-vous ?

Puis

- **supprimer la dépendance** (le lien observé entre les deux rectangles) : **implémenter le constructeur par copie** `RectangleAssoc(const RectangleAssoc&)`et le **destructeur** `~RectangleAssoc()`qui utilisent l'allocation dynamique pour faire une *copie en profondeur* des éléments associés au rectangle original (ses points). Enfin, exécuter à nouveau le programme et vérifier que le problème soit résolu (les rectangles peuvent être translatés *indépendamment*)

Exemple d´extrait de programme:

```
  Point pt1('A', 5, 5), pt2('B', 10, 12);

  // partie 1
  RectangleAssoc R( &pt1, &pt2);
  R.show();   // OK
  pt1.translate(3,5);
  R.show();   // ?!? grmph

// partie 2
  RectangleAssoc copyR(R);
  R.translate(-5,-5);
  R.show();       // OK
  copyR.show();   // ?!? grmph    --> :) après complétion de l'exercice
```