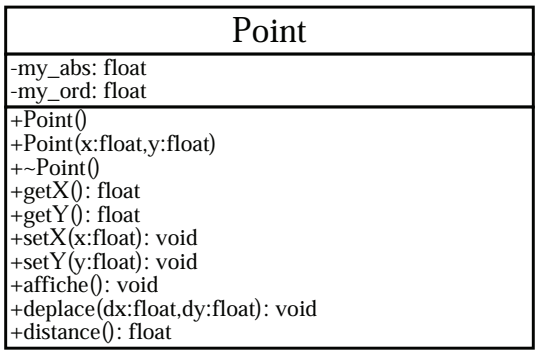
Semaine 1
Introduction aux classes - 1/2

Sources liées à ce TD :
- `classe_Point_a_distribuer.ps` : pour les étudiants
- `classe_Point_profs.ps` : compléments pour les enseignants

1 Rappels et compléments de cours

Exercice 1 : Écrivez une classe `Point`, respectant l'interface ci-dessous, regroupant ce qui a été vu en cours.



Vous écrirez le fichier d'entête (.h) ainsi que le fichier source (.cc).

Note pour les enseignants : guider les étudiants pour qu'ils écrivent la classe donnée en annexe (cf page web : `classe_Point_a_distribuer.ps`), sans les `const` qui seront commentés plus tard.

Écrivez ensuite un exemple de fonction `main` succinct pour observer comment fonctionnent les déclarations d'objets grâce aux constructeurs, ainsi que les appels de méthodes.

À la fin de l'exercice, distribuer la feuille `classe_Point_a_distribuer.ps` pour que les étudiants aient tous une version propre pour la suite (les méthodes constantes seront commentées dans la partie 3).

Pendant le déroulement de cet exercice, on pourra faire des rappels sur :
- **Classe :** ce qu'elle contient
  - données membres (ou attributs),
  - fonctions membres (ou méthodes), qui implémantent la structure de données et

l'ensemble des opérations (interface).
Rappeler la notion de private / public.

- **Objet :** représentant physique (ou instance) d'une classe, défini par un nom.
  - Etat de l'objet = valeurs des attributs à un instant donné.
  - Comportement de l'objet = ensemble des opérations applicables à l'objet.
  On pourra s'aider d'un schéma.
- **Utilisation constructeurs et destructeur :** fonctions membres spécifiques qui sont définies pour assurer l'initialisation et la terminaison (désallocation mémoire) des objets. Fonctions de même nom que la classe (précédé d'un tilde pour le destructeur), sans type de retour (void).
  Le constructeur est invoqué implictement à chaque déclaration d'un objet, le destructeur à chaque "fin d'utilisation" d'un objet (fin du bloc où il a été déclaré).
- **Utilisation des autres fonctions membres :** notation `obj.fonction()`,
- **Constructeur par défaut :** il est possible de définir plusieurs constructeurs à condition que leurs listes de paramètres soient différentes (en nombre et/ou en type). En particulier, un constructeur très utile est celui par défaut (sans paramètre), qui permet de définir un point de la manière suivante : `Point P;`. Commenter l'exemple.
  Ce constructeur est indispensable si on veut définir un tableau de points.
- **Tableaux d'Objets :** il est possible de définir des tableaux d'objets de la même manière que les tableaux de structures vus précédemment. On déclarerait un tableau de `Points` de la manière suivante : `Point T[5];`
  On constate que l'on ne peut pas utiliser un autre constructeur que celui par défaut. Si on veut utiliser ce tableau de Points comme paramètre d'une méthode, seul un passage par référence est possible. Ceci est directement le cas avec la notation suivante :
  ```
  void action(Point T[]){...}
  ```

Une fois la feuille `classe_Point_a_distribuer.ps` distribuée, on pourra faire des commentaires supplémentaires sur les constructeurs et le destructeur en s'appuyant sur le résultat d'exécution (grâce aux `cout` placés dans les constructeurs et le destructeur).

Exercice 2 : Écrivez une fonction **non membre** `voisins` qui prend deux paramètres (un point et un tableau de 4 points) et remplit le tableau avec les quatre voisins du point passé en paramètre.
Prototype : `void voisins( Point a, Point T[])`
Réponse :

```
void
voisins ( Point a, Point T[]) {
  T[0].setX(a.getX()-1);
  T[0].setY(a.getY());
  T[1].setX(a.getX()+1);
  T[1].setY(a.getY());
  T[2].setX(a.getX());
  T[2].setY(a.getY()-1);
  T[3].setX(a.getX());
  T[3].setY(a.getY()+1);
}
```

Exercice 3 : Écrivez ensuite une fonction **non membre** `milieu` qui prend deux points en paramètre et calcule le point qui est au milieu.
Réponse :

```
void
milieu( Point  a,  Point  b, Point & m ) {
  m.setX( (a.getX()+b.getX())/2 );
  m.setY( (a.getY()+b.getY())/2 );
}
```

2 Passage par référence constante

Jusqu'à présent, on vous a présenté trois types de passage de paramètres :

1. Le **passage par valeur**, ou passage en entrée (pas de "&"). A noter qu'une nouvelle variable est créée et une copie de l'argument est réalisé (voir après constructeur de copie).

   Avantages : on est sûr que l'utilisateur ne pourra pas modifier l'argument d'appel car il manipule toujours une *copie* de cet argument et non l'argument lui-même.

   Inconvénients : à chaque fois que l'on passe un paramètre par valeur, une nouvelle instance est créée et le constructeur par copie est appelé. Si l'objet passé par valeur est gros (par exemple une pile ou une liste), le passage par valeur est très coûteux.

2. Le **passage par référence**, ou passage en entrée/sortie (un "&"). Aucune variable n'est créée. Seulement un synonyme du paramètre est créé.

   Avantages : le passage par référence est très rapide (une simple "adresse" de variable est donnée). Les données de la variable sont directement accessibles et aucune copie n'a été effectuée.

   Inconvénients : la fonction peut modifier comme elle l'entend ce paramètre. Il est donc peu fiable d'utiliser ce passage de paramètre pour des paramètres en entrées.

3. Le **passage par adresse (ou pointeur)** est un type particulier de passage par valeur, où la valeur copiée de l'argument d'appel vers le paramètre est une adresse en mémoire. Il faut donc le voir comme un passage par valeur particulier.

On voudrait donc avoir un mécanisme rapide pour passer les paramètres en entrées. La solution : le **passage par référence constante**. Il s'agit juste de mettre le mot-clé `const` devant le paramètre par référence, et la fonction ne peut pas modifier ce paramètre.

Exercice 4 : Modifier la fonction (non membre) `milieu`.
Réponse : `void milieu( const Point & a, const Point & b, Point & m ) { ... idem }`

Exercice 5 : Écrivez une fonction **membre** `distance` qui calcule et retourne la distance entre le point lui-même et un autre point passé en paramètre.
Réponse :

```
float
Point::distance( const Point &p ) const     // remarque const après
```

```
{
  float carrex = (p.my_abs-my_abs) * (p.my_abs-my_abs);
  float carrey = (p.my_ord-my_ord) * (p.my_ord-my_ord);
  return sqrt(carrex + carrey);
}
```

Remarque : On peut faire ici une remarque sur la surcharge. De la même façon qu'un constructeur peut être surchargé (ses paramètres diffèrent en nombre et/ou en type), nous pouvons surcharger une fonction membre comme `distance`. On peut par exemple faire cohabiter dans la même classe les deux fonctions suivantes :

```
float Point::distance(const Point &p) const; // distance entre l'instance et p
float Point::distance( ) const; // distance entre l'instance et l'origine
```

3 Méthodes constantes

Regardons précisément les prototypes des méthodes de la classe `Point`. On constate que certaines méthodes se terminent par le mot-clé `const`, d'autres non. Un examen plus précis montre que **les méthodes qui ne modifient pas leurs données membres ont le mot-clé `const`** tandis que **les méthodes qui modifient leurs données membres n'ont pas le mot-clé `const`**. Une méthode dont le prototype est terminé par `const` est appelée *méthode constante*. Dans le corps d'une méthode constante, il est impossible de modifier les données membres de l'objet, de même qu'il est impossible d'appeler des méthodes non-constantes de cet objet. En gros, un objet est en accès lecture seulement dans ses méthodes constantes.

Intérêt. On peut appeler des méthodes constantes attachées à des références constantes. Ainsi, dans la fonction `milieu` précédente, on peut appeler les méthodes constantes `getX` et `getY`.

Par contre, il est impossible d'appeler une méthode non-constante sur une référence constante. Le compilateur refuse de compiler. Dans la fonction `milieu` précédente, on ne peut écrire `a.setX(5)`.

Retenez que, sur les objets passés par référence constante, il est possible : (i) d'accéder en lecture aux attributs (publics ou même privés si c'est un objet de la même classe), (ii) d'appeler les méthodes constantes de cet objet.

Ainsi, il est intéressant de passer des paramètres sous forme de référence constante pour des raisons d'efficacité. En revanche, l'utilisation de références constantes nous impose une plus grande discipline dans la définition des méthodes : nous devons différencier dans nos méthodes celles qui sont constantes des autres.

4 Constructeur de copie

Nous avons vu précédemment que le passage par valeur d'un objet de la classe `Point` provoquait une recopie de l'objet dans une instance locale de l'objet. Or, dans notre classe,

nous n'avons pas défini comment doit se faire cette recopie. Par défaut, elle se fait membre à membre, ici, cela nous convient bien. Cependant, très bientôt, nous serons amenés à faire de l'allocation dynamique de mémoire, et dans ces cas-là, ce fonctionnement peut s'avérer dramatique (partage d'un même espace mémoire par deux instances différentes).
Ainsi, pour programmer correctement, il faut définir un **constructeur de copie**.

Exercice 6 : Pouvez-vous devinez le prototype du constructeur de copie ? Ecrivez aussi le corps de cette méthode.
Réponse :

Fichier `.h`

```
class Point
{
  private :
    ...
  public :
    ...
    Point( const Point & p );
};
```

Fichier `.cc`

```
Point::Point( const Point & p )
{
   cout << "Point::constructeur par copie" << endl;
   my_abs = p.my_abs;
   my_ord = p.my_ord;
}
```

Le constructeur de copie est également appelé lors de la déclaration suivante pour `p` :

```
Point p1( 4, 5 );
Point p = p1;
```

et aussi lors du retour d'un point dans une fonction, comme par exemple dans la fonction `milieu`, si on retourne un point plutôt que de le passer par référence :

```
Point milieu( const Point & a, const Point & b )
```