f3.1 Notion de classe

Voici la **déclaration** d'une classe **Point** qui remplit le même rôle que la structure **point** vue précédemment.

```
typedef struct
{
    double x;
    double y;
}point;
```

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

public:
    Point(double x, double y);
    Point();
    double getX();
    double getY();
    void setX(double x);
    void setY(double y);
};
```

Les deux **attributs** x et y jouent le rôle des champs de la structure point.

Nous avons ici deux **constructeurs** permettant d'initialiser des instances de **Point** lors de leur création, dont un **constructeur par défaut**, sans paramètre.

Nous avons ici des **méthodes d'accès** qui permettent de lire et modifier les attributs.

(Nous verrons plus tard la signification de **public** et **private**.)

Exercices de découverte

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

public:
    Point(double x, double y);
    Point();
    double getX();
    double getY();
    void setX(double x);
    void setY(double y);
};
```

Qu'est-ce ?

Qu'est-ce ?

Comment appelle-t-on ce constructeur sans paramètre ?

Quel est le rôle de cette méthode ?

Quel est le rôle de cette méthode ?

f3.2 Définition d'un constructeur

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

public:
    Point(double x, double y);

    Point();

    double getX();
    double getY();
    void setX(double x);
    void setY(double y);
};
```

Pour que ce constructeur soit utilisable, il faut le définir. Une pratique recommendée consiste à placer la déclaration dans un fichier d'entête Point.h et la définition dans un fichier source Point.cpp. Pour simplifier, en phase de découverte, nous mettrons les deux dans un même fichier.

```
Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

this est l'adresse de l'instance en cours de construction. **this->x** est un raccourci d'écriture pour **(*this).x**

Exercices de découverte

```
Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

Ceci est :

- la définition d'un constructeur ☐
- la déclaration d'un constructeur ☐

```
Point::Point()
{


}
```

Complétez la définition du constructeur par défaut de manière à ce qu'il initialise les deux attributs avec la valeur 0.0

Tracez un trait entre chacun des deux éléments suivants et sa désignation dans le constructeur à 2 paramètres.

this->x	Ceci est le paramètre x du constructeur.
x	Ceci est l'attribut x de l'instance en cours de construction

f3.3 Utilisation d'un constructeur dans la pile

Cette fonction crée deux instances de **Point** et les place dans des variables locales, situées dans la pile.

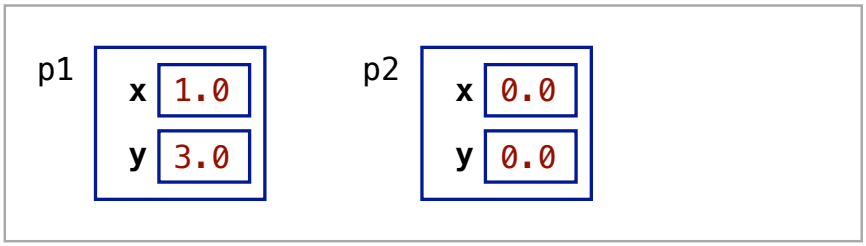
```
void test()
{
    Point p1(1.0, 3.0);

    Point p2;
}
```

p1 est initialisée par le constructeur à 2 paramètres.

p2 est initialisée avec le constructeur par défaut. Notez que dans ce cas il ne faut pas de parenthèses. **Point p2()** est incorrect parce qu'il peut être confondu avec la déclaration d'une fonction.

Configuration en mémoire dans la PILE

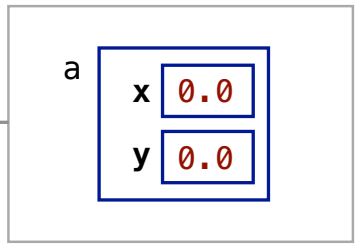


Exercices de découverte

Point p();

Cette déclaration de variable est-elle correcte ?

Donnez deux manières différentes de produire cette configuration en mémoire dans la pile.



f3.4 Méthodes d'accès

```
void Point::setX(double x)
{
    this->x = x;
}

void Point::setY(double y)
{
    this->y = y;
};

double Point::getX()
{
    return x;
}

double Point::getY()
{
    return y;
}
```

Voici les **définitions** des méthodes d'accès aux attributs de la classe **Point**.

Exemple d'utilisation

```
void test()
{
    Point p(1.0, 3.0);
    p.setX(2.0);
    printf("%.02f %.02f\n", p.getX(), p.getY());
}
```

Les attributs de **Point** sont privés (**private**). Seules les méthodes de la classe **Point** peuvent y accéder directement avec une notation telle que **p.x**. La fonction **test** doit donc utiliser des méthodes d'accès publiques pour lire ou modifier ces attributs.

(**printf** peut être utilisé en C++, mais il y a un autre moyen d'affichage que nous verrons plus tard.)

Exercices de découverte

Soit **p** une variable contenant une instance de **Point**.
Donnez **une** ligne de code permettant de déclarer et initialiser une variable **q** ayant exactement le même contenu que **p**. Cette déclaration est supposée être faite dans une fonction n'ayant pas accès aux attributs privés de **Point**.

Dans cette définition, peut-on remplacer this->x par x ? Justifiez brièvement votre réponse.

```
void Point::setX(double x)
{
    this->x = x;
}
```

f3.5 Tableaux d'instances de classe

```
void test()
{
    Point tab1[2];

    tab1[0] = Point(1.0, 3.0);

    Point* tab2 = new Point[2];

    tab2[1] = tab1[0];

    //…

    delete[] tab2;
}
```

Voici un tableau situé dans la pile contenant 2 instances de **Point** initialisées avec le constructeur par défaut de la classe **Point**. Si la classe **Point** n'a pas de constructeur par défaut, ce code ne compile pas.

La première cellule reçoit une instance de **Point** anonyme créée par le constructeur à 2 paramètres.

tab2 pointe une tableau situé dans le tas contenant 2 instances de **Point** initialisées avec le constructeur par défaut de la classe **Point**.

Le tableau dynamique pointé par **tab2** s'utilise exactement comme un tableau local tel que **tab1**.

Le tableau dynamique doit être détruit explicitement par appel à **delete[]**, quand il n'est plus utilisé, alors que le tableau situé dans la pile est détruit automatiquement quand la fonction termine son exécution.

Exercices de découverte

```
void test()
{
    Point tab1[2];

    tab1[0] = Point(1.0, 3.0);

    Point* tab2 = new Point[2];

    tab2[1] = tab1[0];

    //…

    delete[] tab2;
}
```

Dans quelle partie de la mémoire est situé le tableau **tab1** ?

Que contiennent les cellules de **tab1** juste après sa création ?

Ais-je le droit d'écrire **tab1 = tab2** et si oui, quelle est l'effet de cette assignation ?

Ais-je le droit d'écrire **tab2 = tab1** et si oui, quelle est l'effet de cette assignation ?

Pourquoi **tab1** ne doit pas être détruit par **delete[] tab1** ?

f3.6 Instance courante

Ceci est une **méthode d'instance** permettant d'afficher le point représenté par **l'instance courante**.

```
void Point::print()
{
    std::cout << "(" << getX() << "," << getY() << ")";
}
```

L'instance courante ?

Nous découvrons ici un moyen permettant, en C++, d'afficher des informations en les envoyant, avec l'opérateur <<, dans le flux de sortie **cout** (pour console output).

Tout appel d'une méthode d'instance d'une classe Q est lié à une instance Q qui, pendant l'exécution de la méthode, est appelée **instance courante**. Ici, les appel à **getX** et **getY** sont implicitement liés à l'instance courante. On aurait peut écrire **this->getX** et **this->getY**.

```
void test()
{
    Point p(1.0, 2.0);
    p.print();
}
```

Voici un exemple d'appel de la méthode **print** liée à l'instance de **Point** située dans la variable **p**. Pendant l'exécution de **print**, cette instance sera l'instance courante.

Exercices de découverte

```
void Point::print()
{
    std::cout << "(" << x << "," << y << ")";
}
```

Voici une variante de la méthode **print** de la classe **Point**. Cette méthode est-elle valide ? (Compilation et exécution à l'identique de la méthode originale)

```
void Point::print()
{
    std::cout << "(" << this->x << "," << this->y << ")";
}
```

Même question pour cette variante.

```
void test()
{
    Point tab1[2];
    Point* tab2 = new Point[2];
    //…
}
```

Donnez les lignes de code permettant d'afficher les instances de **Point** situées dans **tab1[0]** et **tab2[0]**.

f4.1 Instances en paramètres et en valeur de retour

Ceci est une **méthode d'instance** permettant de calculer le point situé au milieu du segment reliant le point représenté par **l'instance courante** et celui représenté par le paramètre **autre**.

```
Point Point::milieu( Point autre )
{
    double mx = (this->x + autre.x) / 2.0;

    double my = (this->y + autre.y) / 2.0;

    return Point(mx, my);
}
```

Le paramètre **autre** reçoit une copie d'instance de **Point**.

La valeur de retour est une instance de **Point** anonyme.

Voici un exemple d'utilisation de la méthode **milieu**.

```
void test()
{
    Point p1(1.0, 2.0);
    Point p2(3.0, 6.0);
    Point m = p1.milieu(p2);
    m.print();
}
```

Exercices de découverte

```
void test()
{
    Point p1;
    Point p2(0.0,2.0);
    Point p3(1.0,3.0);

    Point m = p3.milieu(p2.milieu(p1));

    //…
}
```

Cette ligne est-elle correcte ? Si oui, quelles sont les coordonnées **x** et **y** du point représenté par **m** ?

Complétez la définition de cette méthode pour qu'elle fasse la même chose que la variante présentée ci-dessus, mais en acceptant un paramètre de type pointeur sur **Point**.

```
Point Point::milieu( Point* autre )
{
    double mx =                          / 2.0;

    double my =                          / 2.0;

    Point m(mx, my);
    return m;
}
```

f4.2 Spécificité des classes monolithiques

Nous appellerons **classe monolithique** toute classe dont les instances sont constituées d'un unique bloc de mémoire. C'est le cas, par exemple, de la classe **Point**. Une des particularités de ces classes est qu'elles ne nécessitent pas de définir un **destructeur** explicite. Elles sont pourvues d'un destructeur implicite, invisible, produit automatiquement par le compilateur. A la fin de l'exécution de toute fonction ou méthode créant des variables locales ayant pour type une telle classe, ce destructeur est appelé de manière silencieuse pour détruire les instances situées dans ces variables. L'initialisation l'assignation d'instances de classes monolithiques avec = ne nécessite pas de précaution particulière.

```
void test()
{
    Point p1(1.0, 2.0);
    Point p2 = p1;
    p2.setY(3,0);
    p1 = p2;
}
```

Le symbole = de cette ligne initialise **p2** avec un **constructeur en copie implicite** qui recopie le contenu de **p1** dans **p2**.

Le symbole = de cette ligne recopie dans **p1** le contenu de **p2**.

Exercices de découverte

```
void test()
{
    Point p1(1.0, 2.0);
    Point p2 = p1;
    p2.setY(3,0);
    p1 = p2;
}
```

Réécrivez cette ligne sans utiliser le symbole =, en utilisant le constructeur à deux paramètres.

Réécrivez cette ligne sans utiliser le symbole =, en utilisant les méthodes d'accès.

f4.3 Passage de paramètres par référence

```
Point Point::milieu( Point& autre )
{
    double mx = (this->x + autre.x) / 2.0;

    double my = (this->y + autre.y) / 2.0;

    return Point(mx, my);
}
```

Dans cette variante de la méthode d'instance **milieu**, la paramètre **autre** est de type **référence sur Point**.
Le seule différence dans le code est le type du paramètre : **Point&** au lieu de **Point**.

À la différence de ce qui se passe avec le passage de paramètre par valeur, quand cette méthode est appelée, le paramètre ne reçoit pas une copie d'instance de **Point** mais un "pointeur caché" sur une instance de **Point**.
On a les avantages d'un passage par pointeur, mais avec une syntaxe simplifiée : pas de dépointage, on utilise le paramètre comme on le ferait avec la variable passée par référence.

```
void test()
{
    Point p1(1.0, 2.0);
    Point p2(3.0, 6.0);
    Point m = p1.milieu(p2);
    m.print();
}
```

La méthode avec passage par référence d'utilise exactement de la même manière que la version avec passage par valeur.

Exercices de découverte

```
void Point::exg(Point& autre)
{




}
```

Réalisez une méthode d'instance **void Point::exg()** qui échange les valeurs des attribut **x** et **y** de l'instance courante et de ceux d'une instance de **Point** passée par référence.

Réalisez une fonction **void test_exg()** qui déclare deux variables **p1** et **p2** de type **Point**, initialisées respectivement avec les coordonnées (1.0, 2.0) et (3.0, 4.0), puis qui appelle **exg** pour échanger les contenus de ces variables, puis qui affiche les valeurs de **p1** et **p2**.

```
void test_exg()
{




}
```

f5 Méthode de classe

```
class Point
{

public:

  static Point milieu(const Point& a, const Point& b);

};
```

Une **méthode de classe** doit être déclarée **static** dans sa classe.

```
Point Point::milieu(const Point& a, const Point& b)
{
    double mx = (a.x + b.x) / 2.0;
    double my = (a.y + b.y) / 2.0;
    return Point(mx, my);
}
```

Lors de son exécution, cette méthode n'a pas accès à une instance courante. On ne peut utiliser **this**.

```
void test()
{
    Point p1(1.0, 2.0); Point p2(3.0, 6.0);
    Point m = Point::milieu(p1,p2);
    m.print();
}
```

L'appel d'une méthode de classe n'est pas lié à une instance. Les deux points dont on veut le milieu sont passés en paramètres, ici par référence.

Exercices de découverte

Complétez cette **méthode de classe** (déclarée **static** dans la classe **Point**) pour qu'elle retourne une instance de **Point** de coordonnées (1.0, 1.0).

```
Point Point::makeUnit()
{



}
```

Donnez la ligne de code permettant de créer une variable de type **Point** et de l'initialiser par un appel à la méthode de classe **makeUnit**. Celle ligne est supposée se trouver dans une fonction n'appartenant pas à la classe **Point**.