Programmation Orientée Objet

Les classes

charles.meunier@u-bourgogne.fr

DES STRUCTURES AUX CLASSES

charles.meunier@u-bourgogne.fr

LES STRUCTURES

```
struct Point
{
    int x;
    int y;
};

void move(struct Point& pt, const struct Point& vector)
{
    pt.x += vector.x;
    pt.y += vector.y;
}

int main()
{
    struct Point pt1;
    struct Point pt2 = {10, 20};

    pt1.x = pt1.y = 0;

    move(pt1, pt2);

    return 0;
}
```

ABSTRACTION

Les structures permettent de regrouper des données au sein d'un ensemble cohérent.

ENCAPSULATION

Les traitements en rapport avec le concept sont réalisés par des fonctions externes à ce dernier qui doivent prendre en paramètre l'instance de la structure à manipuler.

MODULARITÉ

Les structures permettent une décomposition du problème mais ne garantissent pas la protection des données.

LES CLASSES

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

    public:
        Point();
        Point(int vx, int vy);

        void move(const Point& vector);
};

int main()
{
    Point pt1;
    Point pt2(10, 20);

    pt1.move(pt2);

    return 0;
}
```

ABSTRACTION

Les classes permettent de regrouper des données au sein d'un ensemble cohérent.

ENCAPSULATION

Les traitements en rapport avec le concept sont réalisés par des fonctions interne à la classe.

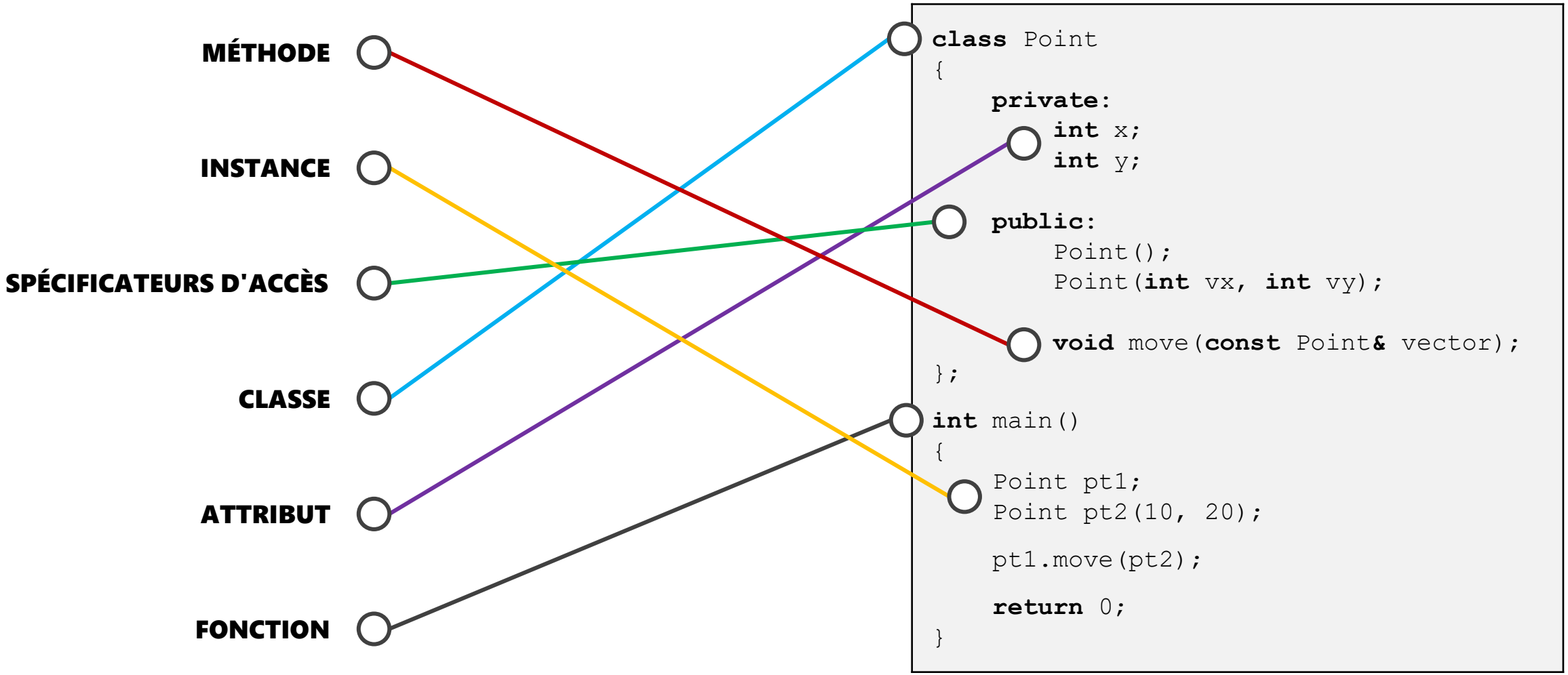
MODULARITÉ

Les classes offrent une décomposition qui permet une meilleure compréhension du problème et garantissent que les données sont protégées.

PAUSE VOCABULAIRE

charles.meunier@u-bourgogne.fr

QUI EST QUOI ?


MÉTHODE
INSTANCE
SPÉCIFICATEURS D'ACCÈS
CLASSE
ATTRIBUT
FONCTION
class Point
{
private:
int x;
int y;
public:
Point();
Point(int vx, int vy);
void move(const Point& vector);
};
int main()
{
Point pt1;
Point pt2(10, 20);
pt1.move(pt2);
return 0;
}

ATTRIBUTS, MÉTHODES, MEMBRES

Les **ATTRIBUTS** sont les données qui caractérisent le concept décrit par la classe. Il est recommandé de les définir privés afin de contrôler leur état.

Les **MÉTHODES** sont des fonctions définies au sein de la classe et qui réalisent des actions en lien avec le concept décrit par la classe. Les méthodes d'une classe peuvent être privées ou publiques.

Les **MEMBRES** d'une classe sont constitués de ses attributs et de ses méthodes.

MÉTHODES VS FONCTIONS

Une **FONCTION** est un bloc de code qui effectue une tâche spécifique.

Une **MÉTHODE** est une fonction qui est définie à l'intérieure d'une classe et qui peut accéder à ses membres.

CLASSES, INSTANCES, OBJETS

ACTEUR

CONCEPT
=
CLASSE

ELEMENTS DU RÉEL
=
INSTANCES / OBJETS

CLASSES, INSTANCES, OBJETS

```
class Point
{
    ...
};

int main()
{
    Point pt1;

    ...
}
```

pt1 est une **INSTANCE DE LA CLASSE** Point

```
#include <string>

int main()
{
    std::string str;

    ...
}
```

str est une **INSTANCE DE LA CLASSE** std::string

```
int main()
{
    int a;

    ...
}
```

i est une **VARIABLE DE TYPE** int

LE TERME INSTANCE EST RÉSERVÉ AUX OBJETS D'UNE CLASSE.

SPÉCIFICATEURS D'ACCÈS

Les spécificateurs d'accès permettent de **CONTRÔLER L'ACCÈS AUX MEMBRES D'UNE CLASSE**.

PRIVATE

Seuls les méthodes membres de la classe peuvent accéder aux membres privées de cette dernière.

PUBLIC

Les membres public sont accessibles depuis l'intérieur comme l'extérieur de la classe

PROTECTED

On verra plus tard… :-)

SPÉCIFICATEURS D'ACCÈS

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

    public:
        Point();
        Point(int vx, int vy);

        void move(const Point& vect);
};

int main()
{
    Point pt1;
    Point pt2;

    pt1.x = 0;

    pt1.move(pt2);

    return 0;
}
```

Les attributs `x` et `y` sont déclarés **PRIVÉS**

La méthode `move` est déclarée **PUBLIQUE**

`main` ne peut pas accéder directement à `x` => **ERREUR**

`main` peut accéder à `move` => **OK**

CYCLE DE VIE D'UNE INSTANCE

charles.meunier@u-bourgogne.fr

INSTANCIATION

```
class Point
{
    ...
};

int main()
{
    Point pt1;

    ...

    return 0;
}
```

DURÉE DE VIE DE L'INSTANCE PT1

- **RÉSERVATION DE L'ESPACE MÉMOIRE**
- **APPEL DU CONSTRUCTEUR**
- **APPEL DU DESTRUCTEUR**
- **LIBÉRATION DE L'ESPACE MÉMOIRE**

CONSTRUCTEUR

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

    public:
        Point();
        Point(int vx, int vy);

        ...
};
```

- Méthode appelée **AUTOMATIQUEMENT** à la création de l'objet et qui initialise les attributs de l'instance.
- Le constructeur ne possède **PAS DE VALEUR DE RETOUR**, pas même `void`.
- En C++, le constructeur **PORTE LE MÊME NOM QUE LA CLASSE**.
- En C++, une classe peut posséder plusieurs constructeurs.

CONSTRUCTEUR PAR DÉFAUT

```
class Point
{
    private:
        int x;
        int y;
};
```

Chaque classe dispose d'un constructeur par défaut qui tente d'initialiser les attributs en appelant leur propre constructeurs s'ils en ont.

Dans l'exemple ci-contre, aucun constructeur n'a été défini pour décrire comment initialisé les attributs `x` et y.

Or `x` et y sont des `int`, un type natif de C++ qui n'est pas une classe et qui ne dispose donc pas de constructeur.

`x` et y seront donc initialisés avec les valeurs présentes en mémoire lors de la réservation de l'espace mémoire de l'instance, ce qui n'est pas terrible.

CONSTRUCTEUR PAR DÉFAUT

point.h

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

    public:
        Point();
};
```

Il est possible de redéfinir le constructeur par défaut d'une classe afin de décrire précisément comment initialiser les attributs de l'instance.

Le constructeur par défaut ne prend pas de paramètre en entrée.

point.cpp

```
Point::Point()
{
    x = 0;
    y = 0;
}
```

La spécification d'une méthode est toujours précédé d'un opérateur de résolution de portée `::` indiquant à quel classe appartient la fonction.

Chaque classe crée son propre espace de nom.

LIGNE D'INITIALISATION

point.cpp

```
Point::Point()
    : x(0), y(0)
{
}
```

Le C++ propose une syntaxe offrant de meilleures performances dans l'initialisation des attributs : la ligne d'initialisation.

La ligne d'initialisation se place juste avant l'accolade ouvrante du constructeur et est précédée du symbole :

La ligne d'initialisation contient la liste des attributs de la classe, séparés par une virgule, associés à leur valeur d'initialisation entre parenthèses.

CONSTRUCTEUR PAR RECOPIE

point.h

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

    public:
        Point();
        Point(const Point& pt);
};
```

point.cpp

```
Point::Point(const Point& pt)
    : x(pt.x), y(pt.y)
{
}
```

Le constructeur par recopie a pour but d'initialiser les attributs de l'instance à partir des attributs d'une autre instance de la classe.

Le constructeur par recopie prend en paramètre une référence constante sur l'instance à copier.

Attention : une instance d'une classe peut accéder aux attributs privés d'une autre instance de la même classe. A utiliser avec précaution pour ne pas violer le principe d'encapsulation.

AUTRES CONSTRUCTEURS

point.h

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

    public:
        Point();
        Point(const Point& pt);
        Point(int vx, int vy);
        Point(const std::string& str);
        Point(int a, int b);
};
```

Il est possible de créer autant de constructeurs qu'on le souhaite à condition qu'ils aient tous une signature différente.

SIGNATURE D'UNE FONCTION
=
NOM DE LA FONCTION
+
TYPE DES PARAMÈTRES D'ENTRÉE.

ERREUR : UNE MÉTHODE AVEC LA MÊME SIGNATURE EXISTE DÉJÀ !

DESTRUCTEUR

point.h

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

    public:
        Point();
        ~Point();

        ...
};
```

point.cpp

```
Point::~Point()
{
    ...
}
```

- Méthode appelée **AUTOMATIQUEMENT** à la destruction de l'objet et qui a pour but de libérer les espaces alloués dynamiquement en mémoire par l'instance.
- Le destructeur ne possède **NI PARAMÈTRES, NI VALEUR DE RETOUR**, pas même `void`.
- En C++, le destructeur **PORTE LE MÊME NOM QUE LA CLASSE PRÉCÉDÉ PAR LE SYMBOLE ~**.
- Chaque classe possède un destructeur par défaut. Dans l'exemple ci-contre, il ne serait pas utile de le redéfinir car le destructeur n'a rien à faire ici (`x` et `y` sont alloués de manière statique).

MÉTHODES

charles.meunier@u-bourgogne.fr

IMPLÉMENTATION DES MÉTHODES

point.h

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

    public:
        Point();
        Point(const Point& pt);
        Point(int vx, int vy);

        void move(const Point& vector);
};
```

point.cpp

```
Point::Point()
    : x(0), y(0)
{
}

Point::Point(const Point& pt)
    : x(pt.x), y(pt.y)
{
}

Point::Point(int vx, int vy)
    : x(vx), y(vy)
{
}

void Point::move(const Point& vector)
{
    x += vector.x;
    y += vector.y;
}
```

N'OUBLIEZ PAS L'OPÉRATEUR DE PORTÉE

OPÉRATEURS

point.h

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

    public:
        Point();
        Point(const Point& pt);
        Point(int vx, int vy);

        void move(const Point& vector);

        Point operator+(const Point& pt);
};
```

point.cpp

```
...
Point Point::operator+(const Point& pt)
{
    return Point(x + pt.x, y + pt.y);
}
```

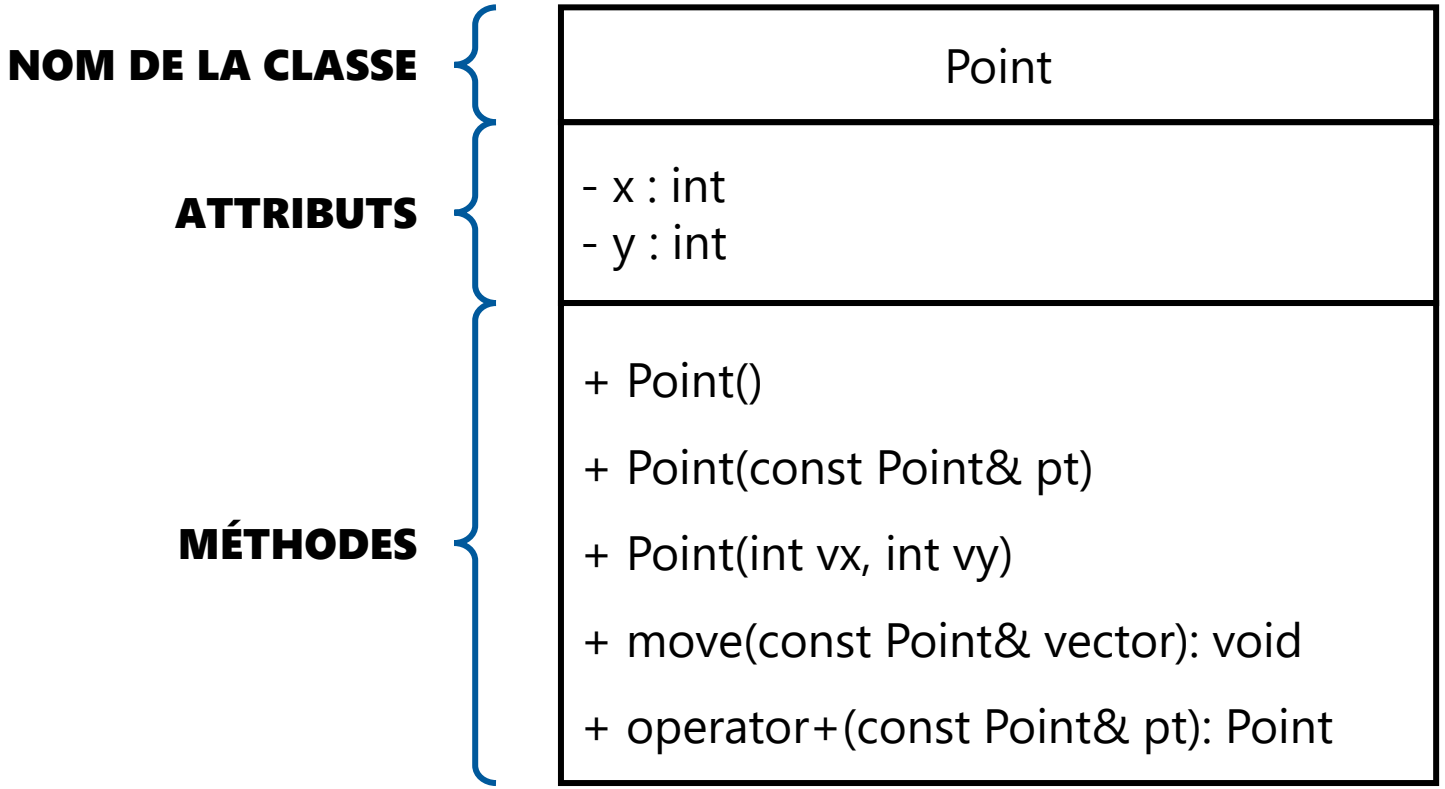
main.cpp

```
int main()
{
    Point pt1(3, 2), pt2(12, -3);

    Point pt3(pt1 + pt2)

    return 0;
}
```

REPRÉSENTATION UML

charles.meunier@u-bourgogne.fr

LES CLASSES EN UML



LES INSTANCES / OBJETS EN UML

NOM DE L'OBJET : TYPE

VALEUR DES ATTRIBUTS

origine : Point	ptA : Point	ptB : Point
x = 0 y = 0	x = 3 y = 2	x = -8 y = 7

POUR TERMINER…

charles.meunier@u-bourgogne.fr

PETIT EXERCICE

DONNEZ LA MODÉLISATION UML DE LA CLASSE CORRESPONDANT AUX IMAGES SUIVANTES



Note : ils/elles courent, mangent, aboient et porteront le nom que vous leur donnerez