



Programmation Orientée Objet

Maxime MARIA

maxime.maria@u-pem.fr



PROGRAMMATION ORIENTÉE OBJET EN C++

On commence les choses sérieuses...

Exemple de définition d'une classe

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

private:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Personnage

m_niveau

m_vie

m_attaque

monterNiveau()

attaquer()

Exemple de définition d'une classe

Ça commence bien son truc...

Comment on utilise un objet ?

Alors, "class", ça doit être une classe...

Ça veut dire quoi ?

On peut mettre quoi comme attributs ?

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

private:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Personnage

m_niveau

m_vie

m_attaque

monterNiveau()

attaquer()

Où sont définies les méthodes ?

Comment j'appelle une méthode ?

Pourquoi dans ce sens ?





**KEEP
CALM
AND
LEARN TO
CODE**

Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

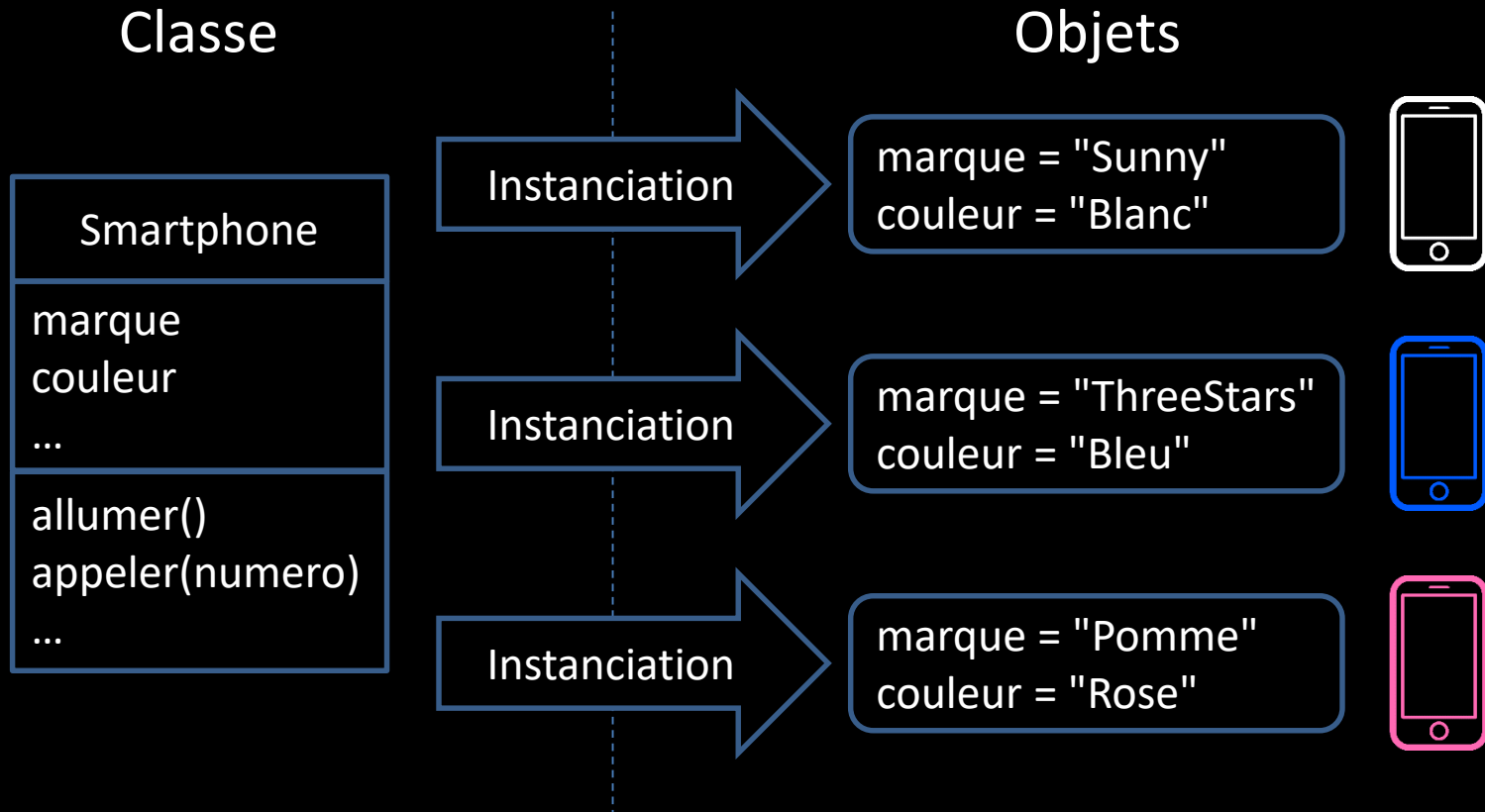
Notion de classe (rappel)

- Classe = structure de données représentant un objet
- Deux types de membres :
 - Attributs : variables définissant l'état de l'objet
 - Méthodes : fonctions définissant le comportement de l'objet

CompteBancaire	← Nom de la classe
nomTitulaire solde	← Attributs
deposerArgent(montant) retirerArgent(montant)	← Méthodes

Notion d'objet (rappel)

- Objet = instance de classe



- Instantiation = construction

Définition d'une classe (1)

Personnage

m_niveau

m_vie

m_attaque

monterNiveau()

attaquer()



Définition d'une classe (1)

- Dans fichier d'en-tête (*header*) : .h, .hpp, ...
 - Mot-clef `class`

personnage.hpp

```
class Personnage  
{  
};
```

← Ne pas oublier le point-virgule !

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()



Définition d'une classe (1)

- Dans fichier d'en-tête (*header*) : .h, .hpp, ...

- Mot-clef `class`

personnage.hpp

```
class Personnage
{
};
```

Ne pas oublier le point-virgule !

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()

- Déclaration des attributs

personnage.hpp

```
class Personnage
{
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Convention de nommage pour distinguer les attributs des variables : `m_vie`, `_vie`, `mVie`, ...



Définition d'une classe (1)

- Dans fichier d'en-tête (*header*) : .h, .hpp, ...
 - Mot-clef `class`

personnage.hpp

```
class Personnage
{
};
```

Ne pas oublier le point-virgule !

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()

- Déclaration des attributs
- Types d'attributs ?

personnage.hpp

```
class Personnage
{
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Convention de nommage pour distinguer les attributs des variables : `m_vie`, `_vie`, `mVie`, ...

- Primitifs (int, float, char)
- Tableaux
- Autres classes !
- Bref, ce qu'on veut...



Définition d'une classe (2)

- Déclaration des méthodes

personnage.hpp

```
class Personnage
{
    // Méthodes
    void monterNiveau();
    int attaquer();

    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Personnage

m_niveau

m_vie

m_attaque

monterNiveau()

attaquer()



Définition d'une classe (2)

- Déclaration des méthodes

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()

personnage.hpp

```
class Personnage
{
    // Méthodes
    void monterNiveau();
    int attaquer();

    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

- Définition

- Dans un fichier source : .cpp, .cc, ...

personnage.cpp

```
#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

On retrouve l'opérateur de
résolution de portée ::



Exemple de définition d'une classe

Ok, c'est déjà mieux !

Comment on utilise un objet ?

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()

Alors, "class", ça doit être une classe...

Ça veut dire quoi ?

On peut mettre quoi comme attributs ?

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

private:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Où sont définies les méthodes ?

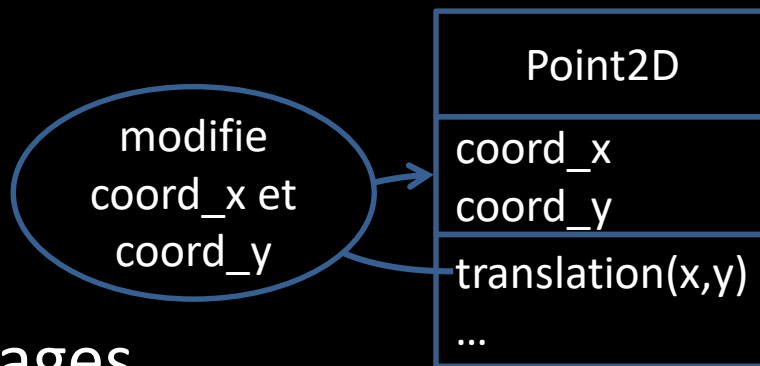
Comment j'appelle une méthode ?

Pourquoi dans ce sens ?



Encapsulation (rappel)

- Implémentation interne cachée
 - On ne manipule pas directement les attributs
 - On utilise les méthodes mises à disposition (interface)
- Exemple : translation d'un point 2D



```
Point2D p;  
p.coord_x += x;  
p.coord_y += y;  
p.translation(x, y);
```

- Avantages
 - Facilite la réutilisation, la maintenance et l'évolution du code
 - Le développeur ne se préoccupe pas de l'implémentation

Visibilité



- Règle le degré d'encapsulation
- Détermine les droits d'accès aux membres d'une classe
- 3 types de modificateurs d'accès :
 - private : accessible QUE depuis les méthodes de la classe
 - public : accessible partout
 - (protected : nous verrons ça dans le prochain CM)
- Portée :
 - Jusqu'au prochain modificateur (ou la fin de classe)

Par défaut !

Exemple

```
class CompteBancaire
{
private:
    float m_solde;

public:
    std::string m_nom;

private:
    void retirer(const float montant)
    {
        m_solde -= montant;
    }

public:
    void deposer(const float montant)
    {
        m_solde += montant;
    }
};
```

OK car public

```
int main()
{
    CompteBancaire cpt;

    std::string nom = cpt.m_nom;

    float solde = cpt.m_solde;

    cpt.deposer(79.f);

    cpt.retirer(17.f);

    return 0;
}
```

Erreur car
private

VOUS NE PASSEREZ PAS !



C'EST PRIVÉ !

Respect du principe d'encapsulation

- Pour une bonne programmation OO :
 - Tout attribut doit être privé
 - Toute méthode non nécessaire à l'utilisateur doit être privée
 - Les autres méthodes sont publiques (interface)

Respect du principe d'encapsulation

- Pour une bonne programmation OO :
 - Tout attribut doit être privé
 - Toute méthode non nécessaire à l'utilisateur doit être privée
 - Les autres méthodes sont publiques (interface)

Mais alors on ne peut pas
accéder aux attributs ?
C'est un peu limité votre
truc alors...



Accesseur (Getter) et Mutateur (Setter)

- Accesseur : méthode pour lire un attribut
- Mutateur : méthode pour modifier un attribut

Accesseur (Getter) et Mutateur (Setter)

- Accesseur : méthode pour lire un attribut
- Mutateur : méthode pour modifier un attribut

```
class UneClasse
{
public:
    // Getter
    int getAttribut() const
    {
        return m_attribut;
    }
    // Setter
    void setAttribut(const int a)
    {
        m_attribut = a;
    }
private:
    int m_attribut;
};
```

Accesseur (Getter) et Mutateur (Setter)

- Accesseur : méthode pour lire un attribut
- Mutateur : méthode pour modifier un attribut

```
class UneClasse
{
public:
    // Getter
    int getAttribut() const
    {
        return m_attribut;
    }
    // Setter
    void setAttribut(const int a)
    {
        m_attribut = a;
    }
private:
    int m_attribut;
};
```

```
int main()
{
    UneClasse objet;

    objet.setAttribut(79);

    int attr = objet.getAttribut();

    return 0;
}
```

VOUS AVEZ UN GETTER ?



ALLEZ Y, PASSEZ !

Accesseur (Getter) et Mutateur (Setter)

- Accesseur : méthode pour lire un attribut
- Mutateur : méthode pour modifier un attribut

```
class UneClasse
{
public:
    // Getter
    int getAttribut() const
    {
        return m_attribut;
    }
    // Setter
    void setAttribut(const int a)
    {
        m_attribut = a;
    }
private:
    int m_attribut;
};
```

const ?

```
int main()
{
    UneClasse objet;

    objet.setAttribut(79);

    int attr = objet.getAttribut();

    return 0;
}
```

VOUS AVEZ UN GETTER ?



ALLEZ Y, PASSEZ !

Méthode constante (const)

- Méthode ne modifiant pas les attributs
 - Identifiée par `const` à la fin de la signature



À spécifier
dès que possible !

```
class UneClasse
{
public:
    // Getter
    int getAttribut() const
    {
        return m_attribut;
    }
    // Setter
    void setAttribut(const int a)
    {
        m_attribut = a;
    }
private:
    int m_attribut;
};
```

Ordre de déclaration des membres

- Aucun ordre imposé !
- Il faut quand même organiser sa classe...
- Un exemple (pas forcément le meilleur) :
 1. Interface (méthodes publiques) au début : directement visible
 2. Méthodes privées
 3. Attributs (privés par encapsulation)



Exemple de définition d'une classe

Reste plus
qu'à savoir
s'en servir...

Comment on utilise un objet ?

Personnage
m_niveau m_vie m_attaque
monterNiveau() attaquer()

Alors, "class", ça doit
être une classe...

Ça veut dire quoi ?

On peut mettre quoi
comme attributs ?

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

private:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Où sont définies
les méthodes ?

Comment j'appelle
une méthode ?

Pourquoi dans
ce sens ?



Création et utilisation d'objets

- Création : comme une variable
- Utilisation : via le point .

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

public:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};

#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

```
int main()
{
    Personnage batman;
    Personnage joker;

    joker.m_vie = 100;

    batman.m_attaque = 10;

    joker.m_vie -= batman.attaquer();

    std::cout << "Vie Joker = " << joker.m_vie << std::endl;
}
```



Création et utilisation d'objets

- Création : comme une variable
- Utilisation : via le point .

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

public:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};

#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

```
int main()
{
    Personnage batman;
    Personnage joker;

    joker.m_vie = 100;

    batman.m_attaque = 10;

    joker.m_vie -= batman.attaquer();

    std::cout << "Vie Joker = " << joker.m_vie << std::endl;
}
```

Création de deux objets Personnage : "batman" et "joker"



Création et utilisation d'objets

- Création : comme une variable
- Utilisation : via le point .

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

public:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};

#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

```
int main()
{
    Personnage batman;
    Personnage joker;

    joker.m_vie = 100;
    batman.m_attaque = 10;

    joker.m_vie -= batman.attaquer();

    std::cout << "Vie Joker = " << joker.m_vie << std::endl;
}
```

Création de deux objets Personnage : "batman" et "joker"

Initialisation de l'attribut "vie" de l'objet "joker"

Initialisation de l'attribut "attaque" de l'objet "batman"



Création et utilisation d'objets

- Création : comme une variable
- Utilisation : via le point .

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

public:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};

#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

```
int main()
{
    Personnage batman;
    Personnage joker;

    joker.m_vie = 100;
    batman.m_attaque = 10;
    joker.m_vie -= batman.attaquer();

    std::cout << "Vie Joker = " << joker.m_vie << std::endl;
}
```

Création de deux objets Personnage : "batman" et "joker"

Initialisation de l'attribut "vie" de l'objet "joker"

Initialisation de l'attribut "attaque" de l'objet "batman"

Appel de la méthode "attaquer()" de l'objet "batman"

Modification de l'attribut "vie" de l'objet "joker"



Création et utilisation d'objets

- Création : comme une variable
- Utilisation : via le point .

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

public:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};

#include "personnage.hpp"

void Personnage::monterNiveau()
{
    m_niveau++;
    m_vie++;
    m_attaque++;
}

int Personnage::attaquer()
{
    return m_attaque;
}
```

```
int main()
{
    Personnage batman;
    Personnage joker;

    joker.m_vie = 100;
    batman.m_attaque = 10;
    joker.m_vie -= batman.attaquer();

    std::cout << "Vie Joker = " << joker.m_vie << std::endl;
}
```

Création de deux objets Personnage : "batman" et "joker"

Initialisation de l'attribut "vie" de l'objet "joker"

Initialisation de l'attribut "attaque" de l'objet "batman"

Appel de la méthode "attaquer()" de l'objet "batman"

Modification de l'attribut "vie" de l'objet "joker"

→ « Vie Joker = 90 »



Exemple de définition d'une classe

Ah mais ça va en fait !

Comment créer un objet ? ✓

Alors, "class", ça doit être une classe... ✓

Ça veut dire quoi ? ✓

On peut mettre quoi comme attributs ? ✓

Où sont définies les méthodes ? ✓

Comment j'appelle une méthode ? ✓

Pourquoi dans ce sens ? ✓

```
class Personnage
{
public:
    // Méthodes
    void monterNiveau();
    int attaquer();

private:
    // Attributs
    int m_niveau;
    int m_vie;
    int m_attaque;
};
```

Personnage

m_niveau

m_vie

m_attaque

monterNiveau()

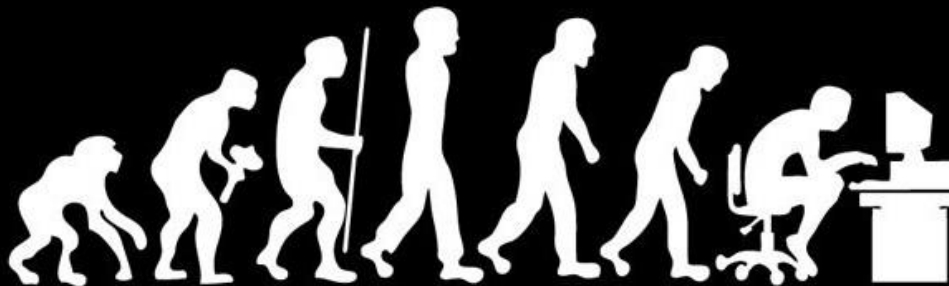
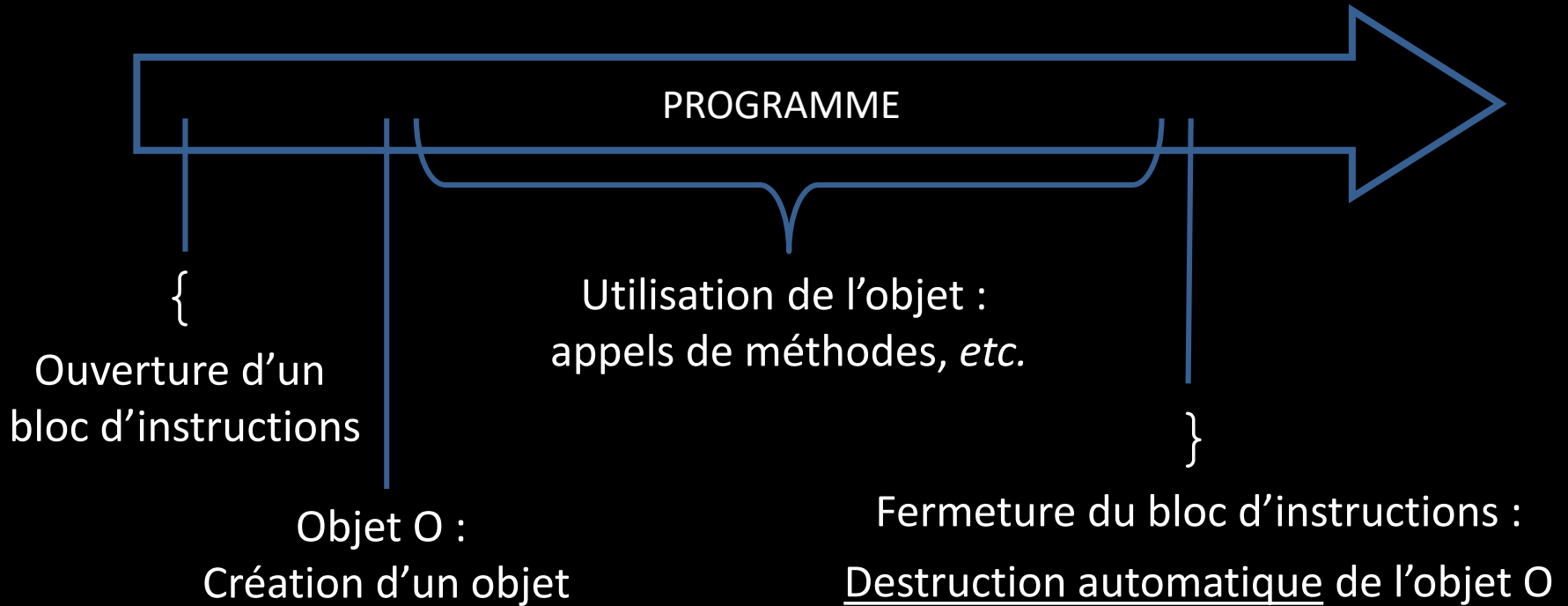
attaquer()



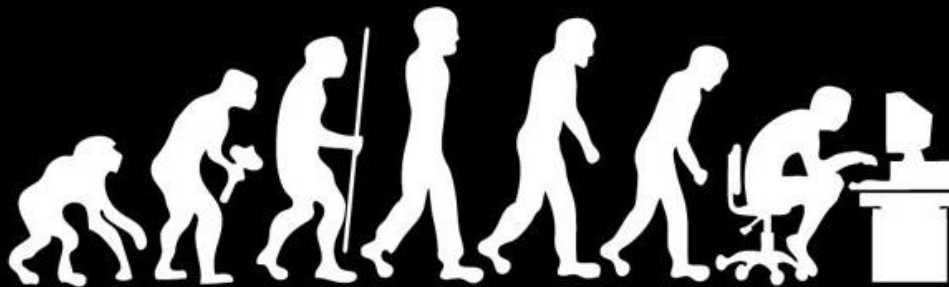
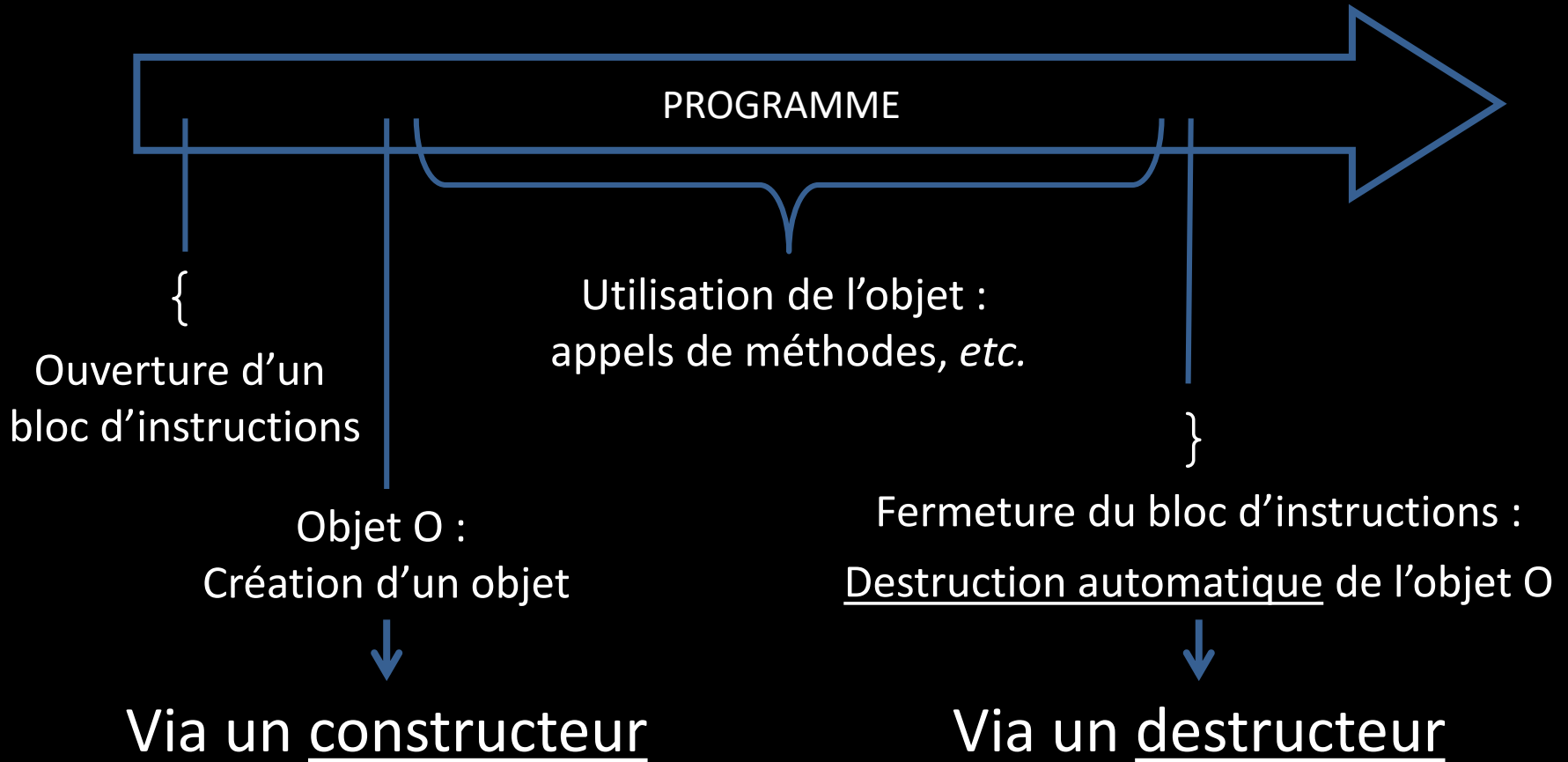
Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Durée de vie d'un objet



Durée de vie d'un objet



Constructeur

- Création d'un objet = constructeur :
 - Initialise les attributs
 - Porte obligatoirement le même nom que la classe
 - N'a jamais de type de retour (même pas `void`)
 - Peut avoir des paramètres (et être surchargé)



Constructeur

- Création d'un objet = constructeur :
 - Initialise les attributs
 - Porte obligatoirement le même nom que la classe
 - N'a jamais de type de retour (même pas `void`)
 - Peut avoir des paramètres (et être surchargé)

```
class Point2D
{
public:
    // Constructeur
    Point2D(const float x, const float y);

private:
    // Attributs
    float m_x;
    float m_y;
};
```

```
Point2D::Point2D(const float x, const float y)
{
    m_x = x;
    m_y = y;
}
```




Constructeur par défaut

- Et quand on fait `Point2D point;` ?
 - C'est le constructeur par défaut qui est appelé

Pas de paramètre !

`Point2D();`



```
Point2D::Point2D()  
{  
    m_x = 0.f;  
    m_y = 0.f;  
}
```


Constructeur par défaut

- Et quand on fait `Point2D point;` ?
 - C'est le constructeur par défaut qui est appelé

Pas de paramètre !

`Point2D();`

```
Point2D::Point2D()  
{  
    m_x = 0.f;  
    m_y = 0.f;  
}
```

Ou "fusion" de deux constructeurs
grâce aux valeurs par défaut...

`Point2D(const float x = 0.f, const float y = 0.f);`

```
Point2D::Point2D(const float x, const float y)  
{  
    m_x = x;  
    m_y = y;  
}
```

Liste d'initialisation

- Après la signature du constructeur ":"
- Initialise les attributs via un de leurs constructeurs
- Dans l'ordre de déclaration ! Peut éviter des bugs...

```
Point2D::Point2D(const float x, const float y)  
→ : m_x(x), m_y(y)  
{  
    // Si besoin on peut mettre du code ici !  
}
```



Liste d'initialisation

- Après la signature du constructeur " : "
 - Initialise les attributs via un de leurs constructeurs
 - Dans l'ordre de déclaration ! Peut éviter des bugs...

```
Point2D::Point2D(const float x, const float y)  
→ : m_x(x), m_y(y)  
{  
    // Si besoin on peut mettre du code ici !  
}
```

- Fonctionne avec n'importe quelle expression C++ valide

```
Point2D::Point2D(const float x, const float y)  
→ : m_x(x), m_y(y), m_dist(sqrtf(x * x + y * y))  
{  
    // Si besoin on peut mettre du code ici !  
}
```



Liste d'initialisation : avantages

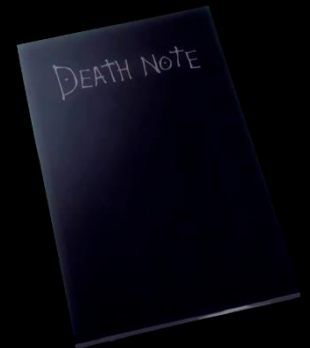
- S'exécute avant le bloc { }
- Sans la liste : initialisation (implicite) puis affectation

```
Point2D::Point2D(const float x, const float y)
{
    m_x = x;
    m_y = y;
}
```

Construction par défaut
de m_x et m_y

Puis affectation...

- Fonctionne pour les références et les constantes
- (Qui doivent être initialisées avant toute chose)
- À utiliser de façon systématique !



Constructeur par copie

- Copie d'un objet

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Référence !

Constructeur par copie

- Copie d'un objet

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Référence !

- Appelé quand :
 - Création d'un objet à partir d'un autre

```
Point2D autrePoint(point);
```

```
Point2D autrePoint2 = point;
```

Constructeur par copie

- Copie d'un objet

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Référence !

- Appelé quand :

- Création d'un objet à partir d'un autre

```
Point2D autrePoint(point);
```

Appel du constructeur par copie

```
Point2D autrePoint2 = point;
```

≠

```
Point2D autrePoint2;  
autrePoint2 = point;
```

Appel du constructeur par défaut puis affectation

Constructeur par copie

- Copie d'un objet

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Référence !

- Appelé quand :

- Création d'un objet à partir d'un autre

```
Point2D autrePoint(point);
```

Appel du constructeur par copie

```
Point2D autrePoint2 = point;
```

≠

Appel du constructeur par défaut puis affectation

```
Point2D autrePoint2;  
autrePoint2 = point;
```

- Passage d'un objet en paramètre par valeur

```
void afficher(const Point2D pt);
```

```
afficher(point);
```

Copie

Constructeur par copie

- Copie d'un objet

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Référence !

- Appelé quand :

- Création d'un objet à partir d'un autre

```
Point2D autrePoint(point);
```

Appel du constructeur par copie

```
Point2D autrePoint2 = point;
```

≠

Appel du constructeur par défaut puis affectation

```
Point2D autrePoint2;  
autrePoint2 = point;
```

- Passage d'un objet en paramètre par valeur

```
void afficher(const Point2D pt);
```

```
afficher(point);
```

Copie

- Retour d'une fonction retournant un objet

```
Point2D scale(const Point2D &pt, const float s);
```

```
autrePoint2 = scale(point, 2.f);
```

Copie du résultat de scale
(puis affectation)

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

```
Point2D *point;
```

← Création d'un pointeur vide

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

```
Point2D *point;
```

← Création d'un pointeur vide

```
point = new Point2D;
```

← Appel du constructeur par défaut

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

```
Point2D *point;
```

← Création d'un pointeur vide

```
point = new Point2D;
```

← Appel du constructeur par défaut

```
point = new Point2D(79.f, 17.f);
```

← Appel d'un constructeur

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

<code>Point2D *point;</code>	←	Création d'un pointeur vide
<code>point = new Point2D;</code>	←	Appel du constructeur par défaut
<code>point = new Point2D(79.f, 17.f);</code>	←	Appel d'un constructeur
<code>point = new Point2D(autrePoint);</code>	←	Appel du constructeur par copie

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

<code>Point2D *point;</code>	←	Création d'un pointeur vide
<code>point = new Point2D;</code>	←	Appel du constructeur par défaut
<code>point = new Point2D(79.f, 17.f);</code>	←	Appel d'un constructeur
<code>point = new Point2D(autrePoint);</code>	←	Appel du constructeur par copie
<code>point->afficher();</code>	←	Accès aux membres via <u>une flèche</u> ->

Allocation dynamique

- Comme pour un type primitif : `new` / `delete`
 - Pointeur vers un objet

<code>Point2D *point;</code>	←	Création d'un pointeur vide
<code>point = new Point2D;</code>	←	Appel du constructeur par défaut
<code>point = new Point2D(79.f, 17.f);</code>	←	Appel d'un constructeur
<code>point = new Point2D(autrePoint);</code>	←	Appel du constructeur par copie
<code>point->afficher();</code>	←	Accès aux membres via <u>une flèche</u> ->
<code>delete point;</code>	←	On n'oublie pas de libérer la mémoire !



Destructeur

- Destruction d'un objet = destructeur :
 - Méthode appelée automatiquement
 - Porte le même nom que la classe précédé d'un tilde ~
 - N'a jamais de type de retour
 - N'a jamais de paramètre (donc jamais surchargé)
- But : libérer les ressources



Destructeur

- Destruction d'un objet = destructeur :
 - Méthode appelée automatiquement
 - Porte le même nom que la classe précédé d'un tilde ~
 - N'a jamais de type de retour
 - N'a jamais de paramètre (donc jamais surchargé)
- But : libérer les ressources

```
class ListeEntier
{
public:
    ListeEntier(const int taille);
    ~ListeEntier();

private:
    int *m_data;
};
```

```
ListeEntier::ListeEntier(const int taille)
{
    m_data = new int[taille];
}

ListeEntier::~~ListeEntier()
{
    delete[] m_data;
}
```



Génération automatique du compilateur

- Le compilateur génère automatiquement si :
 - Constructeur par défaut non défini :
 - Appel des constructeurs par défaut des attributs

```
Point2D::Point2D()  
: m_x(), m_y()  
{}
```

Génération automatique du compilateur

- Le compilateur génère automatiquement si :
 - Constructeur par défaut non défini :
 - Appel des constructeurs par défaut des attributs

```
Point2D::Point2D()  
    : m_x(), m_y()  
{}
```

- Constructeur par copie non défini :
 - Appel des constructeurs par copie des attributs

```
Point2D::Point2D(const Point2D &pt)  
    : m_x(pt.m_x), m_y(pt.m_y)  
{}
```

Génération automatique du compilateur

- Le compilateur génère automatiquement si :
 - Constructeur par défaut non défini :
 - Appel des constructeurs par défaut des attributs

```
Point2D::Point2D()  
: m_x(), m_y()  
{ }
```

- Constructeur par copie non défini :
 - Appel des constructeurs par copie des attributs

```
Point2D::Point2D(const Point2D &pt)  
: m_x(pt.m_x), m_y(pt.m_y)  
{ }
```

- Destructeur non défini
 - Vide

```
Point2D::~~Point2D() { }
```

Le mot-clef `default` (C++11)

- Si on ne souhaite pas définir ces méthodes :
 - On l'indique grâce au mot-clef `default`

```
class Point2D
{
public:
    // On laisse le compilateur définir :
    // - Le constructeur par défaut
    Point2D() = default;

    // - Le constructeur par copie
    Point2D(const Point2D &) = default;

    // - Le destructeur
    ~Point2D() = default;
};
```

Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const  
{  
    return this->m_x;  
}
```

Pointeur vers l'objet courant

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const
{
    return this->m_x;
}
```

Pointeur vers l'objet courant

```
float Point2D::getCoordX() const
{
    return m_x;
}
```

Équivalent car this implicite

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const
{
    return this->m_x;
}
```

```
float Point2D::getCoordX() const
{
    return m_x;
}
```

Pointeur vers l'objet courant

Équivalent car this implicite

- Autres exemples :

Différencier argument et membre

```
void Point2D::setCoordX(const float m_x)
{
    this->m_x = m_x;
}
```

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const
{
    return this->m_x;
}
```

```
float Point2D::getCoordX() const
{
    return m_x;
}
```

Pointeur vers l'objet courant

Équivalent car this implicite

- Autres exemples :

Différencier argument et membre

```
void Point2D::setCoordX(const float m_x)
{
    this->m_x = m_x;
}
```

À éviter !

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const
{
    return this->m_x;
}
```

```
float Point2D::getCoordX() const
{
    return m_x;
}
```

Pointeur vers l'objet courant

Équivalent car this implicite

- Autres exemples :

Différencier argument et membre

```
void Point2D::setCoordX(const float m_x)
{
    this->m_x = m_x;
}
```

À éviter !

Récupérer le pointeur de l'objet

```
Point2D *Point2D::getPointeur() const
{
    return this;
}
```

Après faut trouver l'utilité...

Le pointeur this



- Représente l'adresse de l'objet courant :
 - À l'intérieur d'une de ses méthodes

```
float Point2D::getCoordX() const
{
    return this->m_x;
}
```

```
float Point2D::getCoordX() const
{
    return m_x;
}
```

Pointeur vers l'objet courant

Équivalent car this implicite

- Autres exemples :

Différencier argument et membre

```
void Point2D::setCoordX(const float m_x)
{
    this->m_x = m_x;
}
```

À éviter !

Récupérer le pointeur de l'objet

```
Point2D *Point2D::getPointeur() const
{
    return this;
}
```

Après faut trouver l'utilité...

- On va maintenant voir de vrais exemples...

Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Surcharge d'opérateurs

1 + 1 = 10

- Redéfinition du comportement des opérateurs
- Pour faciliter le développement

Surcharge d'opérateurs

1 + 1 = 10

- Redéfinition du comportement des opérateurs
- Pour faciliter le développement
 - Exemple avec `std::string` (et oui, c'est une classe !)

```
std::string str1 = "Conca";
```

```
std::string str2 = "ténation";
```

```
std::string str = str1 + str2; // Contient "Concaténation"
```


Surcharge d'opérateurs

$1 + 1 = 10$

- Redéfinition du comportement des opérateurs
- Pour faciliter le développement
 - Exemple avec `std::string` (et oui, c'est une classe !)

```
std::string str1 = "Conca";
```

```
std::string str2 = "ténation";
```

```
std::string str = str1 + str2; // Contient "Concaténation"
```

Surcharge de l'opérateur +

Surcharge d'opérateurs

$1 + 1 = 10$

- Redéfinition du comportement des opérateurs
- Pour faciliter le développement
 - Exemple avec `std::string` (et oui, c'est une classe !)

```
std::string str1 = "Conca";
```

```
std::string str2 = "ténation";
```

```
std::string str = str1 + str2; // Contient "Concaténation"
```

Surcharge de l'opérateur +

```
char troisiemeChar = str[2]; // Contient "n"
```

Surcharge de l'opérateur []

Opérateur d'affectation =

- Copie d'un objet
 - Si déjà affecté
 - Sinon appel au constructeur par copie

```
Point2D autrePoint2;  
autrePoint2 = point;
```

≠

```
Point2D autrePoint2 = point;
```

Opérateur d'affectation =

- Copie d'un objet

- Si déjà affecté

```
Point2D autrePoint2;  
autrePoint2 = point;
```

≠

- Sinon appel au constructeur par copie

```
Point2D autrePoint2 = point;
```

```
Point2D &operator=(const Point2D &pt);
```

Opérateur d'affectation =

- Copie d'un objet

- Si déjà affecté

```
Point2D autrePoint2;  
autrePoint2 = point;
```

≠

- Sinon appel au constructeur par copie

```
Point2D autrePoint2 = point;
```

```
Point2D &operator=(const Point2D &pt);
```



Référence car associativité à droite, sinon $a = b = c$ ne fonctionne pas

Opérateur d'affectation =

- Copie d'un objet

- Si déjà affecté

- Sinon appel au constructeur par copie

```
Point2D autrePoint2;  
autrePoint2 = point;
```

≠

```
Point2D autrePoint2 = point;
```

```
Point2D &operator=(const Point2D &pt);
```

Référence car associativité à droite, sinon $a = b = c$ ne fonctionne pas

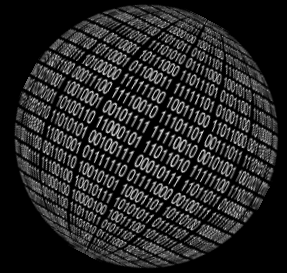
```
Point2D &Point2D::operator=(const Point2D &pt)  
{  
    if (this != &pt)  
    {  
        m_x = pt.m_x;  
        m_y = pt.m_y;  
    }  
    return *this;  
}
```

Pour éviter l'auto-copie

On retourne l'objet courant

- Même principe pour $+=$ $-=$ $*=$ *etc.*

Opérateurs binaires



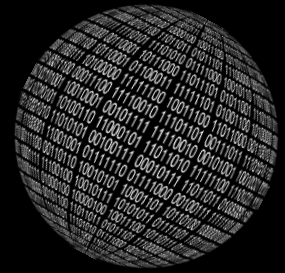
- Arithmétiques : + - * /

```
Point2D Point2D::operator+(const Point2D &p) const
{
    return Point2D( m_x + p.m_x,
                    m_y + p.m_y );
}
```

Ne modifie pas l'objet courant

Nouvel objet égal à (*this + p)

Opérateurs binaires



- Arithmétiques : + - * /

```
Point2D Point2D::operator+(const Point2D &p) const
{
    return Point2D( m_x + p.m_x,
                    m_y + p.m_y );
}
```

Ne modifie pas l'objet courant

Nouvel objet égal à (*this + p)

- Comparaisons : == /= < <= *etc.* (même principe)

```
bool Point2D::operator==(const Point2D &p) const
{
    return ( (m_x == p.m_x)
            && (m_y == p.m_y) );
}
```

Retourne un booléen

Et plein d'autres...

- Incrémentation/décrémentation : ++ --
 - new/delete
 - Indirection : *
 - Décalage de bits : << >>
 - Les flux : << >> (std::cout...)
 - *Etc.*
-
- À expérimenter en TPs !
 - Et on va y revenir un peu plus tard...



Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- **Membres statiques**
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Attribut statique

- Partagé par tous les objets de la classe
 - Existe même si aucune instance n'a été créée
 - Initialisation explicite en dehors de la classe (même si privé)



Attribut statique



- Partagé par tous les objets de la classe
 - Existe même si aucune instance n'a été créée
 - Initialisation explicite en dehors de la classe (même si privé)
- Exemple : comptons le nombre de points créés

```
class Point2D
{
public:
    Point2D();
    ~Point2D();

private:
    // Attributs
    float m_x;
    float m_y;

    static int s_nbPoints;
};
```

Attribut statique
pour compter

Attribut statique



- Partagé par tous les objets de la classe
 - Existe même si aucune instance n'a été créée
 - Initialisation explicite en dehors de la classe (même si privé)
- Exemple : comptons le nombre de points créés

```
class Point2D
{
public:
    Point2D();
    ~Point2D();

private:
    // Attributs
    float m_x;
    float m_y;

    static int s_nbPoints;
};
```

```
Point2D::s_nbPoints = 0;
```

Initialisation avec
NomClasse::

Attribut statique
pour compter

Attribut statique



- Partagé par tous les objets de la classe
 - Existe même si aucune instance n'a été créée
 - Initialisation explicite en dehors de la classe (même si privé)
- Exemple : comptons le nombre de points créés

```
class Point2D
{
public:
    Point2D();
    ~Point2D();

private:
    // Attributs
    float m_x;
    float m_y;

    static int s_nbPoints;
};
```

Initialisation avec
NomClasse::

Attribut statique
pour compter

```
Point2D::s_nbPoints = 0;
```

```
Point2D::Point2D()
{
    m_x = 0.f;
    m_y = 0.f;
    s_nbPoints++;
}
```

Incrémentation

```
Point2D::~~Point2D()
{
    s_nbPoints--;
}
```

Décrémentation

Méthode statique

- Comportement indépendant des objets instanciés
 - Peut accéder uniquement aux attributs statiques

Méthode statique

- Comportement indépendant des objets instanciés
 - Peut accéder uniquement aux attributs statiques
- Exemple : méthode pour afficher `s_nbPoints`

```
class Point2D
{
public:
    Point2D();
    ~Point2D();

    static void afficheNbPoints();
private:
    // Attributs
    float m_x;
    float m_y;

    static int s_nbPoints;
};
```

```
void Point2D::afficheNbPoints()
{
    std::cout << s_nbPoints;
}
```

Méthode
statique

Méthode statique

- Comportement indépendant des objets instanciés
 - Peut accéder uniquement aux attributs statiques
- Exemple : méthode pour afficher `s_nbPoints`

```
class Point2D
{
public:
    Point2D();
    ~Point2D();

    static void afficheNbPoints();
private:
    // Attributs
    float m_x;
    float m_y;

    static int s_nbPoints;
};
```

Méthode
statique

```
void Point2D::afficheNbPoints()
{
    std::cout << s_nbPoints;
}
```

Appel avec
NomClasse ::

```
Point2D::afficheNbPoints();
```

Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Mot-clef inline

- Similaire aux fonctions macros (#define) du C
 - Appel de la fonction substitué par son code

```
int main()  
{  
    int variable;  
  
    ajouterDeux(variable);  
  
    return 0;  
}
```

Compilateur

```
int main()  
{  
    int variable;  
  
    variable += 2;  
  
    return 0;  
}
```

Mot-clef inline

- Similaire aux fonctions macros (#define) du C
 - Appel de la fonction substitué par son code

```
int main()  
{  
    int variable;  
  
    ajouterDeux(variable);  
  
    return 0;  
}
```

Compilateur

```
int main()  
{  
    int variable;  
  
    variable += 2;  
  
    return 0;  
}
```

- Mieux que les macros !
 - Types d'arguments et de retour vérifiés
 - Conversions effectuées

STOP AUX MACROS !



Fonctions inline



- Déclaration et définition dans le header (.hpp)
- Mot-clef `inline`

Au moins un
des deux

```
inline float carre(const float f);

int main()
{
    std::cout << carre(79.f);

    return 0;
}

inline float carre(const float f)
{
    return f * f;
}
```

Fonctions inline



- Déclaration et définition dans le header (.hpp)
- Mot-clef `inline`

En général,
juste la définition

```
float carre(const float f);

int main()
{
    std::cout << carre(79.f);

    return 0;
}

inline float carre(const float f)
{
    return f * f;
}
```

Méthodes inline

- Fonctionnement similaire mais membre d'une classe
- Deux façons de définir :

Déclaration et définition séparées

```
class Point2D
{
public:
    float getCoordX() const;

private:
    // Attributs
    float m_x;
    float m_y;
};

inline
float Point2D::getCoordX() const
{
    return m_x;
}
```

Mais dans le header !

Méthodes inline

- Fonctionnement similaire mais membre d'une classe
- Deux façons de définir :

Déclaration et définition séparées

```
class Point2D
{
public:
    float getCoordX() const;

private:
    // Attributs
    float m_x;
    float m_y;
};

inline
float Point2D::getCoordX() const
{
    return m_x;
}
```

Mais dans le header !

Tout à l'intérieur de la classe

```
class Point2D
{
public:
    float getCoordX() const
    {
        return m_x;
    }

private:
    // Attributs
    float m_x;
    float m_y;
};
```


Bilan inline

- Avantages :
 - Exécution plus rapide : évite l'appel de fonction
 - Meilleure lisibilité (si bien fait)
- Inconvénients :
 - Rend l'exécutable plus lourd (code copié à plusieurs endroits)
 - Peut finalement ralentir l'exécution



Bilan inline

- Avantages :
 - Exécution plus rapide : évite l'appel de fonction
 - Meilleure lisibilité (si bien fait)
- Inconvénients :
 - Rend l'exécutable plus lourd (code copié à plusieurs endroits)
 - Peut finalement ralentir l'exécution
- Quand les utiliser ?
 - Pas vraiment de règle...
 - Le compilateur joue beaucoup !
 - En général : code court, répété souvent
 - Typiquement un getter, un opérateur...



Notions

- Première classe en C++
- Constructeurs et destructeurs
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs


Fonctions amies

- Permet l'accès à des attributs privés de l'extérieur
 - Mot-clef `friend`
- Les fonctions amies sont déclarées dans la classe
 - Ce n'est pas une méthode de la classe!

```
class Personne
{
public:
    Personne()
    {
        m_age = 0;
    }


    friend void changerAge(const int a);

private:
    int m_age;
};
```



```
void changerAge(const int a)
{
    Personne mrX;

    mrX.m_age = a;
}
```



Possible car fonctions amies




Classes amies

- Même principe :
 - Pour toute une classe

```
class UneClasse
{
public:
    UneClasse();

    friend class AutreClasse;

private:
    int m_attribut;
};
```



AutreClasse
a accès à
m_attribut


Classes amies

- Même principe :
 - Pour toute une classe
- Mutuellement amies ?

```
class UneClasse
{
public:
    UneClasse();

    friend class AutreClasse;

private:
    int m_attribut;
};
```



AutreClasse
a accès à
m_attribut


Classes amies

- Même principe :
 - Pour toute une classe
- Mutuellement amies
 - Prédéclaration


```
class UneClasse
{
public:
    UneClasse();

    friend class AutreClasse;

private:
    int m_attribut;
};
```



AutreClasse
a accès à
m_attribut



```
class AutreClasse;

class UneClasse
{
    // [...]
    friend class AutreClasse;
};

class AutreClasse
{
    // [...]
    friend class UneClasse;
};
```

Bilan amitié

- L'amitié n'est pas transitive :
 - « Les amis de mes amis ne sont pas mes amis »
- L'amitié n'est pas héritée :
 - « Mes amis ne sont pas les amis de mes enfants »
- Traduit souvent un défaut de conception du programme
 - Mais parfois indispensable
 - Et bien pratique !



Notions

- Première classe en C++
- Constructeur et destructeur
- Le pointeur `this`
- Surcharge d'opérateurs
- Membres statiques
- Fonctions inline
- L'amitié
- Retour sur la surcharge d'opérateurs

Opérateurs de flux << >>

- Pour utiliser les flux d'entrées/sorties
- Types de flux :
 - Entrée (std::cin) : std::istream

```
std::istream &operator>>(std::istream &is, UneClasse &c);
```

- Sortie (std::cout, std::cerr) : std::ostream

```
std::ostream &operator<<(std::ostream &os, const UneClasse &c);
```

Exemple avec <<

```
class Point2D
{
public:
    friend std::ostream &operator<<(std::ostream &os, const Point2D &pt);

private:
    // Attributs
    float m_x;
    float m_y;
};
```

Pour accéder aux attributs privés : pas obligatoire si getters

```
std::ostream &operator<<(std::ostream &os, const Point2D &pt)
{
    os << "[" << pt.m_x << "," << pt.m_y << "]";
    return os;
}
```

```
Point p(79.f, 17.f);
```

```
std::cout << p << std::endl;
std::cerr << p << std::endl;
```

→ « [79,17] »

Opérateurs binaires (encore ?)

- Comment additionner un point 2D avec un float ?

```
Point2D point(79.f, 17.f);
```

```
float flt = 7.f;
```

```
Point2D resultat = point + flt;
```


```
std::cout << resultat << std::endl;
```

→ « [86,24] »

Opérateurs binaires (encore ?)

- Comment additionner un point 2D avec un float ?

```
Point2D point(79.f, 17.f);  
  
float flt = 7.f;  
  
Point2D resultat = point + flt;  
  
std::cout << resultat << std::endl;
```




« [86,24] »

- Surcharge !

```
inline Point2D Point2D::operator+(const Point2D &p) const  
{  
    return Point2D( m_x + p.m_x, m_y + p.m_y );  
}
```

```
inline Point2D Point2D::operator+(const float f) const  
{  
    return Point2D( m_x + f, m_y + f );  
}
```

Remarquez
inline




Surcharge



Opérateurs binaires (encore ?)

- Ok mais pour faire « float + Point2D » ?

```
Point2D point(79.f, 17.f);  
  
float flt = 7.f;  
  
Point2D resultat = flt + point;  
  
std::cout << resultat << std::endl;
```



« [86,24] »

Opérateurs binaires (encore ?)

- Ok mais pour faire « float + Point2D » ?

```
Point2D point(79.f, 17.f);  
  
float flt = 7.f;  
  
Point2D resultat = flt + point;  
  
std::cout << resultat << std::endl;
```

→ « [86,24] »

- Fonction amie !

```
class Point2D  
{  
public:  
    // [...]  
    friend Point2D operator+(const float f, const Point2D &pt);  
};  
  
inline Point2D operator+(const float f, const Point2D &pt)  
{  
    return pt + f; → On réutilise l'opérateur Point2D + float  
}
```

FIN DU CM !

Exercez-vous bien en TP !

```
void apprendrePOO_CPP(const Etudiant &etudiant)
{
    while (etudiant.aPasCompris())
    {
        etudiant.coder();
    }

    std::cout << "L'étudiant " << etudiant.m_nom
               << " a assimilé le cours !" << std::endl;
}
```