

# Cours C++

Encapsulation des données, constructeurs, destructeurs, copie

Ou comment faire les choses proprement.

# Pré-requis



# Encapsulation

# Encapsulation : définition

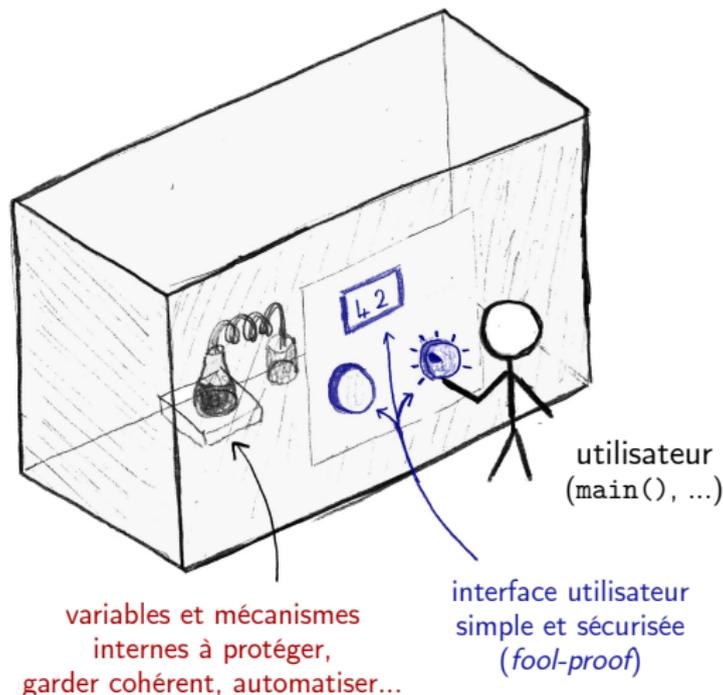
Encapsulation = interdire/cacher l'accès de certains attributs d'une classe à tout code extérieur à la classe

- ▶ il n'est plus possible d'agir **directement** sur les données d'un objet
- ▶ la modification des attributs de l'objet se fait par l'**intermédiaire de méthodes associées**
- ▶ la classe **prend responsabilité** des attributs protégés

**Syntaxe** : les mots clés sont **private**, **protected** et **public**

## Analogie classe = machine

Un objet n'est pas juste une collection d'attributs attachée à des fonctions, mais est une machine dotée d'attributs et de mécanismes internes à protéger, avec laquelle l'utilisateur interagit.



# Exemple

```
class Particule {  
private: // facultatif : private par défaut  
    // Déclaration des attributs  
    double masse_MeV;  
    int charge_u;  
    ...  
};
```

## Conséquence :

```
Particule my_kaonplus;  
my_kaonplus.masse_MeV = 493.678; // erreur de compilation
```

# Exemple

```
class Particule {  
private: // facultatif : private par défaut  
    // Déclaration des attributs  
    double masse_MeV;  
    int charge_u;  
    ...  
};
```

## Conséquence :

```
Particule my_kaonplus;  
my_kaonplus.masse_MeV = 493.678; // erreur de compilation
```

## Exemple complet (1/2)

```
class Particule {  
private:  
    // Déclaration des attributs  
    double masse_MeV;  
    int charge_u;  
  
public:  
    // Déclaration des méthodes d'accès  
    double get_masse_MeV() const;  
    double get_masse_kg() const;  
    void set_masse(double masse_MeV_);  
  
    int get_charge_u() const;  
    double get_charge_C() const;  
    void set_charge(int charge_u_);  
};
```

## Exemple complet (2/2)

```
// Définition des méthodes
double Particule::get_masse_MeV() const {
    return masse_MeV;
}

void Particule::set_masse(double masse_MeV_) {
    if (masse_MeV_ < 0)
        throw std::range_error("Masse de particule négative !");
    masse_MeV = masse_MeV_;
}

void Particule::set_charge(int charge_u_) {
    if (charge_u_ > 1 or charge_u_ < -1)
        throw std::range_error("Charge de particule doit être -1, 0, ou 1");
    charge_u = charge_u_;
}

...

Particule my_kaonplus;
my_kaonplus.set_charge(+1);
double masse = my_kaonplus.get_masse_MeV();
```

## Exemple complet (2/2)

```
// Définition des méthodes
double Particule::get_masse_MeV() const {
    return masse_MeV;
}

void Particule::set_masse(double masse_MeV_) {
    if (masse_MeV_ < 0)
        throw std::range_error("Masse de particule négative !");
    masse_MeV = masse_MeV_;
}

void Particule::set_charge(int charge_u_) {
    if (charge_u_ > 1 or charge_u_ < -1)
        throw std::range_error("Charge de particule doit être -1, 0, ou 1");
    charge_u = charge_u_;
}

...

Particule my_kaonplus;
my_kaonplus.set_charge(+1);
double masse = my_kaonplus.get_masse_MeV();
```

# Attributs privés

*Seules les méthodes* de la classe ont accès aux attributs **private/protected**<sup>†</sup> (à l'exception des fonctions amies).

<sup>†</sup> Les méthodes des classes filles pourront aussi y accéder si **protected**, et ne pourront pas si **private**. Il n'y a pas forcément raison d'avoir confiance envers les classes filles.

## Exemple (incomplet) : Matrice

Il faut interdire l'accès de data à l'utilisateur.

```
class Matrice {
public:
    ...
    double& operator[](int row, int col); // Accès à un élément de la matrice

private:
    double* data;
    unsigned int cols, rows;
};

double& Matrice::operator[](int row, int col) {
    if (row < 0 or row >= rows or col < 0 or col >= cols)
        throw std::out_of_range();
    return data[ row*cols + col ]; // renvoie une référence
}

mat1.accès[1, 2] = 4.2; // modification via référence
```

## Exemple (incomplet) : Matrice

Il faut interdire l'accès de data à l'utilisateur.

```
class Matrice {
public:
    ...
    double& operator[](int row, int col); // Accès à un élément de la matrice

private:
    double* data;
    unsigned int cols, rows;
};

double& Matrice::operator[](int row, int col) {
    if (row < 0 or row >= rows or col < 0 or col >= cols)
        throw std::out_of_range();
    return data[ row*cols + col ]; // renvoie une référence
}

mat1.accès[1, 2] = 4.2; // modification via référence
```

## Exemple (incomplet) : Matrice

Il faut interdire l'accès de data à l'utilisateur.

```
class Matrice {
public:
    ...
    double& operator[](int row, int col); // Accès à un élément de la matrice

private:
    double* data;
    unsigned int cols, rows;
};

double& Matrice::operator[](int row, int col) {
    if (row < 0 or row >= rows or col < 0 or col >= cols)
        throw std::out_of_range();
    return data[ row*cols + col ]; // renvoie une référence
}

mat1.accès[1, 2] = 4.2; // modification via référence
```

# Intérêts de l'encapsulation

- ▶ Lors de la programmation et surtout de la réutilisation d'un objet dans un autre programme, l'encapsulation empêche la modification non voulue des données membres
- ▶ L'objet, vu de l'extérieur, n'est caractérisé que par son *interface*<sup>†</sup>:
  1. pas d'ambiguïté pour l'utilisateur (*ait-je le droit de modifier cet attribut ? est-ce que faire ceci peut faire planter le programme ?*); plus simple; moins d'erreurs
  2. la maintenance et l'amélioration de l'objet sont grandement facilitées; tant que la finalité des méthodes et attributs de l'interface publique ne change pas, on peut modifier les rouages sans perturber pour autant les utilisateurs de la classe
  3. la classe est facilement réutilisable (cf. héritage)

<sup>†</sup> c'est à dire l'ensemble des attributs et méthodes effectivement utiles pour l'utilisateur

# Intérêts de l'encapsulation

- ▶ Lors de la programmation et surtout de la réutilisation d'un objet dans un autre programme, l'encapsulation empêche la modification non voulue des données membres
- ▶ L'objet, vu de l'extérieur, n'est caractérisé que par son *interface*<sup>†</sup>:
  1. pas d'ambiguïté pour l'utilisateur (*ait-je le droit de modifier cet attribut ? est-ce que faire ceci peut faire planter le programme ?*); plus simple; moins d'erreurs
  2. la maintenance et l'amélioration de l'objet sont grandement facilitées; tant que la finalité des méthodes et attributs de l'interface publique ne change pas, on peut modifier les rouages sans perturber pour autant les utilisateurs de la classe
  3. la classe est facilement réutilisable (cf. héritage)

<sup>†</sup> c'est à dire l'ensemble des attributs et méthodes effectivement utiles pour l'utilisateur

# Intérêts de l'encapsulation

- ▶ Lors de la programmation et surtout de la réutilisation d'un objet dans un autre programme, l'encapsulation empêche la modification non voulue des données membres
- ▶ L'objet, vu de l'extérieur, **n'est caractérisé que par son *interface*<sup>†</sup>**:
  1. pas d'ambiguïté pour l'utilisateur (*ait-je le droit de modifier cet attribut ? est-ce que faire ceci peut faire planter le programme ?*); plus simple; moins d'erreurs
  2. la maintenance et l'amélioration de l'objet sont grandement facilitées; tant que la finalité des méthodes et attributs de l'interface publique ne change pas, on peut modifier les rouages sans perturber pour autant les utilisateurs de la classe
  3. la classe est facilement réutilisable (cf. héritage)

<sup>†</sup> c'est à dire l'ensemble des attributs et méthodes effectivement utiles pour l'utilisateur

# Intérêts de l'encapsulation

- ▶ Lors de la programmation et surtout de la réutilisation d'un objet dans un autre programme, l'encapsulation empêche la modification non voulue des données membres
- ▶ L'objet, vu de l'extérieur, **n'est caractérisé que par son *interface*<sup>†</sup>**:
  1. pas d'ambiguïté pour l'utilisateur (*ait-je le droit de modifier cet attribut ? est-ce que faire ceci peut faire planter le programme ?*); plus simple; moins d'erreurs
  2. la maintenance et l'amélioration de l'objet sont grandement facilitées; tant que la finalité des méthodes et attributs de l'interface publique ne change pas, on peut modifier les rouages sans perturber pour autant les utilisateurs de la classe
  3. la classe est facilement réutilisable (cf. héritage)

<sup>†</sup> c'est à dire l'ensemble des attributs et méthodes effectivement utiles pour l'utilisateur

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

# Intérêts de l'encapsulation

N'ayez pas confiance en l'utilisateur de la classe, même si c'est vous !

Règle assez générale :

- ▶ **private** si l'attribut doit être protégé :
  - ▶ pointeurs nus (allocation mémoire manuelle),
  - ▶ contraintes sur les valeurs,
  - ▶ garder une cohérence entre attributs, ...
- ▶ Si on envisage qu'un attribut puisse être contraint dans le futur<sup>†</sup> → **private/protected**
- ▶ Sinon, on *peut* laisser **public**, en particulier si cela évite une lourdeur syntaxique (setters & getters, ex. : `vec3`, `point...`)
- ▶ Dans le doute, préférer **private/protected**

<sup>†</sup> en particulier dans une classe fille, cf. cours sur l'héritage

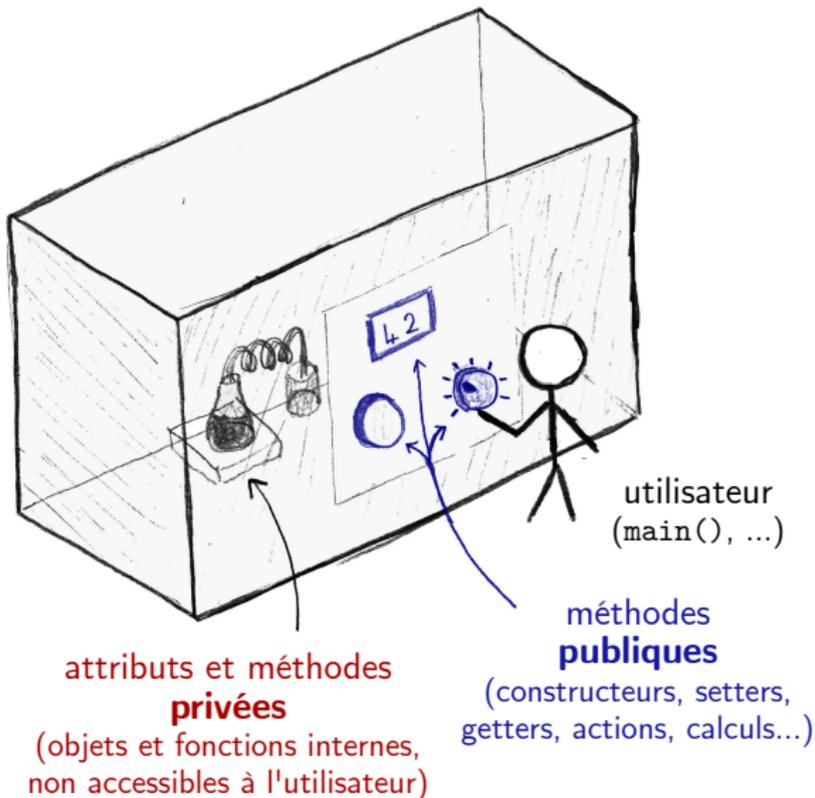
# Méthodes privées

Comme les attributs, on peut déclarer des **méthodes privées** derrière un **private**: ou **protected**:. Comme les attributs privés, elles ne sont accessibles que par les autres méthodes de la classe<sup>†</sup>. Alors que les méthodes derrière un **public**: sont accessibles partout.

Quelle utilité ? Par exemple, effectuer des opérations internes à la classe qui ne concernent pas l'utilisateur final.

<sup>†</sup> et éventuellement celles des classes fille si **protected**

# Résumé avec l'analogie classe = machine



# Constructeurs et destructeurs

# Constructeur de classe

- ▶ Le constructeur et le destructeur sont deux **méthodes i.e. fonctions membres** particulières
  - ▶ le constructeur est appelé à la création d'un objet,
  - ▶ le destructeur est appelé à la destruction d'un objet.
- ▶ constructeur  $\equiv$  **initialisation des membres**
  - ▶ allocation dynamique de mémoire,
  - ▶ ouverture de connexion, de fichier, de fenêtre...
- ▶ destructeur  $\equiv$  **ultimes opérations**
  - ▶ désallocation de la mémoire précédemment allouée
  - ▶ fermeture des ressources, déconnexion...

# Constructeur de classe

- ▶ Le constructeur est une méthode à part entière (d'où la possibilité de la surdéfinir) néanmoins
  - ▶ le constructeur doit porter **le même nom** que la classe,
  - ▶ le constructeur ne doit avoir **aucun type** de retour (pas même void).

# Déclaration de constructeurs

```
class Matrice {  
public:  
    // Constructeur sans arguments (par défaut) : matrice vide  
    Matrice();  
  
    // Surdéfinition de constructeur : matrice de cols x rows  
    Matrice(unsigned int cols, unsigned int rows, double valdef=0.);  
  
private:  
    double* data;  
    unsigned int cols, rows;  
};
```

# Définition de constructeurs

Dans `matrice.cpp` :

```
#include "matrice.h"
```

```
Matrice::Matrice(unsigned int cols_, unsigned int rows_, double valdef) {  
    cols = cols_;  
    rows = rows_;  
    data = new double[cols*rows];  
    for (unsigned int k = 0; k < cols*rows; k++)  
        data[k] = valdef;  
    std::cout << "Matrice " << cols << "x" << rows << " crée @" << data << ".\n";  
}
```

```
Matrice::Matrice() : Matrice(0,0) {}
```

Schématiquement, le compilateur va faire :

```
unsigned int cols; // allocations  
...  
cols = cols_; // initialisation  
...
```

## Définition de constructeurs avec *initializer list*

Dans `matrice.cpp` :

```
#include "matrice.h"
```

```
Matrice::Matrice(unsigned int cols_, unsigned int rows_, double valdef)
    : cols(cols_), rows(rows_), data(nullptr)
{
    data = new double[cols*rows];
    for (unsigned int k = 0; k < cols*rows; k++)
        data[k] = valdef;
    std::cout << "Matrice " << cols << "x" << rows << " crée @" << data << ".\n";
}
```

```
Matrice::Matrice() : Matrice(0,0) {}
```

Schématiquement, le compilateur va faire :

```
unsigned int cols = cols_;
// allocation et initialisation au même moment
// (nécessaire quand le membre n'a pas de constructeur par défaut)
...
```

# Utilisation

```
#include "matrice.h"

void main () {

    Matrice matrice_vider; // appel du constructeur par défaut

    Matrice matrice_pas_vider (3, 5, 1.42);
    // ou
    Matrice matrice_pas_vider = Matrice(3, 5, 1.42);

}
```

# Destructeur de classe

- ▶ Le destructeur est également une méthode, néanmoins
  - ▶ le destructeur doit porter le même nom que la classe préfixé du signe  $\sim$ ,
  - ▶ le destructeur ne possède pas d'argument; il n'est donc pas possible de surdéfinir cette méthode<sup>†</sup>.
- ▶ Lors d'allocation dynamique de mémoire, la présence d'un destructeur est obligatoire

<sup>†</sup> logique, il ne peut n'y avoir qu'une seule façon de détruire un objet

# Déclaration d'un destructeur

```
class Matrice {  
public:  
    // Constructeurs  
    Matrice();  
    Matrice(unsigned int cols, unsigned int rows, double valdef=0.);  
  
    // Destructeur  
    ~Matrice();  
  
    double* data;  
    unsigned int cols, rows;  
};
```

Dans matrice.cpp :

```
Matrice::~~Matrice() {  
    if (data != nullptr)  
        delete[] data;  
    std::cout << "Matrice @" << data << " détruite.\n";  
    data = nullptr;  
}
```

# Utilisation

```
#include "Matrice.h"

void main () {

    cout << "Début\n";

    if (true) {
        Matrice matrice_pas_vide (3, 5, 1.);
        cout << "Milieu\n";
    }

    cout << "Fin\n";
}
```

Début

Matrice 3x5 crée @0x56078abcc330.

Milieu

Matrice @0x56078abcc330 détruite.

Fin

# Utilisation

```
#include "Matrice.h"

void main () {

    cout << "Début\n";

    if (true) {
        Matrice matrice_pas_vide (3, 5, 1.);
        cout << "Milieu\n";
    }

    cout << "Fin\n";
}
```

Début

Matrice 3x5 crée @0x56078abcc330.

Milieu

Matrice @0x56078abcc330 détruite.

Fin

# Copie d'objets non triviaux

## Cas particulier du constructeur par copie

Il existe des situations autres que la déclaration d'objet où un constructeur est nécessaire,

- ▶ lorsqu'un objet est passé par valeur en argument d'une fonction :  
`int rang (Matrice mat);`
- ▶ lorsqu'un objet est renvoyé par valeur comme résultat d'une fonction :  
`Matrice matrice_alea ();`
- ▶ lors de l'initialisation d'un objet par copie d'un objet du même type :  
`Matrice mat2 = mat1;`

Par défaut, le compilateur génère un constructeur par copie trivial : les attributs sont simplement copiés

## Cas particulier du constructeur par copie

Il existe des situations autres que la déclaration d'objet où un constructeur est nécessaire,

- ▶ lorsqu'un objet est passé par valeur en argument d'une fonction :

```
int rang (Matrice mat);
```

- ▶ lorsqu'un objet est renvoyé par valeur comme résultat d'une fonction :

```
Matrice matrice_alea ();
```

- ▶ lors de l'initialisation d'un objet par copie d'un objet du même type :

```
Matrice mat2 = mat1;
```

Par défaut, le compilateur génère un constructeur par copie trivial : les attributs sont simplement copiés

## Cas particulier du constructeur par copie

Il existe des situations autres que la déclaration d'objet où un constructeur est nécessaire,

- ▶ lorsqu'un objet est passé par valeur en argument d'une fonction :  
`int rang (Matrice mat);`
- ▶ lorsqu'un objet est renvoyé par valeur comme résultat d'une fonction :  
`Matrice matrice_alea ();`
- ▶ lors de l'initialisation d'un objet par copie d'un objet du même type :  
`Matrice mat2 = mat1;`

Par défaut, le compilateur génère un constructeur par copie trivial : les attributs sont simplement copiés

## Cas particulier du constructeur par copie

Il existe des situations autres que la déclaration d'objet où un constructeur est nécessaire,

- ▶ lorsqu'un objet est passé par valeur en argument d'une fonction :  
`int rang (Matrice mat);`
- ▶ lorsqu'un objet est renvoyé par valeur comme résultat d'une fonction :  
`Matrice matrice_alea ();`
- ▶ lors de l'initialisation d'un objet par copie d'un objet du même type :  
`Matrice mat2 = mat1;`

Par défaut, le compilateur génère un constructeur par copie trivial : les attributs sont simplement copiés

## Ça va mal se passer...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

Matrice 3x5 crée @0x56078abcc330.

Avant

Matrice @0x56078abcc330 détruite.

Après

Matrice @0x56078abcc330 détruite.

## Ça va mal se passer...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

Matrice 3x5 crée @0x56078abcc330.

Avant

Matrice @0x56078abcc330 détruite.

Après

Matrice @0x56078abcc330 détruite.

## Ça va mal se passer...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

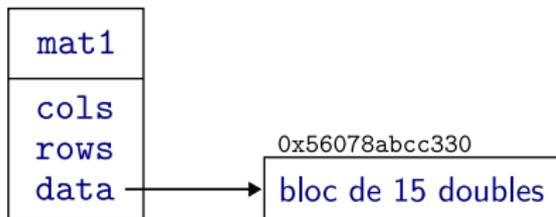
Matrice 3x5 crée @0x56078abcc330.

Avant

Matrice @0x56078abcc330 détruite.

Après

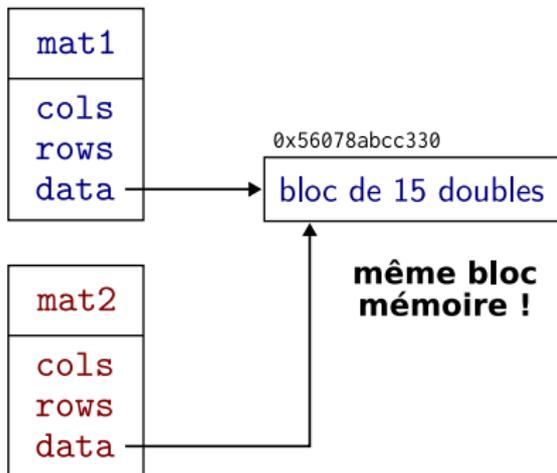
Matrice @0x56078abcc330 détruite.



## Ça va mal se passer...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

copie des  
attributs



Matrice 3x5 crée @0x56078abcc330.  
Avant  
Matrice @0x56078abcc330 détruite.  
Après  
Matrice @0x56078abcc330 détruite.

↳ Copie superficielle  
= *shallow copy*

# Ça se passe mal

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

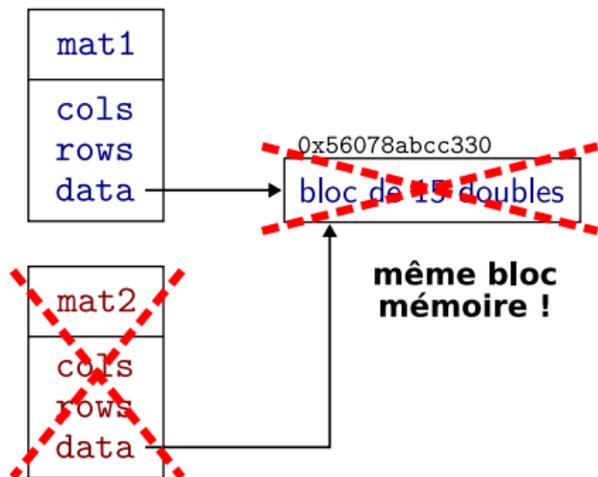
Matrice 3x5 crée @0x56078abcc330.

Avant

Matrice @0x56078abcc330 détruite.

Après

Matrice @0x56078abcc330 détruite.



## Ça se passe mal

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

Matrice 3x5 crée @0x56078abcc330.

Avant

Matrice @0x56078abcc330 détruite.

Après

Matrice @0x56078abcc330 détruite.



## Ce que l'on veut...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

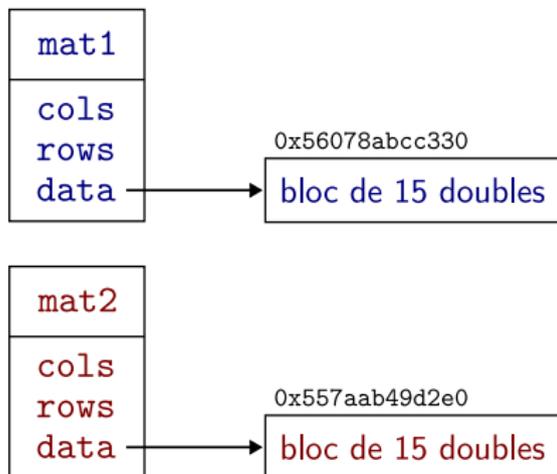
Matrice 3x5 crée @0x56078abcc330.

Avant

*Copie...*

Après

Matrice @0x56078abcc330 détruite.



↳ Copie profonde  
= *deep copy*

## Ça se passe mieux...

```
{  
  Matrice mat1 (3, 5)  
  
  cout << "Avant\n";  
  {  
    Matrice mat2 = mat1;  
    // modification de mat2...  
  }  
  cout << "Après\n";  
  
  // utilisation de mat1...  
}
```

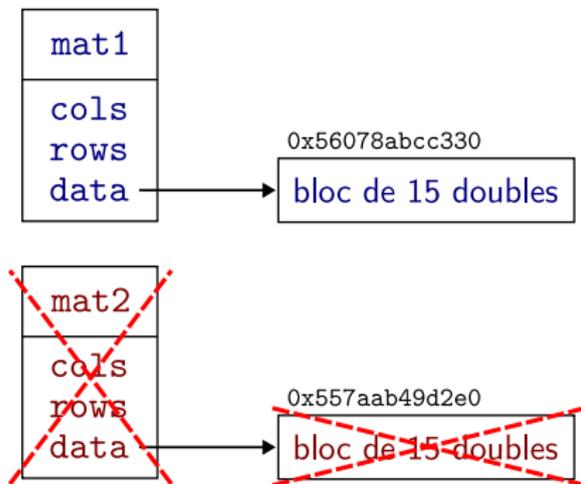
Matrice 3x5 crée @0x56078abcc330.

Avant

*Copie...*

Après

Matrice @0x56078abcc330 détruite.



## Solution : Constructeur par copie

- ▶ Nécessaire pour effectuer une copie *profonde* (copie contenu mémoire allouée dynamiquement...)

- ▶ Déclaration :

```
Matrice (const Matrice&);
```

- ▶ Définition :

```
Matrice::Matrice(const Matrice& other)
    : cols(other.cols), rows(other.rows), data(nullptr)
{
    data = new double[cols*rows];
    for (unsigned int k = 0; k < cols*rows; k++)
        data[k] = other.data[k];
}
```

- ▶ Utilisation : `Matrice mat2(mat1);` ou `Matrice mat2 = mat1;`
- ▶ Voir exemple dans `/public/mphyo/M1-C++/TestCopyConstruct`
- ▶ Alternativement, on peut *interdire la copie* :

```
Matrice (const Matrice&) = delete;
```

# Affectation & méthodes spéciales

# Méthodes spéciales

Pour l'instant, vous connaissez :

- ▶ les constructeurs `Classe::Classe(a,b,c,...)`
- ▶ le constructeur par défaut `Classe::Classe()`
- ▶ le constructeur par copie `Classe::Classe(const Classe&)` qui sert à créer un nouvel objet à partir d'un objet existant :  
`Classe b = a;`
- ▶ le destructeur `Classe::~Classe()`

On peut aussi vouloir assigner un objet à un autre :

```
Classe b;
```

```
...
```

```
b = a;
```

→ Assignation par copie

# Méthodes spéciales

Pour l'instant, vous connaissez :

- ▶ les constructeurs `Classe::Classe(a,b,c,...)`
- ▶ le constructeur par défaut `Classe::Classe()`
- ▶ le constructeur par copie `Classe::Classe(const Classe&)` qui sert à créer un nouvel objet à partir d'un objet existant :  
`Classe b = a;`
- ▶ le destructeur `Classe::~Classe()`

On peut aussi vouloir assigner un objet à un autre :

```
Classe b;
```

```
...
```

```
b = a;
```

→ Assignation par copie

# Opérateur d'affectation =

- ▶ L'opérateur d'affectation = permet d'affecter une nouvelle valeur à un objet **déjà existant** :

```
Matrice mat1(...);  
Matrice mat2(...);  
mat2 = mat1;
```

- ▶ Sa surcharge se fait comme pour n'importe quel opérateur

```
// Déclaration  
class Matrice {  
    ...  
    Matrice& operator= (const Matrice& matrice_à_copier);  
};
```

## Option n°1

Parfois, ça se sert à rien, ou ça n'a pas de sens (en particulier lorsque l'objet n'a pas à être copié). On empêche le compilateur de générer des définitions par défaut :

```
class Classe {  
    ...  
    Classe (const Classe&) = delete; // construction par copie  
    Classe& operator= (const Classe&) = delete; // assignation  
};
```

## Option n°2

Si la classe ne contient aucun attribut qui nécessite un traitement spécial ( $\neq$  allocation dynamique), on laisse le compilateur copier les attributs de façon naïve :

```
class Classe {  
    ...  
    Classe (const Classe&) = default; // construction par copie  
    Classe& operator= (const Classe&) = default; // assignation  
};
```

## Option n°3

Sinon, pour assigner `obj_b` à `obj_a`, il faut

1. détruire ce qui appartenait à `obj_a` (désallocation...)
2. re-contruire `obj_a` pour accepter le contenu de `obj_b` (allocation...)
3. copier (en profondeur) le contenu de `obj_b`

C'est pénible et ça demande de dupliquer du code qui est déjà dans le constructeur par copie, on peut faire des erreurs... Idiomme *copy-and-swap* :

1. copie temporaire de `obj_b` :  
Classe `obj_temp (obj_b)`
2. intervertir les attributs de `obj_a` ↔ `obj_temp`
3. le compilateur va détruire `obj_temp`, et donc en fait l'ancien contenu de `obj_a`

## Option n°3

Sinon, pour assigner `obj_b` à `obj_a`, il faut

1. détruire ce qui appartenait à `obj_a` (désallocation...)
2. re-contruire `obj_a` pour accepter le contenu de `obj_b` (allocation...)
3. copier (en profondeur) le contenu de `obj_b`

C'est pénible et ça demande de dupliquer du code qui est déjà dans le constructeur par copie, on peut faire des erreurs... Idiomme *copy-and-swap* :

1. copie temporaire de `obj_b` :  
Classe `obj_temp` (`obj_b`)
2. intervertir les attributs de `obj_a` ↔ `obj_temp`
3. le compilateur va détruire `obj_temp`, et donc en fait l'ancien contenu de `obj_a`

## Option n°3

Sinon, pour assigner `obj_b` à `obj_a`, il faut

1. détruire ce qui appartenait à `obj_a` (désallocation...)
2. re-contruire `obj_a` pour accepter le contenu de `obj_b` (allocation...)
3. copier (en profondeur) le contenu de `obj_b`

C'est pénible et ça demande de dupliquer du code qui est déjà dans le constructeur par copie, on peut faire des erreurs... Idiomme *copy-and-swap* :

1. copie temporaire de `obj_b` :  
Classe `obj_temp` (`obj_b`)
2. intervertir les attributs de `obj_a` ↔ `obj_temp`
3. le compilateur va détruire `obj_temp`, et donc en fait l'ancien contenu de `obj_a`

## Option n°3

Sinon, pour assigner `obj_b` à `obj_a`, il faut

1. détruire ce qui appartenait à `obj_a` (désallocation...)
2. re-contruire `obj_a` pour accepter le contenu de `obj_b` (allocation...)
3. copier (en profondeur) le contenu de `obj_b`

C'est pénible et ça demande de dupliquer du code qui est déjà dans le constructeur par copie, on peut faire des erreurs... Idiomme *copy-and-swap* :

1. copie temporaire de `obj_b` :  
Classe `obj_temp` (`obj_b`)
2. intervertir les attributs de `obj_a` ↔ `obj_temp`
3. le compilateur va détruire `obj_temp`, et donc en fait l'ancien contenu de `obj_a`

# Copy-and-swap en pratique

```
class Matrice {
    ...
    ~Matrice();
    Matrice (const Matrice& matrice_à_copier);
    Matrice& operator= (const Matrice& matrice_à_copier);
};

// définition du constructeur par copie
Matrice::Matrice (const Matrice& matrice_à_copier) { ... }

// définition de l'assignation par copie
Matrice& Matrice::operator= (const Matrice& matrice_à_copier) {
    Matrice temp (matrice_à_copier); // <- copie
    // échange des attributs
    std::swap(cols, temp.cols)
    std::swap(rows, temp.rows)
    std::swap(data, temp.data)
    return *this;
    // <- destruction de temp automatique
}
```

# Copy-and-swap en pratique (encore plus concis)

```
class Matrice {
    ...
    ~Matrice();
    Matrice (const Matrice& matrice_à_copier);
    Matrice& operator= (const Matrice& matrice_à_copier);
};

// définition du constructeur par copie
Matrice::Matrice (const Matrice& matrice_à_copier) { ... }

// définition de l'assignation par copie
Matrice& Matrice::operator= (Matrice temp) {
    // ^- copie

    // échange des attributs
    std::swap(cols, temp.cols)
    std::swap(rows, temp.rows)
    std::swap(data, temp.data)
    return *this;
    // <- destruction de temp automatique
}
```

# Copy-and-swap en pratique

```
Matrice mat1 = Matrice(3, 5);
mat1[2,2] = 4.2;
cout << "-----\n";
{
    Matrice mat2 = Matrice(1, 1);
    mat2 = mat1; // assignation par copie
    cout << "mat2[2,2] = " << mat2[2,2] << endl;
}
cout << "-----\n";
cout << "mat1[2,2] = " << mat1[2,2] << endl;

Constructeur matrice 3x5 @0x5569ba101eb0.
-----
Constructeur matrice 1x1 @0x5569ba102340.
Constructeur copie matrice 3x5 @0x5569ba102360.
Destructeur @0x5569ba102340.
mat2[2,2] = 4.2
Destructeur @0x5569ba102360.
-----
mat1[2,2] = 4.2
Destructeur @0x5569ba101eb0.
```

# Copy-and-swap en pratique

```
Matrice mat1 = Matrice(3, 5);
mat1[2,2] = 4.2;
cout << "-----\n";
{
    Matrice mat2 = Matrice(1, 1);
    mat2 = mat1; // assignation par copie
    cout << "mat2[2,2] = " << mat2[2,2] << endl;
}
cout << "-----\n";
cout << "mat1[2,2] = " << mat1[2,2] << endl;
    Constructeur matrice 3x5 @0x5569ba101eb0.
    -----
    Constructeur matrice 1x1 @0x5569ba102340.
    Constructeur copie matrice 3x5 @0x5569ba102360.
    Destructeur @0x5569ba102340.
    mat2[2,2] = 4.2
    Destructeur @0x5569ba102360.
    -----
    mat1[2,2] = 4.2
    Destructeur @0x5569ba101eb0.
```

## [À éviter] On pourrait aussi faire...

```
class Matrice {
public:
    ... // déclaration (rappel)
    Matrice& operator= (const Matrice& matrice_à_copier);
private:
    double* data;
    unsigned int cols, rows;
};

// définition de l'assignation par copie
Matrice& Matrice::operator= (const Matrice& matrice_à_copier) {
    if (this != &matrice_à_copier) { // garde-fou
        if (data != nullptr) delete[] data; // déallocation
        cols = matrice_à_copier.cols;
        rows = matrice_à_copier.rows;
        data = new double[cols*rows]; // nouvelle allocation
        for (unsigned int k = 0; k < cols*rows; k++)
            data[k] = other.matrice_à_copier[k]; // copie
    }
    return *this;
}
```

# En résumé : deux règles suivant les cas

## *Règle des trois :*

Si l'on définit un constructeur qui ouvre une ressource, alloue de la mémoire... ou n'importe lesquelles des méthodes spéciales

- Constructeur par copie
- Destructeur
- Assignment par copie

c'est qu'il faut **toutes** les définir !<sup>†</sup>

```
class Classe {  
    Classe (...); // constructeurs  
    Classe (const Classe&); // constructeur par copie  
    Classe& operator= (const Classe&); // assignment par copie  
    ~Classe(); // destructeur  
};
```

<sup>†</sup> Définir ne veut pas dire nécessairement les implémenter, lorsque c'est pertinent on peut supprimer la copie ou l'assignment. Pourquoi cette règle ? Eh bien sinon, le compilateur générera des méthodes spéciales par défaut qui n'auront pas le bon comportement (copie superficielle).

# En résumé : deux règles suivant les cas

## *Règle des trois :*

Si l'on définit un constructeur qui ouvre une ressource, alloue de la mémoire... ou n'importe lesquelles des méthodes spéciales

- Constructeur par copie
- Destructeur
- Assignment par copie

c'est qu'il faut **toutes** les définir !<sup>†</sup>

```
class Classe {  
    Classe (...); // constructeurs  
    Classe (const Classe&); // constructeur par copie  
    Classe& operator= (const Classe&); // assignation par copie  
    ~Classe(); // destructeur  
};
```

<sup>†</sup>Définir ne veut pas dire nécessairement les implémenter, lorsque c'est pertinent on peut supprimer la copie ou l'assignation. Pourquoi cette règle ? Eh bien sinon, le compilateur génèrera des méthodes spéciales par défaut qui n'auront pas le bon comportement (copie superficielle).

## En résumé : deux règles suivant les cas

### *Règle du zéro :*

Utiliser au maximum les objets de bibliothèques existantes (std::vector, std::string) et éviter l'allocation manuelle de mémoire. On peut alors laisser le compilateur implémenter toutes les méthodes spéciales par défaut !

```
class Classe {  
    Classe (...); // constructeurs  
    Classe (const Classe&) = default;  
    Classe& operator= (const Classe&) = default;  
    // pas de destructeur  
};
```

# Projets

☛ Les notions d'encapsulation sont à maîtriser pour les projets

- ▶ Constructeurs dans tous les cas
- ▶ Attributs privés quand pertinent (protection, cohérence...)
- ▶ Règle des trois ou zéro méthodes spéciales
- ▶ Automatiser au maximum les actions effectuées sur l'objet (affichage, sauvegarde dans un fichier, calculs, modifications, interaction avec d'autres objets...) grâce à des méthodes pour en simplifier l'utilisation

Et dans d'autres langages ?

## Et dans d'autres langages ?

La notion d'objets est présente dans de nombreux langages, y compris Python, Java, OCaml... C'est surtout au niveau de la gestion de la mémoire, de la construction/destruction, de la copie... que se trouvent les différences.

En Python, la notion de constructeur existe, mais la notion de destruction ou de copie ne peut être calquée sur le C++ . En effet, le moment où l'objet est effectivement détruit est non déterministe. Pas de **const** non plus.

# Petit exemple en Python

```
class Point:
    def __init__(self, x=0, y=0): # constructeur
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def norme(self):
        return (self.x**2 + self.y**2) ** 0.5

    def afficher(self):
        print(f"({self.x}, {self.y})")

p = Point(1.5, 2.0)
import copy
p2 = copy.copy(p) # la copie doit être explicite
# si non utilisés, ces deux objets seront détruits un jour...
# mais il est possible de forcer la chose :
del p
```

Déplacer au lieu de copier

# Exemple de copie évitable

Mais on peut souvent éviter des copies inutiles :

```
Matrice f () {  
  Matrice mat (3000, 5000);  
  return mat;  
}
```

La variable locale `mat` est copiée dans un objet anonyme retourné par la fonction.

```
Matrice f () {  
  return Matrice(3000, 5000);  
}
```

Il n'y a plus de variable locale. L'objet anonyme est construit directement.

Le compilateur est souvent capable d'effectuer ces optimisations automatiquement, mais mieux vaut prendre des bonnes habitudes. Cependant, ça n'est parfois pas possible, et le compilateur peut ne pas voir certaines optimisations. On peut l'aider...

## Exemple de copie évitable

Mais on peut souvent éviter des copies inutiles :

```
Matrice f () {  
  Matrice mat (3000, 5000);  
  return mat;  
}
```

La variable locale `mat` est copiée dans un objet anonyme retourné par la fonction.

```
Matrice f () {  
  return Matrice(3000, 5000);  
}
```

Il n'y a plus de variable locale. L'objet anonyme est construit directement.

Le compilateur est souvent capable d'effectuer ces optimisations automatiquement, mais mieux vaut prendre des bonnes habitudes. Cependant, ça n'est parfois pas possible, et le compilateur peut ne pas voir certaines optimisations. On peut l'aider...

# Move semantics

Dans le C++ 11, un nouveau concept a été introduit : le déplacement d'objets (*move semantics*). L'objectif est de gagner en performances en évitant les copies inutiles. Ainsi qu'en permettant le déplacement d'un objet même lorsque la copie n'a pas de sens. Par exemple, lors d'un retour de fonction :

```
std::vector<int> ma_fonction () {  
    std::vector<int> vec;  
    ...  
    return vec;  
}
```

```
...  
std::vector<int> resultat = ma_fonction();
```

le compilateur va *déplacer* vec dans resultat, au lieu d'appeler le constructeur par copie de `std::vector<int>`.

## Move semantics

Pour cela, le C++ 11 a introduit les *rvalue references* (que l'on pourrait traduire par "référence de valeur à droite d'une égalité"), par opposition aux usuelles *lvalue references*. Une *rvalue references* d'un type `Type` se note `Type&&`.

Il existe alors deux nouvelles méthodes spéciales : le constructeur par déplacement, et l'assignation par déplacement :

```
class Classe {  
    Classe (...); // constructeurs  
    Classe (const Classe&); // constructeur par copie  
    Classe& operator= (const Classe&); // assignation par copie  
    Classe (Classe&&); // constructeur par déplacement  
    Classe& operator= (Classe&&); // assignation par déplacement  
    ~Classe(); // destructeur  
};
```

qui seront appelés à la place de celles par copie lorsque l'objet de droite est une *rvalue references*.

# Move semantics

Exemple :

```
Classe c1 (...);
```

```
Classe c2 = c2; // constructeur par copie
```

```
Classe c3 = std::move(c1); // constructeur par déplacement
```

Ici, c1 est déplacé dans c3, et non copié. Attention, après cette opération, c1 ne doit plus être utilisé !

# Move semantics

Exemple avec Matrice :

```
// Définition du constructeur par déplacement
Matrice::Matrice (Matrice&& mat_à_déplacer)
    // on "vole" les attributs de `mat_à_déplacer`
    : cols(mat_à_déplacer.cols),
      rows(mat_à_déplacer.rows),
      data(mat_à_déplacer.data)
{
    // on laisse `mat_à_déplacer` dans un état valide :
    mat_à_déplacer.cols = 0;
    mat_à_déplacer.rows = 0;
    mat_à_déplacer.data = nullptr;
}
```

Déplacer un objet, ce n'est rien d'autre que *voler ses attributs* tout en laissant derrière soi un état valide

# Move semantics

Lorsque la classe ne contient que des attributs qui n'ont pas besoin de traitement spécial (fermeture de ressource, désallocation...), on peut laisser le compilateur le définir par défaut :

```
class Classe {  
    Classe (...); // constructeurs  
    Classe (Classe&&) = default; // constructeur par déplacement  
    Classe& operator= (Classe&&) = default; // assignation par déplacement  
};
```

Le compilateur les génèrera automatiquement si il n'y a pas de destructeur de défini.

# Move semantics

Cas d'utilisation :

- ▶ On peut vouloir stocker des objets de type Classe, non copiables (par exemple représentant une ressource physique), dans un `std::vector<Classe>`. C'est possible, mais seulement si le constructeur par déplacement est défini.
- ▶ `std::sort()` passe sa vie à échanger des éléments dans un tableau pour le trier : ça sera beaucoup plus efficace si constructeur par déplacement est défini !
- ▶ Lorsque les types des attributs utilisés sont élémentaires (**int**, **double**...), ça ne sert à rien
- ▶ Ça reste une notion complexe et avancée qu'il n'est pas du tout nécessaire de maîtriser ni même de connaître