



JE TE FORCE PAS HEIN  
MAIS REGAAAAARDE  
COMMENT C'EST SI BEAU ET  
SI AMUSANT !

# Cours de C++

## Segment 2

2025-2026

1. Copie
2. Déplacement
3. L-Value et R-Value
4. Conteneurs
5. Pointeurs intelligents
6. Héritage
7. Classes polymorphes

## 1. Copie.

- a. Construction vs affectation
- b. Constructeur de copie
- c. Opérateur d'affectation par copie
- d. Implémentations par défaut

## 2. Déplacement.

## 3. L-Value et R-Value.

## 4. Conteneurs

## 5. Pointeurs intelligents

## 6. Héritage.

## 7. Classes polymorphes.

Il faut distinguer l'**instanciation** d'un objet de sa **réaffectation**, car ce ne sont pas les mêmes fonctions qui sont appelées.

Si on instancie un tout nouvel objet :

```
Value v1 { 4 };
```



Appel du  
**constructeur**

Si on modifie la valeur d'un objet qui existe déjà :

```
v1 = 3;
```



Appel de  
l'**opérateur d'affectation**

```
struct Value  
{  
    Value(int value)  
        : v { value }  
    {}  
}
```

Opérateur  
d'affectation

```
void operator=(int value)  
{  
    v = value;  
}  
  
int v = 0;  
};
```

! Un seul =

```
struct Value
{
    Value(int value)
        : v { value }
    {}

    void operator=(int value)
    {
        v = value;
    }

    int v = 0;
};
```

Quelles fonctions sont appelées par les instructions suivantes ?

```
Value v1 { 4 };
v1 = 3;
Value v2 = 3;
```

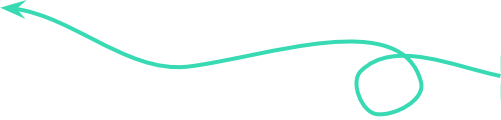
Quelles fonctions sont appelées par les instructions suivantes ?

```
struct Value
{
    Value(int value)
        : v { value }
    {}

    void operator=(int value)
    {
        v = value;
    }

    int v = 0;
};
```

```
Value v1 { 4 };
v1 = 3;
Value v2 = 3;
```



```
struct Value  
{  
    Value(int value)  
        : v { value }  
    {}  
}
```

```
void operator=(int value)  
{  
    v = value;  
}
```

```
int v = 0;  
};
```

Quelles fonctions sont appelées par les instructions suivantes ?

```
Value v1 { 4 };  
v1 = 3;  
Value v2 = 3;
```





```
struct Value
{
    Value(int value)
        : v { value }
    {}

    void operator=(int value)
    {
        v = value;
    }

    int v = 0;
};
```

Quelles fonctions sont appelées par les instructions suivantes ?

```
Value v1 { 4 };
v1 = 3;
Value v2 = 3;
```

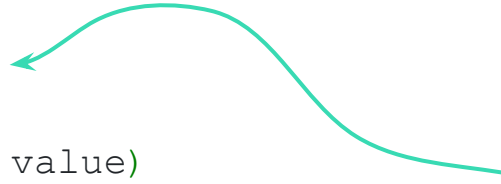
Quelles fonctions sont appelées par les instructions suivantes ?

```
struct Value
{
    Value(int value)
        : v { value }
    {}

    void operator=(int value)
    {
        v = value;
    }

    int v = 0;
};
```

```
Value v1 { 4 };
v1 = 3;
Value v2 = 3;
```



```
struct Value
{
    Value(int value)
        : v { value }
    {}

    void operator=(int value)
    {
        v = value;
    }

    int v = 0;
};
```

Quelles fonctions sont appelées par les instructions suivantes ?

```
Value v1 { 4 };
v1 = 3;
Value v2 = 3;
```

 Des questions?

Le **constructeur de copie** est le constructeur appelé lorsqu'un objet est **instancié** et initialisé à partir d'un objet du **même type**.

```
Animal medor  
// ...  
Animal medor_copy1 { medor };  
Animal medor_copy2 = medor ;
```

```
class Animal
{
public:
    Animal(const std::string& species, const std::string& name)
        : _species { species }, _name { name }
    {}
```

Constructeur de copie

```
    Animal(const Animal& other)
        : _species { other._species }
        , _name { other._name + " 2 " }
    {
        std::cout << _name << " was copied from " << other._name << std::endl;
    }

private:
    std::string _species;
    std::string _name;
};
```

```
class Animal
{
public:
    Animal(const std::string& species, const std::string& name)
        : _species { species }, _name { name }
    {}

    Animal(const Animal& other)
    {
        : _species { other._species }
        , _name { other._name + " 2 " }
    {
        std::cout << _name << " was copied from " << other._name <<
std::endl;
    }

private:
    std::string _species;
    std::string _name;
};
```

Signature

Plus génériquement :

ClassName (const ClassName&)

```
class Animal
{
public:
    Animal(const std::string& species, const std::string& name)
        : _species { species }, _name { name }
    {}

    Animal(const Animal& other)
    {
        : _species { other._species }
        , _name { other._name + " 2 " }
    }

    std::cout << _name << " was copied from " << other._name <<
std::endl;
}

private:
    std::string _species;
    std::string _name;
};
```

Les attributs sont initialisés  
dans la **liste d'initialisation**

```
class Animal
{
public:
    Animal(const std::string& species, const std::string& name)
        : _species { species }, _name { name }
    {}
```

```
    Animal(const Animal& other)
        : _species { other._species }
        , _name { other._name + " 2 " }
    {
        std::cout << _name << " was copied from " << other._name <<
std::endl;
    }
```

Les instructions additionnelles  
vont dans le **corps**

```
private:
    std::string _species;
    std::string _name;
};
```



```
class Animal
{
public:
    Animal(const std::string& species, const std::string& name)
        : _species { species }, _name { name }
    {}

    Animal(const Animal& other)
        : _species { other._species }
        , _name { other._name + " 2 " }
    {
        std::cout << _name << " was copied from " << other._name <<
std::endl;
    }

private:
    std::string _species;
    std::string _name;
};
```

 Des questions?

L'**opérateur d'affectation par copie** est appelé lorsque la valeur d'un objet est affectée à un objet **pré-existant** du **même type**.

```
Animal medor { ... };  
Animal felix { ... };  
medor = felix;
```

```
class Animal
```

```
{
```

```
public
```

```
...
```

Opérateur d'affectation par copie

```
Animal& operator=(const Animal& other)
```

```
{
```

```
    if (this != &other)
```

```
    {
```

```
        _name = other._name;
```

```
    }
```

```
    return *this;
```

```
}
```

```
...
```

```
};
```

```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }

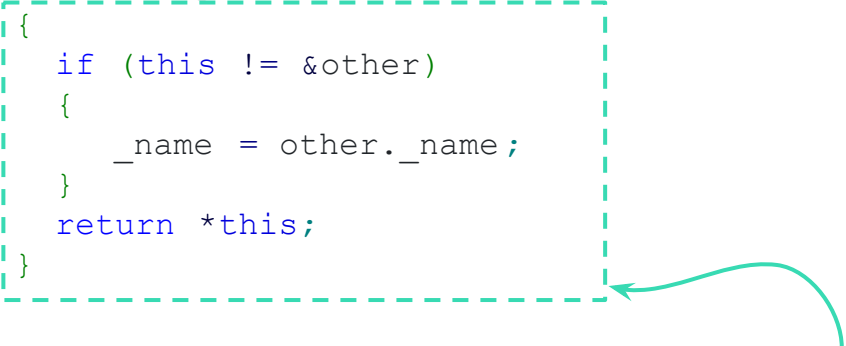
    ...
};
```

Signature

```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }

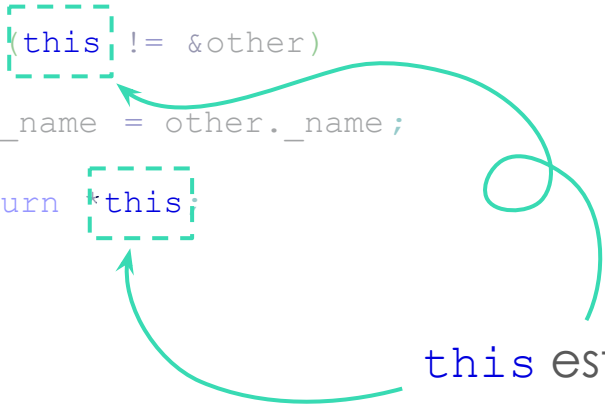
    ...
};
```



Le **corps** de la fonction contient les instructions exécutées par l'affectation

```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }
    ...
};
```




`this` est un **pointeur** permettant  
d'accéder à l'**instance courante**

```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }

    ...
};
```

 **Par convention:**  
valeur de retour  
=  
**référence** sur  
l'instance courante

Cela permet de chaîner les appels :  
`felix = medor = ginger;`

```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }
    ...
};
```



## Attention !

vérifiez toujours que  
l'**objet courant** et  
l'**objet à copier** sont  
des **instances**  
**distinctes**

Cela peut éviter des problèmes  
lorsqu'on réaffecte un objet à  
lui-même



```
class Animal
{
public:
    ...

    Animal& operator=(const Animal& other)
    {
        if (this != &other)
        {
            _name = other._name;
        }
        return *this;
    }

    ...
};
```

 Des questions?

Si vous copiez un objet sans définir la fonction appropriée (constructeur de copie ou opérateur d'affectation par copie), le **compilateur** essaie de générer une **implémentation par défaut**.

## Implémentation par défaut du constructeur de copie

```
ClassName(const ClassName& other)
    : _attr1 { other._attr1 }
    , _attr2 { other._attr2 }
    , ...
{
}
```

## Implémentation par défaut de l'opérateur d'affectation par copie

```
ClassName& operator=(const ClassName& other)
{
    if (this != &other)
    {
        _attr1 = other._attr1;
        _attr2 = other._attr2;
        ...
    }
    return *this;
}
```

⚠ Suivant les cas, il se peut que le compilateur ne génère pas d'implémentations par défaut (Voir site du cours)

1. Copie.

## 2. Déplacement.

- a. Concept
- b. Constructeur de déplacement
- c. Opérateur d'affectation par déplacement
- d. Implémentations par défaut
- e. Variables de types fondamentaux

3. L-Value et R-Value.

4. Conteneurs

5. Pointeurs intelligents.

6. Héritage.

7. Classes polymorphes.

Copier certains objets est **coûteux**.  
Même en les passant par référence, certaines copies ne  
sont pas évitées...

*Où se trouve la copie dans le code suivant ?*


```
std::string name = "Celine";  
Person      celine { name };
```

```
class Person  
{  
public:  
    Person(const std::string& name)  
        : _name { name }  
    {}  
  
private:  
    std::string _name;  
};
```

*Où se trouve la copie dans le code suivant ?*

```
std::string name = "Celine";  
Person      celine { name };
```

```
class Person  
{  
public:  
    Person(const std::string& name)  
        : _name { name }  
    {}  
  
private:  
    std::string _name;  
};
```





On aimerait bien pouvoir **déplacer** le contenu de *name*  
à l'intérieur de *celine.\_name*

```
std::string name = "Celine";  
Person celine { name };
```

```
class Person  
{  
public:  
    Person(const std::string& name)  
        : _name { name }  
    {}  
private:  
    std::string _name;  
};
```

Le déplacement consiste donc à **transférer le contenu** d'une instance A à l'intérieur d'une instance B.

Déplacer A dans B est plus intéressant que copier A dans B si :

1. vous savez que la copie est **coûteuse**
2. **vous n'utilisez plus A** dans la suite du code

# Qu'est-ce que le déplacement ?

*Objet dont on a plus besoin*

Données internes à  
l'objet  
(Usuellement sur la pile)

Données externes de l'objet  
d'origine (Usuellement, sur le tas)



Nouvel objet qu'on veut construire  
en déplaçant celui d'au dessus.

# Qu'est-ce que le déplacement ?

*Objet dont on a plus besoin*

Données internes à  
l'objet  
(Usuellement sur la pile)

Copie des  
données  
internes

Données externes de l'objet  
d'origine (Usuellement, sur le tas)

Nouvel objet qu'on veut construire  
en déplaçant celui d'au dessus.

# Qu'est-ce que le déplacement ?

*Objet dont on a plus besoin*

Données internes à  
l'objet  
(Usuellement sur la pile)

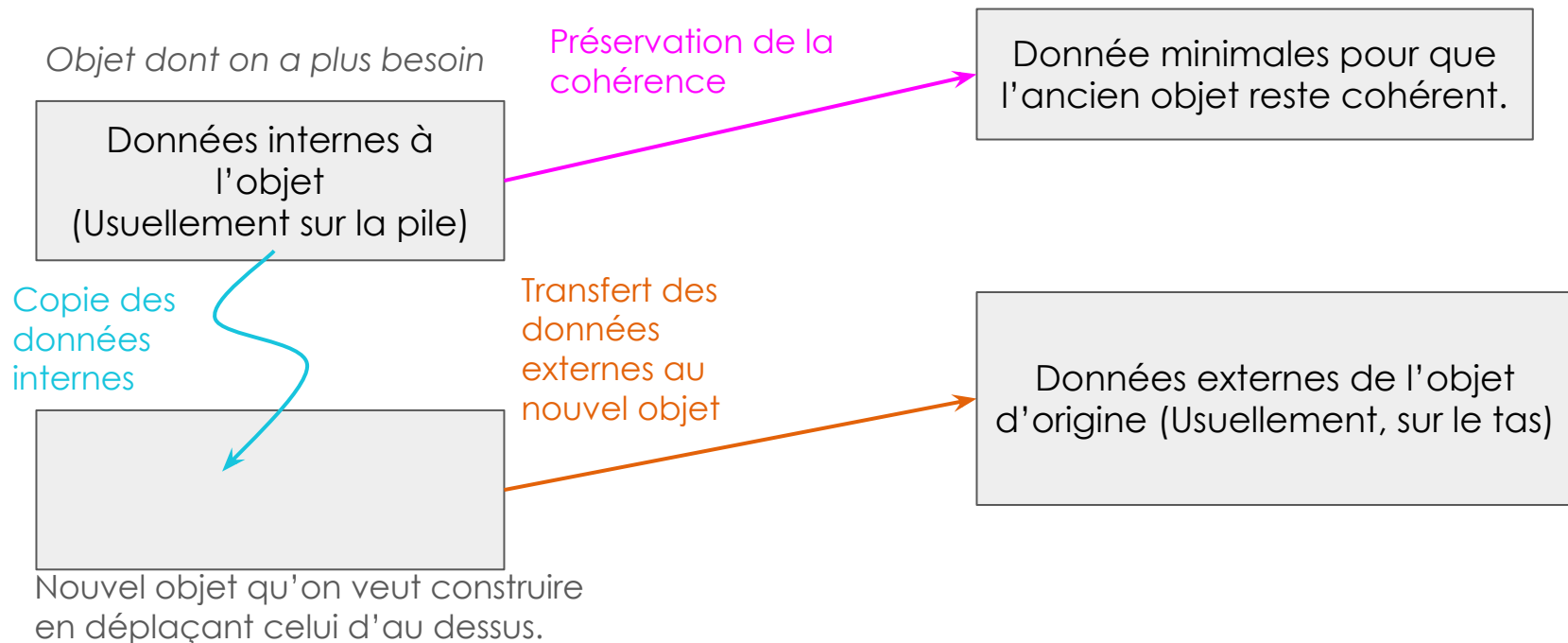
Copie des  
données  
internes

Transfert des  
données  
externes au  
nouvel objet

Données externes de l'objet  
d'origine (Usuellement, sur le tas)

Nouvel objet qu'on veut construire  
en déplaçant celui d'au dessus.

# Qu'est-ce que le déplacement ?



🤔 Des questions?

La librairie standard fournit la fonction `std::move`,  
qui permet de **transférer** le **contenu** d'un objet

```
std::string name = "Celine";  
Person celine { std::move(name) };  
  
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
  
private:  
    std::string _name;  
};
```

The diagram illustrates the use of `std::move` to transfer the content of a variable to a class constructor parameter. A dashed box highlights the `std::move(name)` expression in the `Person` object initialization. Another dashed box highlights the `std::move(name)` expression inside the `Person` constructor. A curved arrow points from the first dashed box to the second, indicating the transfer of the string's content from the `name` variable to the `_name` member variable.

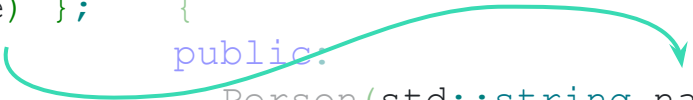
On transfère le contenu de  
name à l'intérieur du 1er paramètre  
du constructeur

La librairie standard fournit la fonction `std::move`,  
qui permet de **transférer** le **contenu** d'un objet

```
std::string name = "Celine";  
Person celine { std::move(name) };  
                ^
```

On utilise `std::move`

```
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
  
private:  
    std::string _name;  
};
```



A green curved arrow originates from the `std::move(name)` expression in the `Person` constructor call within the `celine` object initialization and points to the `std::move(name)` argument in the `Person` constructor's member initialization list.

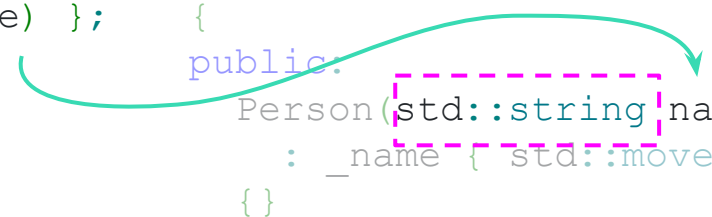


La librairie standard fournit la fonction `std::move`,  
qui permet de **transférer** le **contenu** d'un objet

```
std::string name = "Celine";  
Person celine { std::move(name) };
```

On ne passe **pas**  
le paramètre par référence,  
puisque l'on construit un nouvel objet  
à partir du contenu de l'autre

```
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
  
private:  
    std::string _name;  
};
```




La librairie standard fournit la fonction `std::move`,  
qui permet de **transférer** le **contenu** d'un objet

```
std::string name = "Celine";  
Person celine { std::move(name) };
```

On transfère à nouveau le  
contenu de `name` au constructeur  
de l'attribut `_name`

```
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
private:  
    std::string _name;  
};
```



La variable d'origine est maintenant **vide**, puisque son contenu a été déplacé ailleurs !

```
std::string name = "Celine";  
Person celine { std::move(name) };  
  
std::cout << "< " << name << " >";
```



```
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
  
private:  
    std::string _name;  
};
```

La variable d'origine est maintenant **vide**, puisque son contenu a été déplacé ailleurs !

```
std::string name = "Celine";  
Person celine { std::move(name) };  
  
std::cout << "< " << name << " >";
```

```
class Person  
{  
public:  
    Person(std::string name)  
        : _name { std::move(name) }  
    {}  
  
private:  
    std::string _name;  
};
```

 Des questions?

Lorsqu'on déplace un objet pour en **instancier** un autre du **même type**, c'est le **constructeur de déplacement** qui est appelé.

```
Animal new_medor = std::move(medor);
```

```
class Animal  
{  
public:  
    ...
```

Constructeur de déplacement

```
Animal(Animal&& other)  
    : _species { std::move(other._species) }  
    , _name { std::move(other._name) }  
    {}
```

```
    ...  
};
```

```
class Animal
{
public:
    ...

    Animal(Animal&& other)
    : _species { std::move(other._species) }
      , _name { std::move(other._name) }
    {}

    ...
};
```

Signature

Plus génériquement :

```
ClassName (ClassName&&)
```

```
class Animal
{
public:
    ...

    Animal(Animal&& other)
        : _species { std::move(other._species) }
        , _name { std::move(other._name) }
    {}

    ...
};
```

```
Animal medor { "dog", "medor" };
Animal new_medor = std::move(medor);
```

Quelles sont les valeurs de :

- *medor.\_species* ?
- *medor.\_name* ?
- *new\_medor.\_species* ?
- *new\_medor.\_name* ?



L'**opérateur d'affectation par déplacement** est appelé lorsqu'un objet est déplacé dans une instance **pré-existante** du **même type**.

```
Animal medor { ... };  
Animal felix { ... };  
medor = std::move(felix);
```

```
class Animal  
{  
public:  
...
```

Opérateur d'affectation  
par déplacement

```
Animal& operator=(Animal&& other)  
{  
    if (this != &other)  
    {  
        _name = std::move(other._name);  
    }  
    return *this;  
}  
  
...  
};
```

```
class Animal
{
public:
    ...

    Animal& operator=(Animal&& other)
    {
        if (this != &other)
        {
            _name = std::move(other._name);
        }
        return *this;
    }

    ...
};
```

Signature

```
class Animal
{
public:
    ...

    Animal& operator=(Animal&& other)
    {
        if (this != &other)
        {
            _name = std::move(other._name);
        }
        return *this;
    }

    ...
};
```

Mêmes contraintes que pour  
l'affectation par copie :

- valeur de retour = `*this`
- s'assurer que les instances  
sont bien distinctes

Comme pour les fonctions de copie, le compilateur peut générer des **implémentations par défaut** pour les fonctions de déplacement.

## Implémentation par défaut du constructeur de déplacement

```
ClassName (ClassName&& other)
: _attr1 { std::move(other._attr1) }
, _attr2 { std::move(other._attr2) }
, ...
{
}
```

## Implémentation par défaut de l'opérateur d'affectation par déplacement

```
ClassName& operator=(ClassName&& other)
{
    if (this != &other)
    {
        _attr1 = std::move(other._attr1);
        _attr2 = std::move(other._attr2);
        ...
    }
    return *this;
}
```

*Que se passe-t-il lorsque vous déplacez une variable de type fondamental dans une autre ?*

```
int a = 4;  
int b = std::move(a);
```

```
auto* ptr_1 = &a;  
auto* ptr_2 = std::move(ptr_1);
```

Lorsque vous déplacez une variable de type fondamental dans une autre, cela équivaut à **faire une copie**.

Le contenu de la **variable source** reste donc **inchangé** !

```
int a = 4;  
int b = std::move(a);
```

a vaut toujours 4,  
pas 0

```
auto* ptr_1 = &a;  
auto* ptr_2 = std::move(ptr_1);
```

ptr\_1 vaut &a et  
non pas nullptr

1. Copie
2. Déplacement
- 3. L-Value et R-Value**
  - a. Expression
  - b. Catégorisation
  - c. Overloading
4. Conteneurs
5. Pointeurs intelligents
6. Héritage
7. Classes polymorphes



Une **expression** est une combinaison d'**opérandes**, d'**opérateurs**, d'appel de fonctions, pouvant être **évaluée**.

L'**évaluation** d'une expression peut **parfois** produire une **valeur**.

Exemples :

```
(a + b) / 3  
15  
&r  
fcn(r, c + 3, t)  
a = b = c  
r == 8 || call(g) ==  
'c'  
++it
```

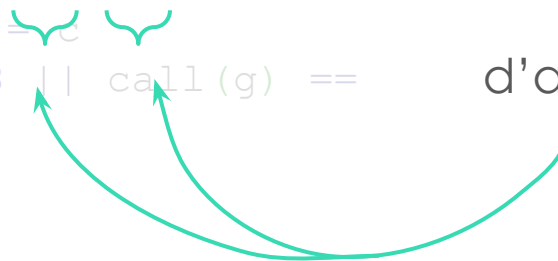
Exemples :

```
(a + b) / 3
15
&r
fcn(r, c + 3, t)
a = b = c
r == 8 || call(g) ==
'c'
++it
```

Une expression peut  
être composée de  
sous-expressions

Exemples :

```
(a + b) / 3
15
&r
fcn(r, c + 3, t)
a = b = c
r == 8 || call(g) ==
'c'
++it
```



Une expression peut  
être composée de  
sous-expressions

...

qui peuvent-elles aussi  
être constituées  
d'autres sous-expressions

Exemples :



Les expressions produisant des valeurs sont catégorisées soit en tant que **L-value**, soit en tant que **R-value**.

Les expressions produisant des valeurs sont catégorisées soit en tant que **L-value**, soit en tant que **R-value**.

- Une **L-value** est une expression dont l'évaluation renvoie une donnée ayant déjà une **adresse mémoire** (ex: variable, référence).

Les expressions produisant des valeurs sont catégorisées soit en tant que **L-value**, soit en tant que **R-value**.

- Une **L-value** est une expression dont l'évaluation renvoie une donnée ayant déjà une **adresse mémoire** (ex: variable, référence).
- Une **R-value** est une expression dont l'évaluation produit un **résultat temporaire**, qui n'a pas forcément d'emplacement mémoire associé (ex: littéral entier, retour d'une fonction par valeur).



*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

5 est un littéral entier

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

5 est un littéral entier



R-value

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

v1 est une variable

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

v1 est une variable



L-value

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector{ 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```



*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector{ 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

le résultat du calcul  
n'est pas encore  
stocké en mémoire

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector{ 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

le résultat du calcul  
n'est pas encore  
stocké en mémoire



R-value

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

on construit un tout  
nouvel objet

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

on construit un tout  
nouvel objet



R-value

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector { 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

on retourne une  
référence sur l'  
élément ajouté au  
tableau

*L-value ou R-value ?*

```
auto v1 = 5;  
auto v2 = v1;  
auto v3 = v2 + 5 - v1;  
auto v4 = std::vector{ 1, 2, 3 };  
auto v5 = v4.emplace_back(4);
```

on retourne une  
référence sur l'  
élément ajouté au  
tableau



L-value



*L-value ou R-value ?*

```
auto v1 = 5; // R
auto v2 = v1; // L
auto v3 = v2 + 5 - v1; // R
auto v4 = std::vector { 1, 2, 3 }; //R
auto v5 = v4.emplace_back(4); //L
```

 Des questions?

Une bonne manière d'identifier si une expression est une L-value ou une R-value est de se demander si on peut la placer à **gauche** d'un **=**

Si oui, c'est une L-value (L comme **Left**), si non, c'est une R-value.

Exemple :

`v1 = ...` // OK



L-value

`v2 + 5 - v1 = ...` // Ça n'a pas de sens



R-value

## Rappel

L'**overloading** (ou **surcharge**) est le mécanisme permettant de définir deux fonctions du même nom si elles ont un **nombre différent** de paramètres ou que les paramètres n'ont **pas le même type**.

Il est possible de créer des surcharges à partir de la **catégorie de valeur** (L-value ou R-value) des arguments.

Il est possible de créer des surcharges à partir de la **catégorie de valeur** (L-value ou R-value) des arguments.

C'est d'ailleurs ce que nous avons fait plus tôt avec

- les **constructeurs de copie** (qui attendent des **L-values**) et
- les **constructeurs de déplacement** (qui attendent des **R-values**).

Il est possible de créer des surcharges à partir de la **catégorie de valeur** (L-value ou R-value) des arguments.

C'est d'ailleurs ce que nous avons fait plus tôt avec

- les **constructeurs de copie** (qui attendent des **L-values**) et
- les **constructeurs de déplacement** (qui attendent des **R-values**).

L'instructionne `std::move` ne fait que transformer une L-value en R-value

1. Copie
2. Déplacement
3. L-Value et R-Value
- 4. Conteneurs**
  - a. Conteneurs séquentiels
  - b. Conteneurs associatifs
  - c. Tuples
5. Pointeurs intelligents
6. Héritage
7. Classes polymorphes

Un **conteneur séquentiel** est un conteneur

- dans lequel les éléments sont stockés dans un **ordre bien défini**,
- de telle sorte que le **1er élément, 2e élément, etc.** aient un sens

Par exemple :

- `std::array`
- `std::vector`
- `std::list`



Un **conteneur associatif** est un conteneur dans lequel **chaque élément est indexé par une clé**.

Cette indexation peut-être réalisée

- soit au moyen du **tri** des clés,
- soit au moyen de leur **hashage**.

Par exemple :

- `std::set` et `std::unordered_set`
- `std::map` et `std::unordered_map`

## Indexation par tri

Accès:  $O(\log n)$   
Insertion:  $O(\log n)$   
Suppression:  $O(\log n)$

Contraintes sur les clés:  
- comparables

## Indexation par hashage

Accès:  $O(1)$  amorti  
Insertion:  $O(1)$  amorti  
Suppression:  $O(1)$  amorti

Contraintes sur les clés:  
- équivalences  
- hashables

`std::map` et `std::unordered_map` sont des **dictionnaires** : à chaque clé est associé un seul et unique élément.

```
auto persons_by_name = std::map<std::string, Person> {  
    { "Celine", celine },  
    { "Julien", julien },  
};
```

```
persons_by_name.emplace("Donatien", donatien);  
persons_by_name.erase("Julien");
```

Indexation par tri

Indexation par hashage

`std::map` et `std::unordered_map` sont des **dictionnaires** : à chaque clé est associé un seul et unique élément.

```
auto persons_by_name = std::map<std::string, Person> {  
    { "Celine", celine },  
    { "Julien", julien },  
};
```

```
persons_by_name.emplace("Donatien", donatien);  
persons_by_name.erase("Julien");
```

`std::set` et `std::unordered_set` sont des **ensembles** : un élément ne peut être inséré que s'il n'est pas déjà présent dans le conteneur

```
auto persons = std::unordered_set<std::string> {  
    { "Celine" },  
    { "Julien" },  
};
```

```
auto gerald_it = persons.find("Gerald");  
auto has_gerald = (gerald_it != persons.end());
```

Indexation par tri

Indexation par hashage

`std::set` et `std::unordered_set` sont des **ensembles** : un élément ne peut être inséré que s'il n'est pas déjà présent dans le conteneur

```
auto persons = std::unordered_set<std::string> {  
    { "Celine" },  
    { "Julien" },  
};
```

```
auto gerald_it = persons.find("Gerald");  
auto has_gerald = (gerald_it != persons.end());
```

Les **tuples** permettent de stocker:

- plusieurs éléments
- de **types potentiellement différents**

(le nombre et type des éléments soit être connu statiquement)

La librairie standard propose les types `std::pair` et `std::tuple`.

Ils permettent notamment d'**éviter la définition de types-structurés** qui ne serviraient qu'à un seul endroit du programme.

Les **tuples** permettent de stocker un nombre **prédéfini** d'éléments de **types potentiellement différents**.

```
std::pair<std::string, unsigned int>  
get_name_and_age(const Person& person)  
{  
    return std::make_pair(person.get_name(), person.get_age());  
}
```



1. Copie.
2. Déplacement.
3. L-Value et R-Value
4. Conteneurs
- 5. Pointeurs intelligents**
6. Héritage.
7. Classes polymorphes.

Un **pointeur-intelligent** (ou **smart-pointer**) est un objet qui :

- contient un pointeur vers une donnée **allouée dynamiquement**
- **désalloue automatiquement** la donnée lorsqu'il est détruit
- gère de manière cohérente sa **copie** et son **déplacement**

Dans du code moderne :

- **tous** les **pointeurs-ownants** doivent être encapsulés dans des instances de **smart-pointers** ;
- les **pointeurs-nus** sont nécessairement des **pointeurs-observants**.

Un **pointeur-intelligent** (ou **smart-pointer**) est un objet qui :

- contient un pointeur vers une donnée **allouée dynamiquement**
- **désalloue automatiquement** la donnée lorsqu'il est détruit
- gère de manière cohérente sa **copie** et son **déplacement**

Les pointeurs intelligents fournis par la librairie standard sont :

- `std::unique_ptr`
- `std::shared_ptr`

Dans ce cours, nous nous intéresserons uniquement au premier.

- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme “unique”
  - des pointeurs bruts peuvent pointer vers la donnée

- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé

- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé
- On utilise `std::make_unique<type>` pour créer un `unique_ptr<type>`

- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé
- On utilise `std::make_unique<type>` pour créer un `unique_ptr<type>`
- La copie est interdite



- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé
- On utilise `std::make_unique<type>` pour créer un `unique_ptr<type>`
- La copie est interdite
- Peut être déplacé (avec `std::move`)

- Une sorte de pointeur (contient une adresse, peut être **nul**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé
- On utilise `std::make_unique<type>` pour créer un `unique_ptr<type>`
- La copie est interdite
- Peut être déplacé (avec `std::move`)
- Disponible dans `<memory>`

- Une sorte de pointeur (contient une adresse, peut être **null**)
- Invariant: un seul `unique_ptr` pointe vers la même donnée
  - d'où le terme "unique"
  - des pointeurs bruts peuvent pointer vers la donnée
- Contrairement à un pointeur brut, un `unique_ptr` **own** l'objet pointé
- On utilise `std::make_unique<type>` pour créer un `unique_ptr<type>`
- La copie est interdite
- Peut être déplacé (avec `std::move`)
- Disponible dans `<memory>`



Des questions?

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

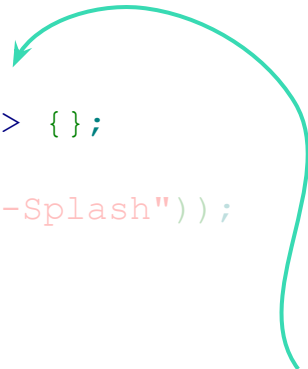
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}
```

```
int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



on instancie un `vector` de  
`unique_ptr<Car>`

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



on alloue dynamiquement un  
Car avec `make_unique`

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}
```

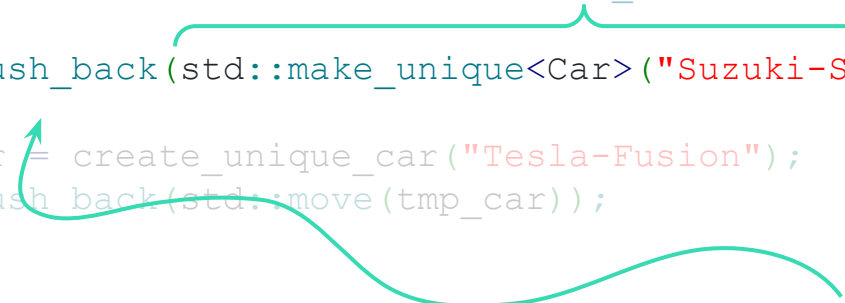
```
int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};
```

**R-value**

```
    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



l'élément est **déplacé** dans  
le tableau

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}
```


```
int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

on appelle  
`create_unique_car`





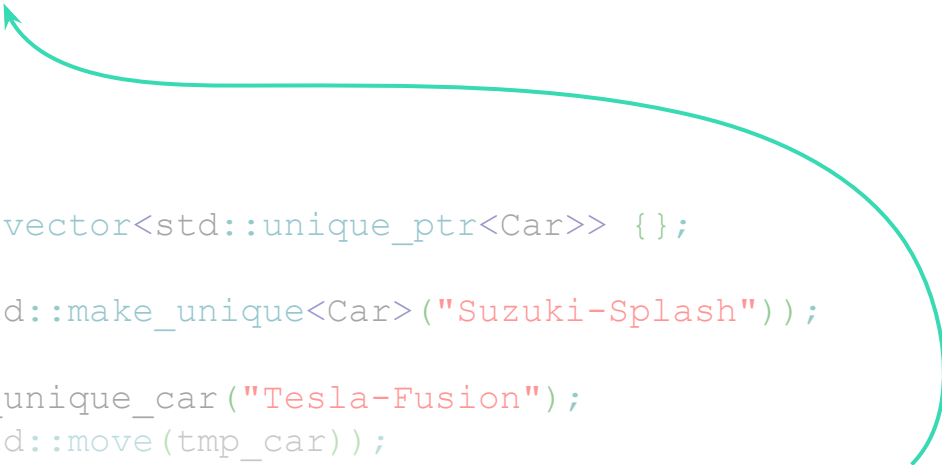
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



on alloue dynamiquement un  
Car avec `make_unique`

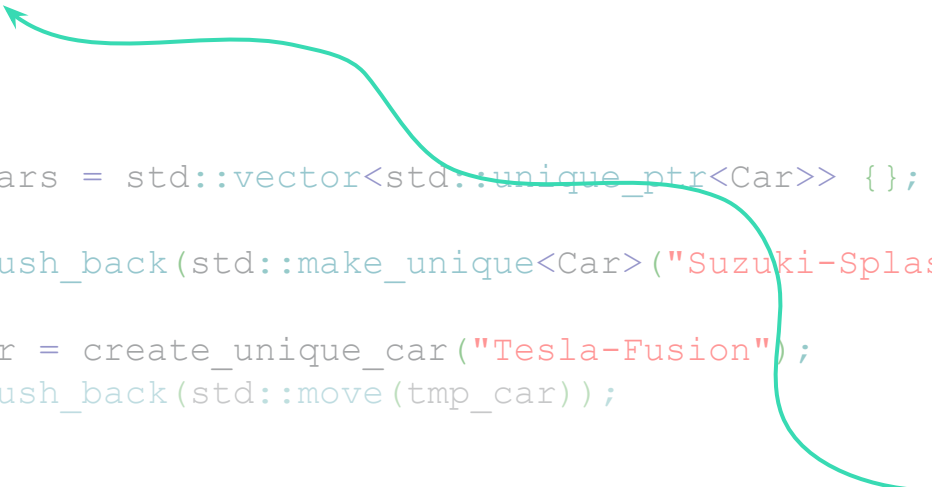
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



on renvoie le  
`unique_ptr` par valeur

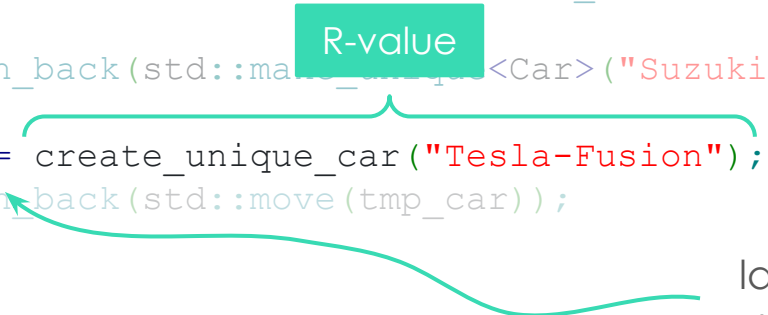
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



The diagram illustrates the concept of an R-value and move semantics. A green box labeled "R-value" is positioned over the expression `std::make_unique<Car>("Suzuki-Splash")` in the `push_back` call. A bracket connects this expression to the `create_unique_car` function call in the subsequent line. Another bracket connects the `create_unique_car` call to the `std::move(tmp_car)` expression in the `push_back` call. A curved arrow points from the `std::move(tmp_car)` expression to the `tmp_car` variable, indicating the transfer of ownership.

la valeur de retour est **déplacée**  
dans la variable `tmp_car`

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

`tmp_car` est une **L-value** ; si on l'ajoute au tableau directement, le compilateur va essayer de **copier** le `unique_ptr`



L-value

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```

tmp\_car est une L-value  
tableau direct  
ess

**ERREUR DE COMPILATION**

! !

unique\_ptr

L-value

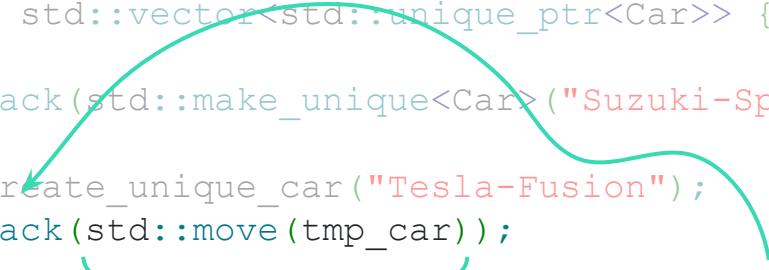
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}
```

```
int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



R-value

on utilise `std::move` pour **déplacer**  
le `unique_ptr` dans le tableau

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



`tmp_car` est désormais **vide**

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    many_cars.push_back(std::make_unique<Car>("Suzuki-Splash"));

    auto tmp_car = create_unique_car("Tesla-Fusion");
    many_cars.push_back(std::move(tmp_car));

    return 0;
}
```



Des questions?



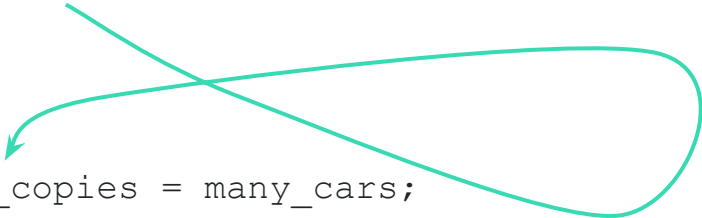
```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}

int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};

    /* .. */

    auto car_copies = many_cars;

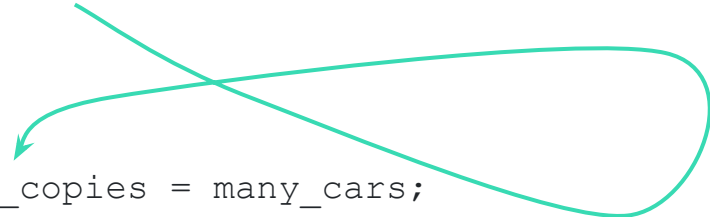
    return 0;
}
```



On essaie de copier un vecteur  
de unique ptr .  
Est-ce que ça va marcher?

```
std::unique_ptr<Car> create_unique_car(const std::string& model)
{
    auto car = std::make_unique<Car>(model);
    return car;
}
```

```
int main()
{
    auto many_cars = std::vector<std::unique_ptr<Car>> {};
```



```
    /* .. */
    auto car_copies = many_cars;

    return 0;
}
```

On essaie de copier un vecteur  
de unique ptr .  
Est-ce que ça va marcher?

Non ! copier le vecteur demande de  
copier les std::unique\_ptr

1. Copie
2. Déplacement
3. L-Value et R-Value
4. Conteneurs
5. Pointeurs intelligents
- 6. Héritage.**
  - a. Syntaxe
  - b. Instance d'une classe dérivée
7. Classes polymorphes.

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

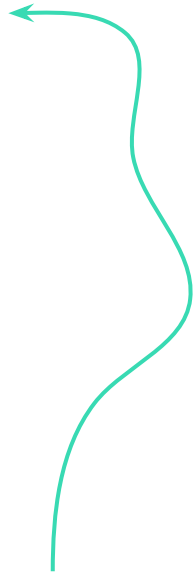
    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```



toute instance de Derived peut être  
considérée comme une instance de Base

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

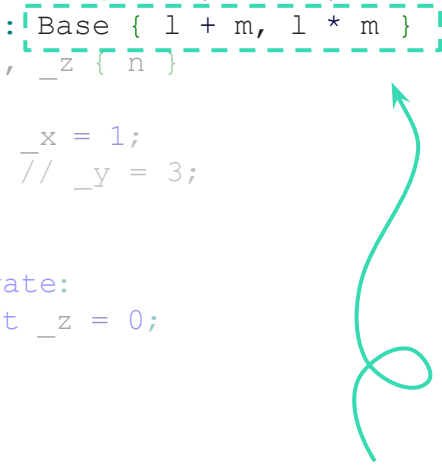
    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```



permet d'appeler le  
constructeur de la classe-parente

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}
```

```
    int get_y() const
    {
        return _y;
    }
```

```
protected:
    int _x = 0;
```

```
private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

permet l'accès aux attributs  
depuis les instances-filles

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

```

```
    int get_y() const
    {
        return _y;
    }

```

```
protected:
    int _x = 0;

```

```
private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {

```

```
        _x = 1;
        // _y = 3;
    }

```

```
private:
    int _z = 0;
};
```

accès  
valide



```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

```

```
    int get_y() const
    {
        return _y;
    }

```

```
protected:
    int _x = 0;

```

```
private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

```

```
private:
    int _z = 0;
};
```

accès  
invalide

```
class Base
{
public:
    Base(int x, int y)
        : _x { x }
        , _y { y }
    {}

    int get_y() const
    {
        return _y;
    }

protected:
    int _x = 0;

private:
    int _y = 0;
};
```

```
class Derived : public Base
{
public:
    Derived(int l, int m, int n)
        : Base { l + m, l * m }
        , _z { n }
    {
        _x = 1;
        // _y = 3;
    }

private:
    int _z = 0;
};
```

 Des questions?

On peut appeler les fonctions publiques de la classe parente sur les instances de la classe fille.

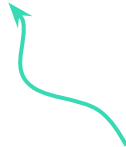
```
int main()
{
    auto derived = Derived { ... };
    std::cout << derived.get_y() << std::endl;

    return 0;
}
```

On peut appeler les fonctions publiques du type-parent sur les instances-filles.

```
int main()
{
    auto derived = Derived { ... };
    std::cout << derived.get_y() << std::endl;

    return 0;
}
```



`get_y()` est définie dans la partie publique de `Base`,  
donc on peut l'appeler sur une instance de `Derived`

On peut appeler les fonctions publiques de la classe parente sur les instances de la classe fille.

```
int main()
{
    auto derived = Derived { ... };
    std::cout << derived.get_y() << std::endl;

    return 0;
}
```

 Des questions?

On peut ensuite référencer les instances du type-enfant par le type-parent.

```
int main()
{
    auto derived = Derived { ... };
    Base& ref_base = derived;

    return 0;
}
```

```
void fcn(const Base& base)
{
    ...
}

int main()
{
    auto derived = Derived { ... };
    fcn(derived);

    return 0;
}
```

On peut ensuite référencer les instances du type-enfant par le type-parent.

```
int main()
{
    auto derived = Derived { ... };
    Base& ref_base = derived;
    -----
    return 0;
}
```

derived peut être référencé par  
son type parent Base

```
void fcn(const Base& base)
{
    -----
    ...
}
```

```
int main()
{
    auto derived = Derived { ... };
    fcn(derived);

    return 0;
}
```

On peut ensuite référencer les instances du type-enfant par le type-parent.

```
int main()
{
    auto derived = Derived { ... };
    Base& ref_base = derived;

    return 0;
}
```

```
void fcn(const Base& base)
{
    ...
}

int main()
{
    auto derived = Derived { ... };
    fcn(derived);

    return 0;
}
```

 Des questions?



Cela fonctionne aussi avec des pointeurs **bruts**.

```
int main()
{
    auto derived = Derived { ... };
    Base* ref_base = &derived;

    return 0;
}
```

```
void fcn(const Base* base)
{
    ...
}

int main()
{
    auto derived = Derived { ... };
    fcn(&derived);

    return 0;
}
```

Cela fonctionne aussi avec des pointeurs **bruts**.

```
int main()
{
    auto derived = Derived { ... };
    Base* ref_base = &derived;

    return 0;
}
```

Derived\* est convertible en Base\*

```
void fcn(const Base* base)
{
    ...
}
```

```
int main()
{
    auto derived = Derived { ... };
    fcn(&derived);

    return 0;
}
```

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

Voyez vous le problème dans  
cette ligne ?



```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

On essaie de stocker une Base sur la pile.

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

On n'a donc pas la place de stocker la Derived renvoyée par f

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

On essaie de stocker une Base sur la pile.

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

On n'a donc pas la place de stocker la Derived renvoyée par f

⚠ Une Derived est une Base donc le compilateur va **tronquer** ce qu'il dépasse

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

Voyez-vous le problème dans  
cette ligne ?



```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

On essaie encore de stocker  
une Base sur la pile.

```
int main()  
{  
  Base base1 = f(...);  
  Base base2 = g(...);  
  Base& base3 = g(...);  
}
```

g renvoie une L-value donc on  
devrait la copier, mais on ne peut  
pas stocker la copie

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

On essaie encore de stocker  
une Base sur la pile.

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

g renvoie une L-value (Derived&)  
donc on devrait la copier, mais  
Base ne sait se copier qu'une  
Base&

! Une Derived est une Base donc le  
compilateur va copier la Derived **tronquée**



```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

Voyez vous le problème dans  
cette ligne ?

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

Voyez vous le problème dans  
cette ligne ?

Il n'y en a pas :)  
Une Base& et une Derived& prennent  
la même place en mémoire.  
On pourra récupérer une Derived&  
plus tard

```
Derived f(...) { ... }  
Derived& g(...) { ... }
```

```
int main()  
{  
    Base base1 = f(...);  
    Base base2 = g(...);  
    Base& base3 = g(...);  
}
```

 Des questions?

Rappel:

- **Statique** = au moment de la compilation
- **Dynamique** = au moment de l'exécution

Et pour le type?

- **Type statique** = type déclarée dans le code
- **Type dynamique** = type réel à l'exécution

```
class Base
{
    /* .. */
};
```

```
class Derived1 : public Base
{
    /* .. */
};
```

```
class Derived2 : public Base
{
    /* .. */
};
```

```
Derived1& f1() {...}
Derived2& f2() {...}
Base& f3() {...}
```

```
int main()
{
    Base& x1 = f1();

    Base& x2 = f2();

    Base& x3 = f3();
}
```

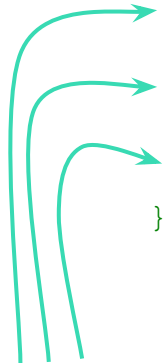
```
class Base
{
    /* .. */
};
```

```
class Derived1 : public Base
{
    /* .. */
};
```

```
class Derived2 : public Base
{
    /* .. */
};
```

```
Derived1& f1() {...}
Derived2& f2() {...}
Base& f3() {...}
```

```
int main()
{
    Base& x1 = f1();
    Base& x2 = f2();
    Base& x3 = f3();
}
```



Les types statiques de x1,  
x2 et x3 sont tous Base&

```
class Base
{
    /* .. */
};
```

```
class Derived1 : public Base
{
    /* .. */
};
```

```
class Derived2 : public Base
{
    /* .. */
};
```

```
Derived1& f1() {...}
Derived2& f2() {...}
Base& f3() {...}
```

```
int main()
{
    Base& x1 = f1();
    Base& x2 = f2();
    Base& x3 = f3();
}
```

Le type **dynamique** de  
x1 est probablement  
Derived1&

Le type **dynamique** de  
x2 est probablement  
Derived2&

Les types statiques de x1,  
x2 et x3 sont tous Base&

```
class Base
{
    /* .. */
};
```

```
class Derived1 : public Base
{
    /* .. */
};
```

```
class Derived2 : public Base
{
    /* .. */
};
```

```
Derived1& f1() {...}
Derived2& f2() {...}
Base& f3() {...}
```

```
int main()
```

```
{
    Base& x1 = f1();
    Base& x2 = f2();
    Base& x3 = f3();
}
```

Le type **dynamique** de  
x1 est probablement  
Derived1&

Le type **dynamique** de  
x2 est probablement  
Derived2&

! On ne connaît pas le  
type **dynamique** de x3

Les types statiques de x1,  
x2 et x3 sont tous Base&



🤔 Des  
questions?

```
class Base
{
    /* .. */
};
```

```
class Derived1 : public Base
{
    /* .. */
};
```

```
class Derived2 : public Base
{
    /* .. */
};
```

```
Derived1& f1() {...}
Derived2& f2() {...}
Base& f3() {...}
```

```
int main()
```

```
{
    Base& x1 = f1();
    Base& x2 = f2();
    Base& x3 = f3();
}
```

Le type **dynamique** de  
x1 est probablement  
Derived1&

Le type **dynamique** de  
x2 est probablement  
Derived2&

⚠ On ne connaît pas le  
type **dynamique** de x3

Les types statiques de x1,  
x2 et x3 sont tous Base&

1. Copie
2. Déplacement
3. L-Value et R-Value
4. Conteneurs
5. Pointeurs intelligents
6. Héritage
7. **Classes polymorphes**
  - a. Définition
  - b. Redéfinir le comportement d'une classe
  - c. Résolution d'appels
  - d. Fonctions virtuelles pures

En C++, l'héritage permet de répondre à 2 besoins orthogonaux :

- éviter la duplication de code
- spécialiser un comportement

En C++, l'héritage permet de répondre à 2 besoins orthogonaux :

- éviter la duplication de code
- spécialiser un comportement

Une classe dont on a pu **redéfinir le comportement** via héritage est une classe dont les instances peuvent se comporter différemment selon le **type dynamique** de l'objet.

On parle de **classes polymorphes**.

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};
```

```
class Instrument
{
public:
    [virtual] std::string get_name() const
    {
        return "???";
    }

    void describe() const
    {
        std::cout << "This is a " << get_name() << std::endl;
    }
};
```

indique que la fonction peut-être  
redéfinie par les classes-filles



```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};

class Guitar: public Instrument
{
public:
    std::string get_name() const override
    {
        return "guitar";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};

class Guitar: public Instrument
{
public:
    std::string get_name() const override
    {
        return "guitar";
    }
};
```

demande au compilateur de **vérifier**  
que la fonction est bien virtuelle

Optionnel, mais fortement conseillé



```
int main()
{
    Piano piano;
    Guitar guitar;

    std::vector<Instrument*> instruments { &piano, &guitar };

    for (const auto* instrument: instruments)
    {
        std::cout << instrument->get_name() << std::endl;
    }

    return 0;
}
```

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions.

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???" ;
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano" ;
    }
};
```

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

fonction  
virtuelle

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

fonction  
virtuelle

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

donc virtuelle  
aussi

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

fonction  
virtuelle

! **Attention** !  
**aux signatures**

```
class Piano: public Instrument
{
public:
    std::string get_name()
    {
        return "piano";
    }
};
```

ne redéfinit  
**pas** l'autre  
fonction

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???" ;
    }
};
```

fonction  
virtuelle

! **Attention** !  
**aux signatures**

```
class Piano: public Instrument
{
public:
    std::string get_name()
    {
        return "Piano" ;
    }
};
```

ne redéfinit  
**pas** l'autre  
fonction

! **BUG OBSCUR** !

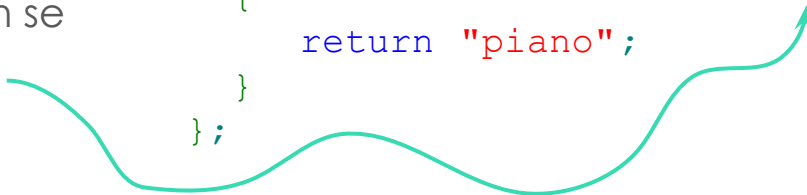


```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

 **Attention**   
**aux signatures**

**toujours** mettre `override` pour que  
le compilateur nous prévienne si on se  
trompe dans la signature

```
class Piano: public Instrument
{
public:
    std::string get_name() override
    {
        return "piano";
    }
};
```



```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???" ;
    }
};
```

**toujours** mettre `override` pour que  
le compilateur nous prévienne si on se  
trompe dans la signature

 **Attention**   
**aux signatures**

```
class Piano: public Instrument
{
public:
    std::string get_name() override
    {
        // ...
    }
};
```

**ERREUR DE COMPILATION**

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???" ;
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

🤔 Des questions?

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions.  
Pour garantir qu'un objet **polymorphe** sera **correctement détruit**, en particulier dans le cas d'**allocations dynamiques**, il faut toujours définir un **destructeur virtuel** dans la classe-mère (même s'il ne fait "rien").

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

type statique

on résout l'appel à `get_name()`

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

fonction non  
virtuelle

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

type statique

on résout l'appel à `get_name()`

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

fonction non  
virtuelle

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

type statique

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

on réalise un **appel statique**



```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

???

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```

```
class Instrument
{
public:
    std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const
    {
        return "piano";
    }
};
```



Des questions?

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions.

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
int main()
{
    Piano type statique
    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on résout l'appel à `get_name()`



fonction  
virtuelle

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

on résout l'appel à `get_name()`

fonction  
virtuelle

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

int type dynamique

```
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

on réalise un **appel dynamique**

fonction  
virtuelle

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

int type dynamique

```
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

on réalise un **appel dynamique**



```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

piano

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```

```
class Instrument
{
public:
    virtual std::string get_name() const
    {
        return "???";
    }
};
```

```
int main()
{
    Piano piano;

    Instrument& instrument = piano;
    std::cout << instrument.get_name() << std::endl;

    return 0;
}
```

```
class Piano: public Instrument
{
public:
    std::string get_name() const override
    {
        return "piano";
    }
};
```



Des questions?

1. Une fonction virtuelle dans une classe-mère est également virtuelle dans les classes-filles (si elle a la **même signature**)
2. Si une fonction n'est pas virtuelle, on appelle la version définie dans le **type statique** de l'objet
3. Si une fonction est virtuelle, on appelle la version définie dans le **type dynamique** de l'objet
4. L'appel au **destructeur** répond aux mêmes règles que les autres fonctions.

Pour garantir qu'un objet **polymorphe** sera **correctement détruit**, en particulier dans le cas d'**allocations dynamiques**, il faut toujours définir un **destructeur virtuel** dans la classe-mère (même s'il ne fait "rien").

Si une fonction n'a pas de sens à être définie dans la classe-mère, il n'est pas nécessaire de lui fournir une implémentation. On parle de **fonctions virtuelles pures**.

Si une classe contient des fonctions virtuelles pures, elle devient **abstraite** et n'est plus instanciable.

Les classes-filles doivent **redéfinir toutes les fonctions virtuelles pures** des types-parents pour **pouvoir être instanciées**.

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```

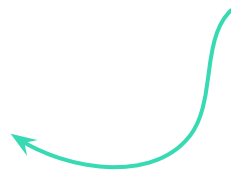
définit une fonction **virtuelle pure**

```
class Instrument
{
public:
    virtual std::string get_name() const [= 0];
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```



```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```

← Instrument est donc **abstraite**

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```

Instrument est donc **abstraite**  
donc **n'est pas instanciable**





```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```

Instrument est donc **abstraite**  
donc **n'est pas instanciable**

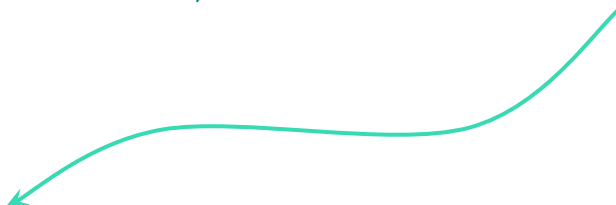


**ERREUR DE COMPILATION**

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

Flute définit get\_name

```
class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
```



```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

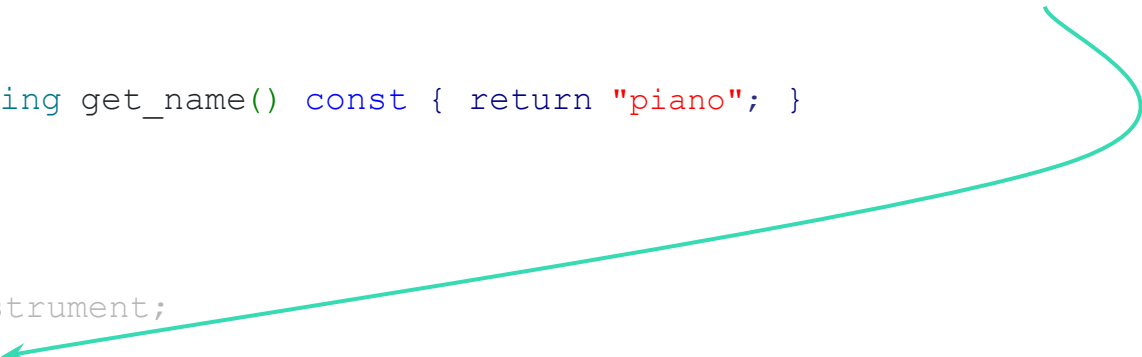
```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

```
class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};
```

```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

Flute n'a pas de fonctions  
virtuelle pure  
Elle est donc instanciable.



```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

```
class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};
```

```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

Flute n'a pas de fonctions  
virtuelle pure  
Elle est donc instanciable.



Pas de problème!

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

```
class Flute : public Instrument
{
public:
    virtual std::string get_name()      { return "piano"; }
```

Flute, j'ai oublié const



```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name()      { return "piano"; }
};

int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

Flute définit une **autre**  
fonction-membre



```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

Flute est donc **abstraite**

```
class Flute : public Instrument
{
public:
    virtual std::string get_name()      { return "piano"; }
};
```

```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

```
class Flute : public Instrument
{
public:
    virtual std::string get_name()      { return "piano"; }
```

```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

Flute est donc **abstraite**  
donc **pas instanciable** 😱



```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};
```

```
class Flute : public Instrument
{
public:
    virtual std::string get_name()      { return "piano"; }
};
```

```
int main()
{
    //Instrument instrument;
    Flute flute;

    return 0;
}
```

Flute est donc **abstraite**  
donc **pas instanciable** 😱

ERREUR DE COMPILATION OBSCURE

```
class Instrument
{
public:
    virtual std::string get_name() const = 0;
};

class Flute : public Instrument
{
public:
    virtual std::string get_name() const { return "piano"; }
};

int main()
{
    Instrument instrument;
    Flute flute;

    return 0;
}
```



Des questions?

- La copie
- Le déplacement
- Comment éviter (encore plus) de copies
- Conteneurs de base
- Utilisation des `std::unique_ptr`
- Héritage
- Résolution d'appel dynamique