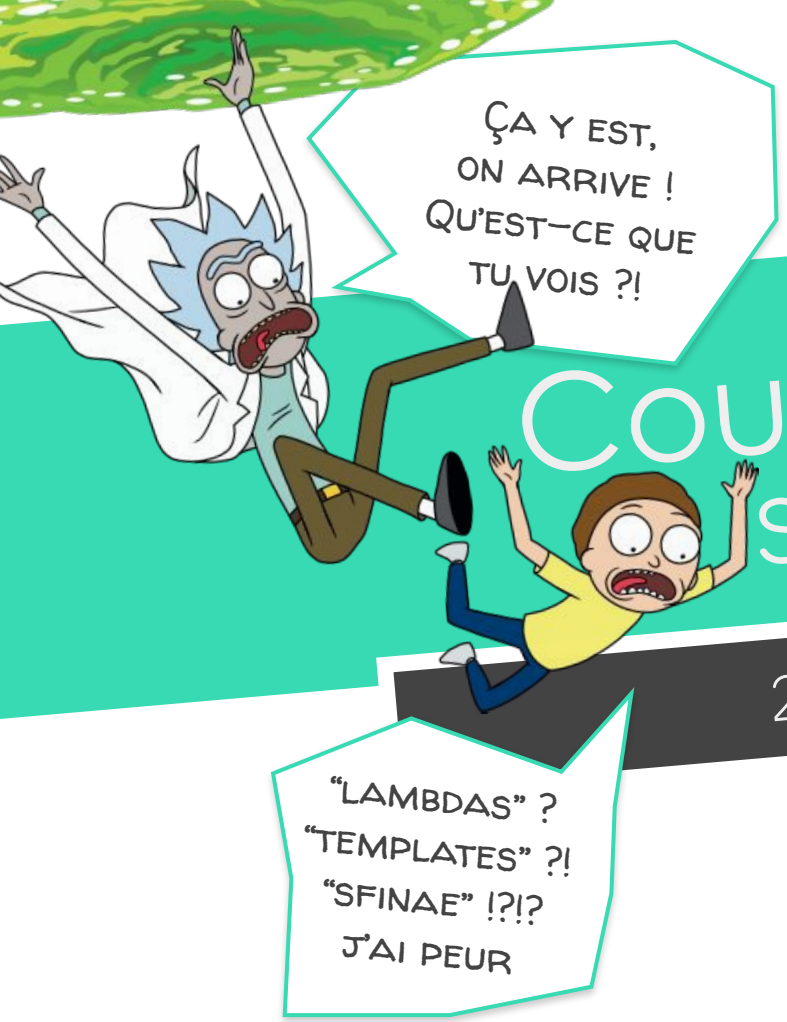


Cours de C++

Segment 3

2025-2026

Sommaire

1. Plages d'éléments et algorithmes de la STL
2. Foncteurs et Lambdas
3. Templates
4. Spécialisation de template
5. SFINAE
6. constexpr

Une **plage de données** est une suite d'éléments délimitée par

- un **itérateur de début**; et
- un **itérateur de fin**.

Pour récupérer la plage associée à un conteneur `ctn`, on utilise

- les fonctions-libres **`std::begin(ctn)`** et **`std::end(ctn)`**; ou
- les fonctions-membres **`ctn.begin()`** et **`ctn.end()`**.

```cpp
auto it_begin = std::begin(container);
auto it_end   = std::end(container);

auto half_range_size = std::distance(container.begin(), container.end())/2;
auto it_middle       = std::next(container.begin(), half_range_size);

auto it_rbegin = std::make_reverse_iterator(std::end(container));
auto it_rend   = std::make_reverse_iterator(std::begin(container));
```

Université Gustave Eiffel

(it_begin; it_end)
est une plage valide

```
auto it_begin = std::begin(container);
auto it_end   = std::end(container);

auto half_range_size = std::distance(container.begin(), container.end())/2;
auto it_middle       = std::next(container.begin(), half_range_size);

auto it_rbegin = std::make_reverse_iterator(std::end(container));
auto it_rend   = std::make_reverse_iterator(std::begin(container));
```

Université Gustave Eiffel

(it_begin; it_middle) et (it_middle; it_end) sont des plages valides

```
auto it_begin = std::begin(container);
auto it_end   = std::end(container);

auto half_range_size = std::distance(container.begin(), container.end())/2;
auto it_middle       = std::next(container.begin(), half_range_size);

auto it_rbegin = std::make_reverse_iterator(std::end(container));
auto it_rend   = std::make_reverse_iterator(std::begin(container));
```

Université Gustave Eiffel

(it_rbegin; it_rend) est une plage valide

```
auto it_begin = std::begin(container);
auto it_end   = std::end(container);

auto half_range_size = std::distance(container.begin(), container.end())/2;
auto it_middle       = std::next(container.begin(), half_range_size);

auto it_rbegin = std::make_reverse_iterator(std::end(container));
auto it_rend   = std::make_reverse_iterator(std::begin(container));
```

Université Gustave Eiffel

(it_rbegin; it_rend) est une plage valide

```
auto it_begin = std::begin(container);
auto it_end   = std::end(container);

auto half_range_size = std::distance(container.begin(), container.end())/2;
auto it_middle       = std::next(container.begin(), half_range_size);

auto it_rbegin = std::make_reverse_iterator(std::end(container));
auto it_rend   = std::make_reverse_iterator(std::begin(container));
```

🤨 Des questions?

On peut ensuite parcourir la plage à l'aide d'une **boucle for**.

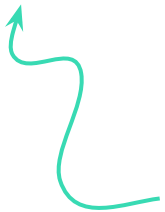
L'élément courant est obtenu en utilisant l'**`operator*`** de l'**itérateur**.

```
for (auto it = it_middle; it != it_end; ++it)
{
    auto& element = *it;
    ...
}
```

On peut ensuite parcourir la plage à l'aide d'une **boucle for**.

L'élément courant est obtenu en utilisant l'**`operator*`** de l'**itérateur**.

```
for (auto it = it_middle; it != it_end; ++it)
{
    auto& element = *it;
    ...
}
```



it_end pointe **après** le dernier élément de la plage

For each

La boucle **foreach** est un **raccourci syntaxique** permettant de parcourir les éléments de la plage allant du **début** du conteneur jusqu'à sa **fin**.

```
for (auto& value: ctn)
{
    ...
}
```

équivaut à

```
for (auto it = std::begin(ctn), it_end = std::end(ctn); it != it_end; ++it)
{
    auto& value = *it;
    ...
}
```

Opérations usuelles

La librairie standard expose un certain nombre de fonctions permettant de traiter ou de modifier des plages de données.

Ces fonctions sont disponibles dans les headers `<algorithm>` et `<numeric>`.

Université Gustave Eiffel

`std::find`

recherche un élément équivalent

```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la valeur a été trouvée
}
```

Université Gustave Eiffel

`std::find`

recherche un élément équivalent

```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la vale
}
```

fin de plage

début de plage

valeur à trouver

`std::find`

recherche un élément équivalent

```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la valeur a été trouvée
}
```

Contrainte

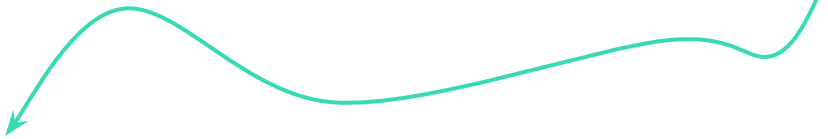
un **`operator==`** permettant de comparer un élément de `ctn` et `value_to_find` doit être défini

Opérations usuelles

`std::find`

recherche un élément équivalent

retourne un **itérateur** sur l'élément trouvé



```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la valeur a été trouvée
}
```

`std::find`

recherche un élément équivalent

retourne un **itérateur** sur l'élément trouvé

```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la valeur a été trouvée
}
```

Ou l'**iterateur de fin** si aucun n'est trouvé

`std::find`

recherche un élément équivalent

```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);
if (it_value != ctn.end())
{
    // la valeur a été trouvée
}
```

🤨 Des questions?

`std::find_if`

recherche un élément vérifiant un prédicat

```
auto it_char = std::find_if(str.begin(), str.end(), is_lowercase);
if (it_char != str.end())
{
    // la valeur a été trouvée
}
```

`std::find_if`

recherche un élément vérifiant un prédicat

fin de plage

```
auto it_char = std::find_if(str.begin(), str.end(), is_lowercase);
if (it_char != str.end())
{
    // la valeu
}
```

début de plage

prédicat

`std::find_if`

recherche un élément vérifiant un prédicat

```
auto it_char = std::find_if(str.begin(), str.end(), is_lowercase);
if (it_char != str.end())
{
    // la valeur a été trouvée
}
```

Contrainte

il faut qu'il soit possible d'appeler `is_lowercase` en lui passant un élément de `str`

`std::find_if`

recherche un élément vérifiant un prédicat

```
auto it_char = std::find_if(str.begin(), str.end(), is_lowercase);
if (it_char != str.end())
{
    // la valeur a été trouvée
}
```

retourne un itérateur sur l'élément si il a été trouvé, ou sur la fin de la plage sinon

Opérations usuelles

`std::all_of`, `std::any_of`, `std::none_of`
indique si chaque élément vérifie un prédicat (resp. un ou aucun)

```
auto has_no_caps = std::all_of(str.begin(), str.end(), is_lowercase);
```

Opérations usuelles

`std::all_of`, `std::any_of`, `std::none_of`
indique si chaque élément vérifie un prédicat (resp. un ou aucun)

```
auto has_no_caps = std::all_of(str.begin(), str.end(), is_lowercase);
```

début de plage

fin de plage

prédicat

`std::all_of`, `std::any_of`, `std::none_of`
indique si chaque élément vérifie un prédicat (resp. un ou aucun)

```
auto has_no_caps = std::all_of(str.begin(), str.end(), is_lowercase);
```

Contrainte

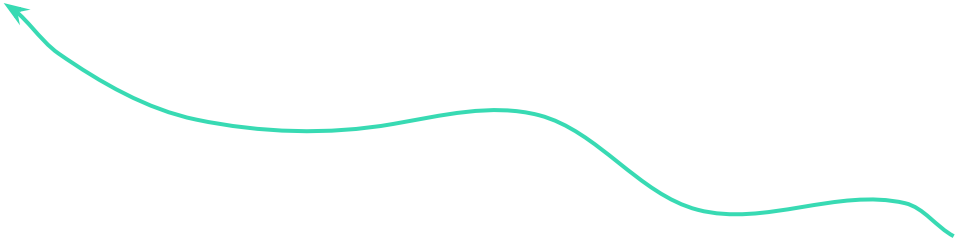
il faut qu'il soit possible d'appeler `is_lowercase` en lui passant un élément de `str`

Opérations usuelles

`std::all_of`, `std::any_of`, `std::none_of`

indique si chaque élément vérifie un prédicat (resp. un ou aucun)

```
auto has_no_caps = std::all_of(str.begin(), str.end(), is_lowercase);
```



retourne un booléen

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```

Université Gustave Eiffel

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

fin de plage

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```

début de plage

prédicat

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```

Contrainte

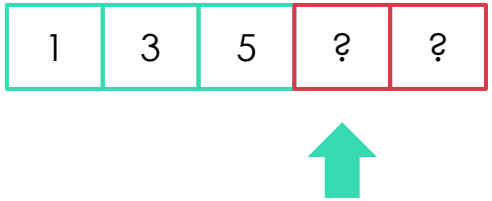
il faut qu'il soit possible d'appeler `is_negative` en lui passant un élément de `vals`

Université Gustave Eiffel

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```

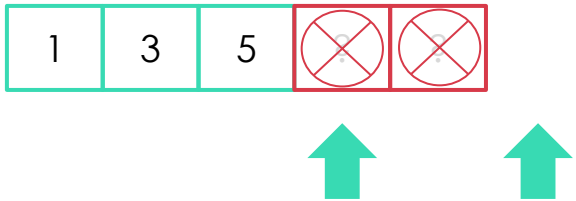


retourne l'itérateur sur la fin de la plage contenant les éléments ne vérifiant pas le prédicat

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```



on peut ensuite utiliser `erase` pour **supprimer effectivement** les éléments du conteneur

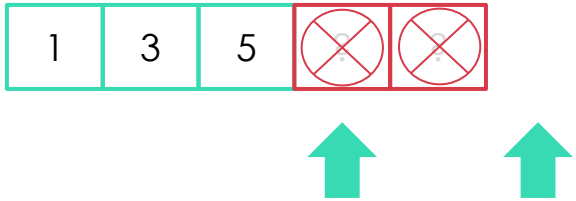
Université Gustave Eiffel

🤨 Des questions?

`std::remove_if`

réordonne une plage pour éliminer les éléments vérifiant un prédicat

```
auto it_end = std::remove_if(vals.begin(), vals.end(), is_negative);
vals.erase(it_end, vals.end());
```



on peut ensuite utiliser `erase` pour **supprimer effectivement** les éléments du conteneur

1. Plages d'éléments et algorithmes de la STL
2. **Foncteurs et Lambdas**
3. Templates
4. Spécialisation de template
5. SFINAE
6. constexpr

Un **foncteur** est un **objet** pouvant être utilisé comme une **fonction**.

Pour créer un foncteur, il faut définir une classe définissant un `operator()`, puis instancier cette classe.

```cpp
struct IsPositiveNumber
{
    bool operator()(int nb) const
    {
        return nb >= 0;
    }
};

auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```

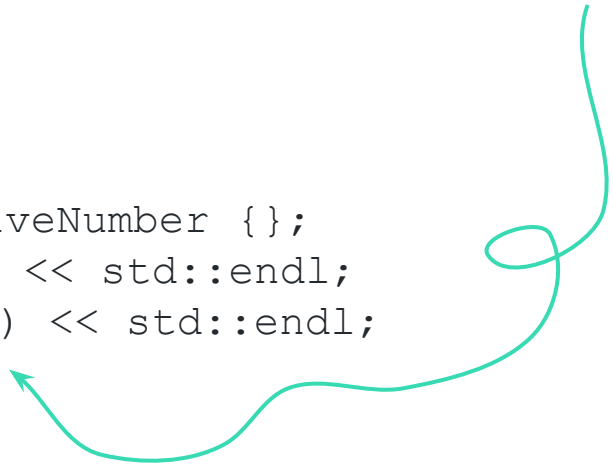
```
struct IsPosi Nom mber
{
    bool operator()(int nb) const
    {
        return nb >= 0;
    }
};

auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```

Paramètres

```
struct IsPositiveN
{
    bool operator()(int nb) const
    {
        return nb >= 0;
    }
};

auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```

```
struct IsPositiveNumber
{
    bool operator()(int nb) const
    {
        return nb >= 0;
    }
};

auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```

`functor` peut être utilisé comme une fonction ayant la même signature que l'`operator()`



```cpp
struct IsPositiveNumber
{
    bool operator()(int nb) const
    {
        return nb >= 0;
    }
};

auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```

🤨 Des questions?

Université Gustave Eiffel

Les foncteurs peuvent avoir des **attributs**, puisqu'il s'agit d'objets.

```
class EqualValue
{
public:
     EqualValue(int value) : _value{value} {}

     bool operator()(int other) const
     {
          return _value == other;
     }
     int _value;
};
```

```
auto equals_3 = EqualValue {  3 };
std::cout << equals_3(3) << std::endl;


std::cout << equals_3(42) << std::endl;
```

Les foncteurs peuvent avoir des **attributs**, puisqu'il s'agit d'objets.

```
class EqualValue
{
public:
    EqualValue(int value) : _value{value} {}

    bool operator()(int other) const
    {
        return _value == other;
    }
    int _value;
};
```

On construit un `EqualValue` avec `_value` qui vaut 3

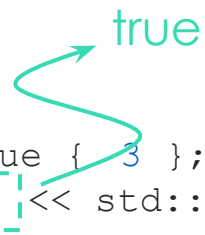
```
auto equals_3 = EqualValue {  3 };
std::cout << equals_3(3) << std::endl;


std::cout << equals_3(42) << std::endl;
```

Université Gustave Eiffel

Les foncteurs peuvent avoir des **attributs**, puisqu'il s'agit d'objets.

```
class EqualValue
{
public:
     EqualValue(int value) : _value{value} {}

     bool operator()(int other) const
     {
          return _value == other;
     }
     int _value;
};
```

```
auto equals_3 = EqualValue { 3 };
std::cout << equals_3(3) << std::endl;


std::cout << equals_3(42) << std::endl;
```



Université Gustave Eiffel

Les foncteurs peuvent avoir des **attributs**, puisqu'il s'agit d'objets.

```
class EqualValue
{
public:
     EqualValue(int value) : _value{value} {}

     bool operator()(int other) const
     {
          return _value == other;
     }
     int _value;
};
```

```
auto equals_3 = EqualValue { 3 };
std::cout << equals_3(3) << std::endl;
```

true

```
std::cout << equals_3(42) << std::endl;
```

false

Université Gustave Eiffel

Les foncteurs peuvent avoir des **attributs**, puisqu'il s'agit d'objets.

```
class EqualValue
{
public:
     EqualValue(int value) : _value{value} {}

     bool operator()(int other) const
     {
          return _value == other;
     }
     int _value;
};
```

```
auto equals_3 = EqualValue {  3 };
std::cout << equals_3(3) << std::endl;
```

```
std::cout << equals_3(42) << std::endl;
```

🤨 Des questions?

Université Gustave Eiffel

Comment utiliser le contenu de `input` à l'intérieur du prédicat ?

```
bool has_any_equal_to_input(const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;


    return std::any_of(values.begin(), values.end(), ????);
}
```

Université Gustave Eiffel

On peut utiliser notre `classe` EqualValue !

```
bool has_any_equal_to_input (const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;

    auto equals_input = EqualValue { input };
    return std::any_of(values.begin(), values.end(), equals_input);
}
```

On instancie notre foncteur

Université Gustave Eiffel

On peut utiliser notre `classe` EqualValue !

```
bool has_any_equal_to_input (const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;

    auto equals_input = EqualValue { input };
    return std::any_of(values.begin(), values.end(), equals_input);
}
```

On instancie notre foncteur

Et on passe à `std::any_of`

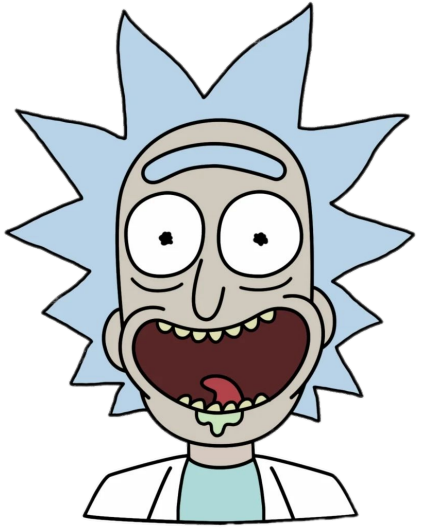
🤨 Des questions?

```
bool has_any_equal_to_input(const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;


    auto equals_input = EqualValue { input };
    return std::any_of(values.begin(), values.end(), equals_input);
}
```

Mais c'est horriblement **pas pratique**
et sacrément **verbeux** !!!

Mais c'est horriblement **pas pratique**
et sacrément **verbeux** !!!

Heureusement, les **lambdas** sont là !



Une **lambda** est un

- foncteur
- dont le type est **implicite**
- instancié en **une seule instruction** via la syntaxe suivante :

```
[input] (int value) { return value == input; }
```

Une **lambda** est un

- foncteur
- dont le type est **implicite**
- instancié en **une seule instruction** via la syntaxe suivante :

```
[input] (int value) { return value == input; }
```

Paramètres

Une **lambda** est un

- foncteur
- dont le type est **implicite**
- instancié en **une seule instruction** via la syntaxe suivante :

```
[input] (int value) { return value == input; }
```

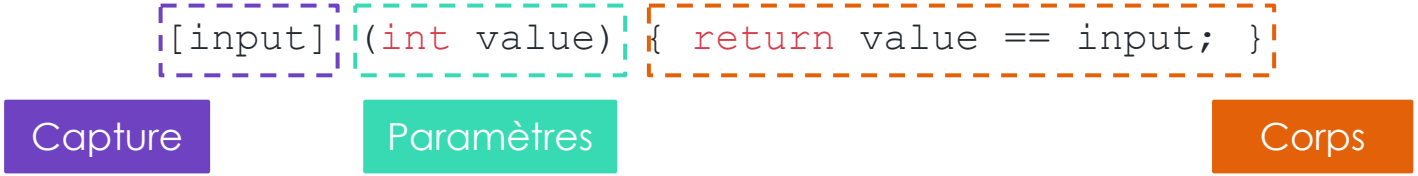
Paramètres

Corps

Université Gustave Eiffel

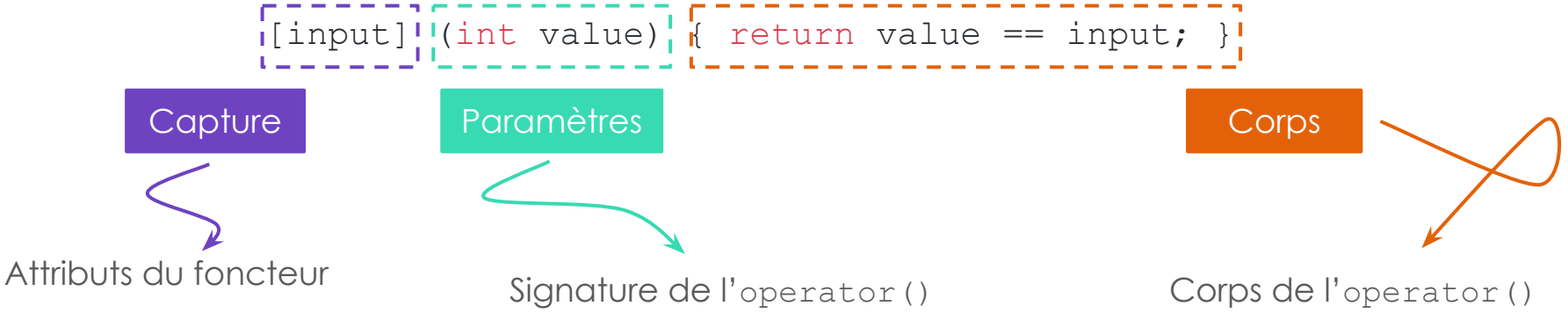
Une **lambda** est un

- foncteur
- dont le type est **implicite**
- instancié en **une seule instruction** via la syntaxe suivante :



Une **lambda** est un

- foncteur
- dont le type est **implicite**
- instancié en **une seule instruction** via la syntaxe suivante :



On peut ainsi réécrire le code suivant :

```
struct EqualValue
{ … };

bool has_any_equal_to_input(const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;

    auto equals_input = EqualValue { input };
    return std::any_of(values.begin(), values.end(), equals_input);
}
```

de la façon suivante :

```
bool has_any_equal_to_input(const std::vector<int>& values)
{
    auto input = 0;
    std::cin >> input;

    return std::any_of(
        values.begin(),
        values.end(),
        [input](int value) { return value == input; }
    );
}
```

Capture par valeur ou référence

Notez également que les variables locales peuvent être capturées :

- soit par valeur : `[var1, var2]`
- soit par référence : `[&var1, &var2]`

On peut également créer de nouvelles variables en les assignant à l'intérieur de la capture : `[sum = var1 + var2]`

Comprendre les deux types de captures

Université Gustave Eiffel

```cpp
bool has_any_equal_to_input(const std::vector<int>& values)
{
    int input = 0;
    std::cin >> input;

    bool b1 = std::any_of(
        values.begin(),
        values.end(),
        [input](int value) { return value == input; }
    );

    bool b2 = std::any_of(
        values.begin(),
        values.end(),
        [&input](int value) { return value == input; }
    );
}
```

Comprendre les deux types de captures

Université Gustave Eiffel

```
bool has_any_equal_to_input(const std::vector<int>& values)
{
    int input = 0;
    std::cin >> input;

    bool b1 = std::any_of(
        values.begin(),
        values.end(),
        [input](int value) { return value == input; }
    );

    bool b2 = std::any_of(
        values.begin(),
        values.end(),
        [&input](int value) { return value == input; }
    );
}
```

On crée un foncteur, avec
`auto input = input;`

On crée un foncteur, avec
`auto &input = input;`

Comprendre les deux types de captures

Université Gustave Eiffel

```
bool has_any_equal_to_input(const std::vector<int>& values)
{
    int input = 0;
    std::cin >> input;

    bool b1 = std::any_of(
        values.begin(),
        values.end(),
        [input](int value) { return value == input; }
    );

    bool b2 = std::any_of(
        values.begin(),
        values.end(),
        [&input](int value) { return value == input; }
    );
}
```

On crée un foncteur, avec
`auto input = input;`

On crée un foncteur, avec
`auto &input = input;`

Des questions?

Si vous ne souhaitez **rien** capturer dans votre lambda, il faut quand même écrire les crochets de la capture :

```
[](const std::string& str) { return str.empty(); }
```

Spécifier le type retour

Parfois, le compilateur ne pourra pas déduire tout seul le type de retour de la lambda, on peut le spécifier avec ->

```
[](std::string str) -> std::vector<std::string> {  return {“Hello”, str}; }
```

Il est possible de **stocker** une lambda dans une **variable locale**.

Pour cela, il faut forcément utiliser `auto`, car le **type** de la lambda est généré **pendant la compilation**.

```
auto is_empty_str = [](const std::string& str)
                                                { return str.empty(); };
```

Si vous souhaitez stocker une lambda dans un **attribut** d'une classe, il est nécessaire de l'**encapsuler** dans un objet de type `std::function<...>`.

Ce type est défini dans `<functional>`.

```
struct MyStringPredicate
{
    std::function<bool(const std::string&)> predicate;
};
```

Stocker une lambda

Si vous souhaitez stocker une lambda dans un **attribut** d'une classe, il est nécessaire de l'**encapsuler** dans un objet de type `std::function<...>`.

Ce type est défini dans `<functional>`.

Type de retour

```
struct MyS                    e
{
    std::function<bool(const std::string&)> predicate;
};
```

Si vous souhaitez stocker une lambda dans un **attribut** d'une classe, il est nécessaire de l'**encapsuler** dans un objet de type `std::function<...>`.

Ce type est défini dans `<functional>`.

```
struct MyStringPredicate
{
    std::function<bool(const std::string&)> predicate;
};
```

Paramètres

Une `std::function` peut stocker une fonction-libre, un foncteur ou bien une lambda, du moment que leur prototype correspond à celui attendu par la `std::function`.

```
auto pred = MyStringPredicate {};

pred.predicate = is_empty_str;
std::cout << pred.predicate("") << std::endl;

pred.predicate = [name](const std::string& str) { return name == str; };
std::cout << pred.predicate("Celine") << std::endl;
```

1. Plages d'éléments et algorithmes de la STL
2. Foncteurs et Lambdas
3. **Templates**
4. Spécialisation de template
5. SFINAE
6. constexpr

C'est quoi un template ?

Un template, ou patron, est un modèle qui sert à **générer du code** automatiquement.

Les templates permettent de faire du **polymorphisme** et de la **généricité statique** en C++.

Que peut-on templater?

- Les classes-template
  Ex: `std::vector`, `std::map`, et autres conteneurs
- Fonctions-template,
  Ex: `std::make_unique` ou `std::move`

Université Gustave Eiffel

C'est quoi un template ?

Un template, ou patron, est un modèle qui sert à **générer du code** automatiquement.

Les templates permettent de faire du **polymorphisme** et de la **généricité statique** en C++.

Que peut-on templater?

- Les classes-template
  Ex: `std::vector`, `std::map`, et autres conteneurs
- Fonctions-template,
  Ex: `std::make_unique` ou `std::move`

Université Gustave Eiffel

C'est quoi un template ?

Un template, ou patron, est un modèle qui sert à **générer du code** automatiquement.

Les templates permettent de faire du **polymorphisme** et de la **généricité statique** en C++.

Que peut-on templater?

- Les classes-template
  Ex: `std::vector`, `std::map`, et autres conteneurs
- Fonctions-template,
  Ex: `std::make_unique` ou `std::move`

Notre But: Faire bosser le compilo

Syntaxe

```
template <typename T>
void print_between_parentheses( const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Syntaxe

```
template <typename T>
void print_between_parentheses( const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Mot-clé utilisé pour indiquer qu'on crée un template.

Syntaxe

Paramètres du template entre <>
Ici: un paramètre qui est un `typename`, c'est-à-dire un nom de type

```
template <typename T>
void print_between_parentheses( const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Syntaxe

```
template <typename T>
void print_between_parentheses( const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Prototype usuel de la fonction

Université Gustave Eiffel

Syntaxe

Les valeurs des paramètres **template** doit être connues **statiquement**

```
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Les valeurs des paramètres classiques ne peuvent être connues que **dynamiquement**

Syntaxe

```
template <typename T>
void print_between_parentheses( const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

🤨 Des questions?

Université Gustave Eiffel

Instanciation

« *Instancier un template* » signifie que le compilateur va **générer** une instance du modèle.

Ex: `std::min<int>` et `std::min<std::string>` sont **deux instances différentes** de `std::min`. Ce sont deux fonctions différentes dans le code compilés.

Les templates sont instanciées automatiquement par le compilateur quand on les utilise.

⚠️ **Attention :** Il faut que le compilateur ait le code du template pour pouvoir en faire une instanciation. Il faut donc mettre tout le code dans les headers.

Instanciation de fonctions-template

Université Gustave Eiffel

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Instanciation de fonctions-template


```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Instanciation de fonctions-template

Université Gustave Eiffel

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Instanciation de fonctions-template

Le compilateur va générer la fonction `print_between_parentheses<std::string>` à partir du modèle.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Université Gustave Eiffel

Le compilateur va générer la fonction `print_between_parentheses<std::string>` à partir du modèle.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Instanciation

```cpp
void print_between_parentheses<std::string>(const std::string& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Université Gustave Eiffel

Le compilateur va générer la fonction `print_between_parentheses<std::string>` à partir du modèle.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parenthe
}
```

Instanciation

Cette fonction est ajoutée à l'unité de compilation courante (fichier `.o`) et sera correctement linkée au programme.

```cpp
void print_between_parentheses<std::string>(const std::string& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Université Gustave Eiffel

🤔 Des questions?

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    print_between_parentheses<std::string>("pouet");
}
```

Instanciation

```cpp
void print_between_parentheses<std::string>(const std::string& value)
{
    std::cout << '(' << value << ')' << std::endl;
}
```

Déduction du paramètre template

Université Gustave Eiffel

Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être **automatiquement déduits** au moment de **l'appel** à la fonction.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    std::string ping = "ping";
    print_between_parentheses<std::string>(ping);

    print_between_parentheses<std::string>("pong");
}
```

Pas nécessaire car le compilo peut le **déduire**

Déduction du paramètre template

Université Gustave Eiffel

Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être **automatiquement déduits** au moment de **l'appel** à la fonction.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    std::string ping = "ping";
    print_between_parentheses(ping);

    print_between_parentheses("pong");
}
```

On laisse le compilo déduire, mais que va-t-il instancier ici?

Déduction du paramètre template

Université Gustave Eiffel

Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être **automatiquement déduits** au moment de **l'appel** à la fonction.

```cpp
#include <iostream>

template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

int main()
{
    std::string ping = "ping";
    print_between_parentheses(ping);

    print_between_parentheses("pong");
}
```

Des questions?

Déduction du paramètre template

Si les paramètres du template sont utilisés dans la signature de la fonction, alors ceux-ci peuvent être **automatiquement déduits** au moment de **l'appel** à la fonction.

```cpp
template <typename T>
void print_between_parentheses(const T& value)
{
    std::cout << '(' << value << ')' << std::endl;
}

template <typename T>
T cast_integer(int i)
{
    return static_cast<T>(i);
}
```

`T` est utilisé dans la signature, la déduction pourra se faire.

`T` n'est pas utilisé dans la signature, la déduction ne pourra pas se faire.

Déduire comme un compilo

Université Gustave Eiffel

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

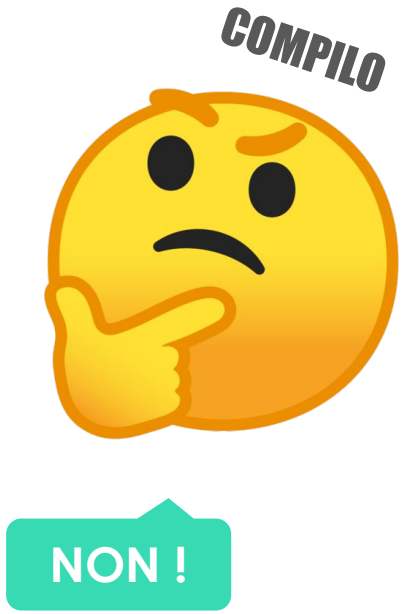
Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

Est-ce que je connais une fonction `min` à 2 paramètres ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```



Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

Est-ce que je connais une fonction-template `min` à 2 paramètres ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

OUI !

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```


```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

Est-ce que les types des arguments me permettent de déduire les paramètres du template ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```

int int

OUI !

```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

T = int T = int

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```


```cpp
int min<int>(int v1, int v2) { ... }
```

J'instancie `std::min<int>`

Université Gustave Eiffel

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(13, 4) << std::endl;
}
```


```cpp
int min<int>(int v1, int v2) { ... }
```

🤨 Des questions?

Déduire comme un compilo

Université Gustave Eiffel

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

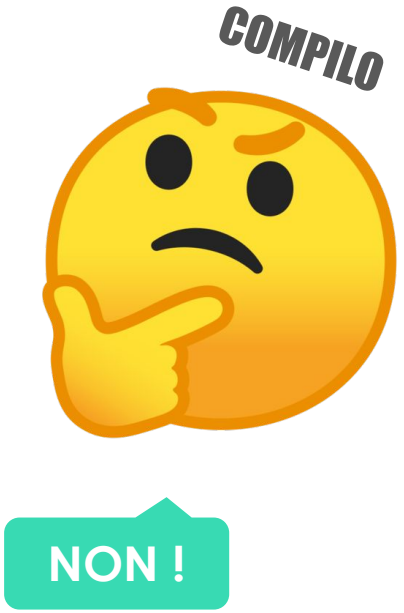
Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

Est-ce que je connais une fonction `min` à 2 paramètres ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```



Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

Est-ce que je connais une fonction-template `min` à 2 paramètres ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

OUI !

Déduire comme un compilo

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```


```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

Est-ce que les types des arguments me permettent de déduire les paramètres du template ?

Déduire comme un compilo


```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

double

int

```cpp
template <typename T>
T min(T v1, T v2) { ... }
```

T = double

T = int

NON !

Déduire comme un compilo

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

J'insulte le développeur !!

```
<source>: In function 'int main()':
<source>:23:13: error: no matching function for call to 'min(double, int)'
   23 |     std::min(1.3, 4);
      |     ~~~~~~~~^~~~~~~~
In file included from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/string:50,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/locale_classes.h:40,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/ios_base.h:41,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/ios:42,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/ostream:38,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/iostream:39,
                 from <source>:1:
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/stl_algobase.h:230:5: note: candidate:
'template<class _Tp> constexpr const _Tp& std::min(const _Tp&, const _Tp&)'
  230 |     min(const _Tp& __a, const _Tp& __b)
      |     ^~~
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/stl_algobase.h:230:5: note:  template
argument deduction/substitution failed:
<source>:23:13: note:  deduced conflicting types for parameter 'const _Tp' ('double' and 'int')
   23 |     std::min(1.3, 4);
      |     ~~~~~~~~^~~~~~~~
```

Déduire comme un compilo

```cpp
#include <algorithm>
#include <iostream>

int main()
{
    std::cout << std::min(1.3, 4) << std::endl;
}
```

🤨 Des questions?

Syntaxe

```
template <typename T>
struct Fraction
{
    T dividende = {};
    T diviseur = {};
};
```

Université Gustave Eiffel

Syntaxe

mot-clef pour indiquer qu'on définit un template

```
template <typename T>
struct Fraction
{
    idende
    iseur = {};
};
```

nom du template

liste de paramètres du template

Instanciation

`Fraction<int>` et `Fraction<double>` sont des **instances** du **template** `Fraction`.

`Fraction<int>` et `Fraction<double>` sont des **types**, mais `Fraction` n'est **pas un type**.

Comme pour les fonctions-templates, le compilateur instancie les classes templates dès qu'on les utilise.

⚠️ **Attention :** pour que cela fonctionne, il faut que le compilateur ait eu connaissance du template. Pensez donc bien à mettre **tout le code** de vos templates dans les **headers**.

```cpp
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

le compilateur enregistre le template Printer dans sa base de données

```cpp
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

```cpp
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};
```

```cpp
int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

= **Instanciation**

le compilateur va générer le type `Printer<double>` à partir du modèle = instanciation

Université Gustave Eiffel

```
template <typename
class Printer
{
public:
    void parenthe
    void quote(co
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

```
class Printer<double>
{
public:
    Printer() {} // constructeur généré par défaut
};
```

les fonctions-membres sont générées uniquement au moment de leur **utilisation**

```cpp
template <typenam
class Printer
{
public:
    void parenthe
    void quote(co
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

```cpp
class Printer<double>
{
public:
    Printer() {}
    void quote(const double& v) const { ... }
};
```

les fonctions-membres sont générées uniquement au moment de leur **utilisation**

```cpp
template <typename T>
class Printer
{
public:
    void parentheses(const T& v) const { std::cout << '(' << v << ')'; }
    void quote(const T& v) const { std::cout << '"' << v << '"'; }
};

int main()
{
    const auto printer = Printer<double>{};
    printer.quote(5.2);
}
```

🤨 Des questions?

"Faut-il mettre le code dans le hpp ou dans un cpp séparé?"

- Cpp séparé => meilleure lisibilité
- Cpp séparé => plusieurs modules de compilation (un par cpp)
- Le code des templates doit être dans le hpp
- Les template sont instanciés dans chaque module (chaque ccp)
  - Ex: std::vector<std::string> a des chances d'être généré dans chaque module
  - Duplication du code binaire => explosion de la taille du binaire totale
  - Contre-intuitivement, la compilation peut-être plus longue en séparant puisque le compilateur peut refaire plusieurs fois le même travail de déduction

1. Plages d'éléments et algorithmes de la STL
2. Foncteurs et Lambdas
3. Templates
4. **Spécialisation de template**
5. SFINAE
6. constexpr

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui du template au moment de l'instanciation.

```
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

Université Gustave Eiffel

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui du template au moment de l'instanciation.

template

```
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

spécialisation pour <0>

Université Gustave Eiffel

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que c
moment de l'instanciation.

```cpp
int main()
{
    divide_by<3>(5);
    divide_by<0>(5);
}
```

```cpp
template <int Diviseur>
void divide_by(int dividende)
{
    std::cout << (dividende / Diviseur) << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
    std::cerr << "Cannot divide " << dividende << " by 0!" << std::endl;
}
```

Pourquoi faire simple quand on peut faire compliqué

Université Gustave Eiffel

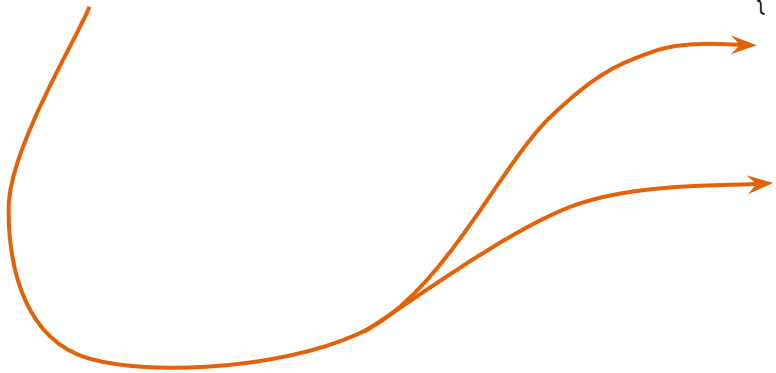
```
template <int Diviseur>
void divide_by(int dividende)
{
  std::cout << (dividende/Diviseur)
            << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
  std::cout << "Cannot divide "
            << dividende
            << " by 0!"
            << std::endl;
}
```

VS

```
void divide_by(int Diviseur,
               int dividende)
{
  if (Diviseur == 0)
    std::cout << (dividende/Diviseur)
              << std::endl;
  else
    std::cerr << "Cannot divide "
              << dividende
              << " by 0!"
              << std::endl;
}
```

A votre avis, quelle différence?

Le test Diviseur==0 est **dynamique**
Il sera fait à chaque appel de la fonction



```
void divide_by(int Diviseur,
               int dividende)
{
  if (Diviseur == 0)
    std::cout << (dividende/Diviseur)
              << std::endl;
  else
    std::cerr << "Cannot divide "
              << dividende
              << " by 0!"
              << std::endl;
}
```

Pourquoi faire simple quand on peut faire compliqué

Université Gustave Eiffel

```
template <int Diviseur>
void divide_by(int dividende)
{
  std::cout << (dividende/Diviseur)
            << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
  std::cerr << "Cannot divide "
            << dividende
            << " by 0!"
            << std::endl;
}
```

Le test Diviseur==0 est **statique,**

- Il n'est fait qu'une fois par le compilo
- La valeur de Diviseur doit être connu au moment de la compilation

Pourquoi faire simple quand on peut faire compliqué

Université Gustave Eiffel

```cpp
template <int Diviseur>
void divide_by(int dividende)
{
  std::cout << (dividende/Diviseur)
            << std::endl;
}

template <>
void divide_by<0>(int dividende)
{
  std::cerr << "Cannot divide "
            << dividende
            << " by 0!"
            << std::endl;
}
```

VS

```cpp
void divide_by(int Diviseur,
               int dividende)
{
  if (Diviseur == 0)
    std::cout << (dividende/Diviseur)
              << std::endl;
  else
    std::cerr << "Cannot divide "
              << dividende
              << " by 0!"
              << std::endl;
}
```

🤨 Des questions?

On peut spécialiser une classe-template de trois manières différentes :

- Spécialisation totale de classe-template.

```
template <>
class std::vector<bool> { … }
```

- Spécialisation partielle de classe-template.

```
template <typename Key>
class std::map<Key, bool> { … }
```

- Spécialisation de fonction-membre (forcément totale)

1. Plages d'éléments et algorithmes de la STL
2. Foncteurs et Lambdas
3. Templates
4. Spécialisation de template
5. **SFINAE**
6. constexpr

SFINAE = **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror

SFINAE = **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror

Ok… mais qu'est-ce que ça veut dire?

- Permettre au compilateur d'échouer à instancier des templates:
  - Le compilateur essaie d'instancier un template
  - S'il réussit, parfait
  - S'il échoue, pas grave, il essaie une autre instanciation

SFINAE = **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror

Ok… mais qu'est-ce que ça veut dire?

- Permettre au compilateur d'échouer à instancier des templates:
  - Le compilateur essaie d'instancier un template
  - S'il réussit, parfait
  - S'il échoue, pas grave, il essaie une autre instanciation
- Plus ou moins un **if-then-else statique**
  - La branche non choisie n'a pas besoin de compiler !
  - Plusieurs implémentations possibles d'une fonction, et le compilateur choisi statiquement la meilleure !

Détecter un type avec le SFINAE

Université Gustave Eiffel

```
struct HasNothing {};

struct HasValueType {
  using value_type = void;};
```

On a deux types

On veut que `my_function` fasse quelque chose de différent suivant s'il y a un value_type

```
int main() {
    HasValueType a;
    my_function(a);
    HasNothing b;
    my_function(b);
}
```

Détecter un type avec le SFINAE

Université Gustave Eiffel

```cpp
struct HasNothing {};

struct HasValueType {
    using value_type = void;};
```

Paramètre template

Type de retour

```cpp
template <typename T>
typename T::value_type
my_function(const T& object)
{
    std::cout << "J'ai un value_type :)" << std::endl;
}
```

```cpp
int main() {
    HasValueType a;
    my_function(a);
    HasNothing b;
    my_function(b);
}
```

Détecter un type avec le SFINAE


```
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

Paramètre template

Type de retour

```
template <typename T>
typename T::value_type
my_function(const T& object)
{
   std::cout << "J'ai un value_type :)" << std::endl;
}
```

myfunction<HasValueType> compile !

```
int main() {
   HasValueType a;
   my_function(a);
   HasNothing b;
   my_function(b);
}
```

Détecter un type avec le SFINAE

Université Gustave Eiffel

```
struct HasNothing {};

struct HasValueType {
    using value_type = void;};
```

Paramètre template

Type de retour

```
template <typename T>
typename T::value_type
my_function(const T& object)
{
    std::cout << "J'ai un value_type :)" << std::endl;
}
```

myfunction<HasValueType> compile !

myfunction<HasNothing> ne compile **pas**!
car HasNothing::value_type n'existe pas

```
int main() {
    HasValueType a;
    my_function(a);
    HasNothing b;
    my_function(b);
}
```

Détecter un type avec le SFINAE


```
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

Donc on crée une autre `my_function` pour le "cas de base"

```
template <typename T>
void
my_function(const T& object)
{
   std::cout << "Pas de value_type :(" <<std::endl;
}
```

```
int main() {
   HasValueType a;
   my_function(a);
   HasNothing b;
   my_function(b);
}
```

Détecter un type avec le SFINAE

Université Gustave Eiffel

```cpp
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

```cpp
template <typename T>
void
my_function(const T& object)
{
   std::cout << "Pas de value_type :(" <<std::endl;
}
```

```cpp
template <typename T>
typename T::value_type
my_function(const T& object)
{
   std::cout << "J'ai un value_type :)" << std::endl;
}
```

```cpp
int main() {
   HasValueType a;
   my_function(a);
   HasNothing b;
   my_function(b);
}
```

On a maintenant un problème !
Le voyez-vous?

Détecter un type avec le SFINAE


```cpp
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

```cpp
template <typename T>
typename T::value_type
my_function(const T& object)
{
   std::cout << "J'ai un value_type :)" << std::endl;
}
```

lui ou lui??

```cpp
template <typename T>
void
my_function(const T& object)
{
   std::cout << "Pas de value_type :(" <<std::endl;
}
```

Ambigu !!

```cpp
int main() {
   HasValueType a;
   my_function(a);
   HasNothing b;
   my_function(b);
}
```

On a maintenant un problème !
Le voyez-vous?

Détecter un type avec le SFINAE

Université Gustave Eiffel

```
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

```
template <typename T>
typename T::value_type
my_function(const T& object, unsigned)
{
   std::cout << "J'ai un value_type :)" << std::endl;
}
```

```
template <typename T>
void
my_function(const T& object, int)
{
   std::cout << "Pas de value_type :(" <<std::endl;
}
```

```
int main() {
   HasValueType a;
   my_function(a, 0u);
   HasNothing b;
   my_function(b, 0u);
}
```

Astuce sordide pour résoudre le problème

Détecter un type avec le SFINAE

Université Gustave Eiffel

```cpp
struct HasNothing {};

struct HasValueType {
   using value_type = void;};
```

```cpp
template <typename T>
void
my_function(const T& object, int)
{
   std::cout << "Pas de value_type :(" <<std::endl;
}
```

```cpp
template <typename T>
typename T::value_type
my_function(const T& object, unsigned)
{
   std::cout << "J'ai un value_type :)" << std::endl;
}
```

```cpp
int main() {
   HasValueType a;
   my_function(a, 0u);
   HasNothing b;
   my_function(b, 0u);
}
```

🤨 Des questions?

Au lieu d'une astuce sordide on désambiguïsera plutôt avec:

- **La spécialisation** : le compilateur va essayer la spécialisation la plus précise en premier !
- **Les outils de la STL dans <type_traits>** : `std::enable_if` etc.
- **La construction `if constexpr`** que l'on va voir juste après
- Une implémentation propre de file de priorité (tp11)

1. Plages d'éléments et algorithmes de la STL
2. Foncteurs et Lambdas
3. Templates
4. Spécialisation de template
5. SFINAE
6. **Constexpr**

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

```
struct MyClass {
    constexpr static unsigned i = 0;

    inline static unsigned j = 0;

    constexpr unsigned i = 0;
};
```

Université Gustave Eiffel

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

Ceci est calculé par le compilateur, sa valeur pourra être utilisé pendant la compilation

```
struct MyClass {
  constexpr static unsigned i = 0;

   inline static unsigned j = 0;

   constexpr unsigned i = 0;
};
```

Université Gustave Eiffel

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

Ceci est calculé par le compilateur, sa valeur pourra être utilisé pendant la compilation

```
struct MyClass {
  constexpr static unsigned i = 0;

  inline static unsigned j = 0;

    constexpr unsigned i = 0;
};
```

Ceci n'est pas calculé au moment de la compilation

Université Gustave Eiffel

Le mot-clef **constexpr** déclare au compilateur que quelque chose doit être calculé à la compilation.

Ceci est calculé par le compilateur, sa valeur pourra être utilisé pendant la compilation

```
struct MyClass {
  constexpr static unsigned i = 0;

  inline const static unsigned j = 0;

    constexpr unsigned i = 0;
};
```

Ceci n'est pas calculé au moment de la compilation (même `const`)

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

Ceci est calculé par le compilateur, sa valeur pourra être utilisé pendant la compilation

```
struct MyClass {
  constexpr static unsigned i = 0;

  inline const static unsigned j = 0;

  constexpr unsigned i = 0;
};
```

Ceci n'est pas calculé au moment de la compilation (même `const`)

Ceci n'a aucun sens ! Pourquoi?

Université Gustave Eiffel

Le mot-clef **`constexpr`** déclare au compilateur que quelque chose doit être calculé à la compilation.

Ceci est calculé par le compilateur, sa valeur pourra être utilisé pendant la compilation

```
struct MyClass {
  constexpr static unsigned i = 0;

  inline const static unsigned j = 0;

  constexpr unsigned i = 0;
};
```

Ceci n'est pas calculé au moment de la compilation (même `const`)

Ceci n'a aucun sens !

Des questions?

If constexpr


```cpp
struct WithTrait {
  constexpr static bool trait = true;
};
```

```cpp
struct NoTrait {
  constexpr static bool trait = false;
};
```

```cpp
template<typename T>
void my_function(const T& object) {
  if constexpr (T::trait)
    std::cout << "Yes !" << std::endl;
  else
    std::cout << "Bouhouh :(" << std::endl;
}
```

If constexpr

Université Gustave Eiffel

```
struct WithTrait {
  constexpr static bool trait = true;
};
```

```
struct NoTrait {
  constexpr static bool trait = false;
};
```

```
template<typename T>
void my_function(const T& object) {
  if constexpr (T::trait)
    std::cout << "Yes !" << std::endl;
  else
    std::cout << "Bouhouh :(" << std::endl;
}
```

Instanciation par le compilo

```
void my_function<WithTrait>(..) {
    std::cout << "Yes !" << std::endl;
}
```

```
void my_function<NoTrait>(..) {
    std::cout << "Bouhouh :(" << std::endl;
}
```

Ok mais pourquoi faire?

- Efficacité: le test est fait une fois par le compilateur
- La partie du code inutile n'a pas besoin de compiler

Ok mais pourquoi faire?

- Efficacité: le test est fait une fois par le compilateur
- La partie du code inutile n'a pas besoin de compiler

```
struct WithTrait {
    constexpr static bool trait = true;
    void f() { /* .. */ }
};
```

```
struct NoTrait {
    constexpr static bool trait = false;
    void g() { /* .. */ }
};
```

```
template<typename T>
void my_function(const T& object) {
    if constexpr (T::trait)
        object.f();
    else
        object.g(); }
```

Ok mais pourquoi faire?

- Efficacité: le test est fait une fois par le compilateur
- La partie du code inutile n'a pas besoin de compiler

```cpp
struct WithTrait {
    constexpr static bool trait = true;
    void f() { /* .. */ }
};
```

```cpp
struct NoTrait {
    constexpr static bool trait = false;
    void g() { /* .. */ }
};
```

```cpp
template<typename T>
void my_function(const T& object) {
    if constexpr (T::trait)
        object.f();
    else
        object.g(); }
```

Ne compile pas pour T=WithTrait

Ne compile pas pour T=NoTrait

Un polymorphisme **statique** plus flexible:

- On regroupe les classes par ce qui les rassemble
  Ex: les conteneurs
- Ils ont une interface commune
  Ex: `begin()`, `end()`…
- Chaque classe a des traits qui indique des propriétés
  Ex: random access, bidirectionnel
- Spécialisation d'algorithme en fonction de ces traits
- Pas besoin d'implémenter les parties non-pertinentes de l'interface
  Ex: List en java

Ce qu'on va apprendre dans le segment 3

- Les algorithmes de la STL
- Lambdas
- Création de templates
- Mieux comprendre les messages d'erreurs du compilateurs

Et un peu de

- If constexpr
- Référence universelle
- Spécialisation
- SFINAE