



ÇA Y EST,  
ON ARRIVE !  
QU'EST-CE QUE  
TU VOIS ?!

# Cours de C++

## Segment 3

2024/2025

"TEMPLATES"...  
"LAMBDA"...  
"SFINAE" !??  
J'AI PEUR !!!

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 de données associée à un conteneur, on utilise

- les fonctions-libres **std::begin** et **std::end**; ou bien
- les fonctions-membres **begin** et **end**.

```
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));
```

(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));
```

(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));
```

(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));
```

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. On peut aussi appeler une fonction-membre ou accéder à un attribut avec l'**operator->** (comme pour un pointeur).

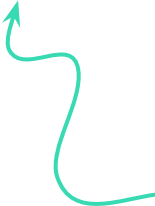
```
for (auto it = it_middle; it != it_end; ++it)
{
    auto& element = *it;
    ...
}
```



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. On peut aussi appeler une fonction-membre ou accéder à un attribut avec l'**operator->** (comme pour un pointeur).

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



it\_end pointe **après**  
le dernier élément de  
la plage

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)  
{  
    ...  
}
```

équivalent à

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

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>`.

`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  
}
```

`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  
}
```

fin de plage

début de plage

valeur à trouver

`std::find`  
recherche un élément équivalent

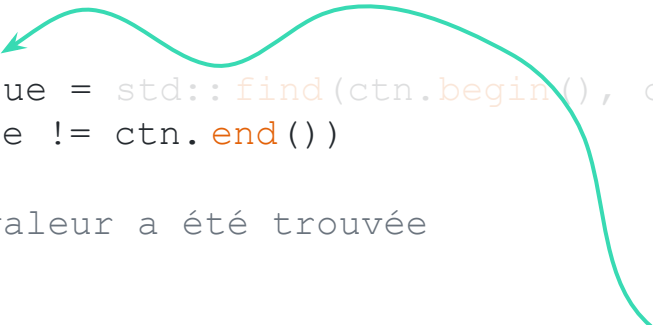
```
auto it_value = std::find(ctn.begin(), ctn.end(), value_to_find);  
if (it_value != ctn.end())  
{  
    // la valeur a été trouvée  
}
```

## Contrainte

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

`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  
}
```



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

`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 valeur  
}
```

début de plage

prédicat

`std::find_if`  
recherche un élément vérifiant un prédicat

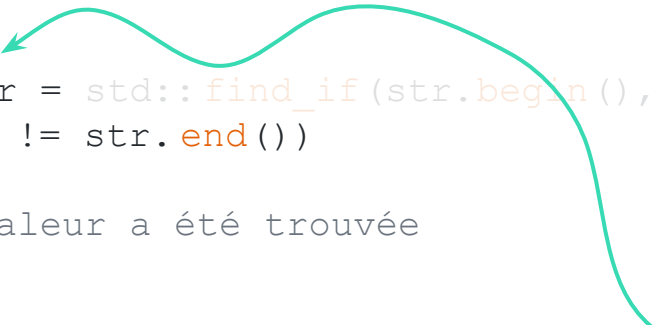
```
auto it_char = std::find_if(str.begin(), str.end(), is_lowercase);  
if (it_char != str.end())  
{  
    // la valeur a été trouvée  
}
```

## 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

`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);
```

`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);
```

fin de plage

début 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`

`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());
```



`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());
```

fin de plage

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`

`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

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.

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

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

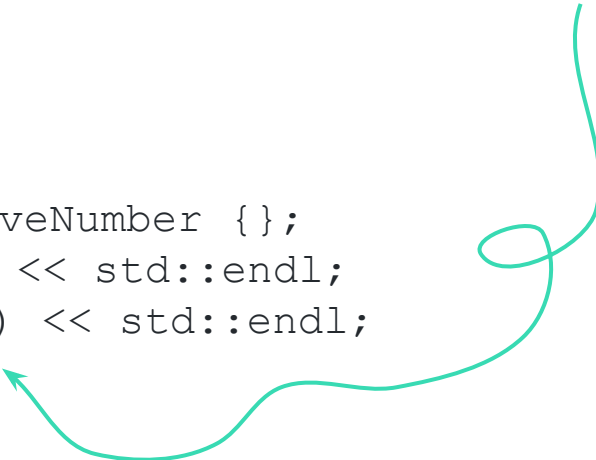
```
struct IsPositiveNumber {  
    bool operator()(int nb) const  
    {  
        return nb >= 0;  
    }  
};
```

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

```
struct 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()



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;

equals_3._value = 5;
std::cout << equals_3(3) << std::endl;
```

```
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(), [?????]);
}
```

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

On peut utiliser notre foncteur `EqualValue` ...

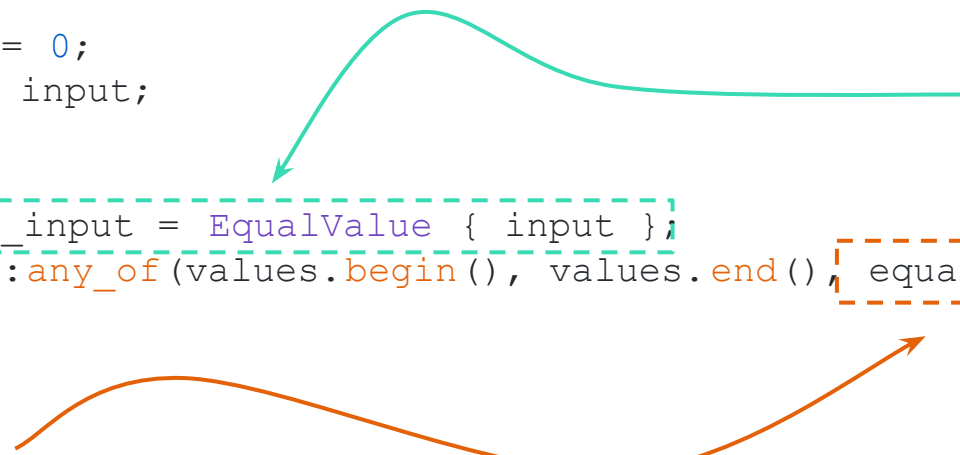
```
class EqualValue
{
public:
    EqualValue(int value) : _value{value}
    {}

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

... puis l'instancier pour l'utiliser dans l'appel à `any_of`

```
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);
}
```



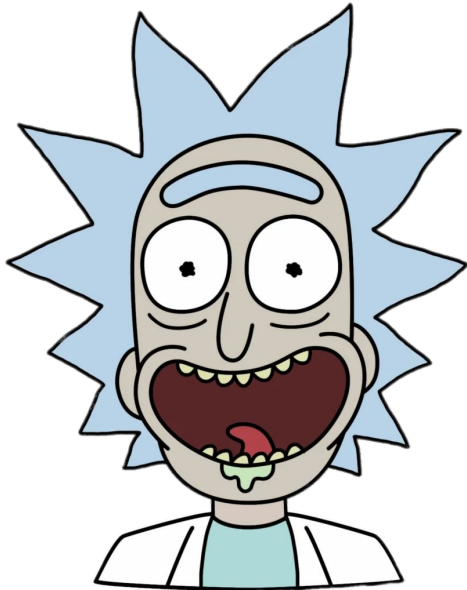
On instancie  
notre foncteur

Et on passe à `std::any_of`

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

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; }
```

Capture

Paramètres

Corps

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; }
```

Capture

Paramètres

Corps

Attributs du foncteur

Signature de l'operator ()

Corps de l'operator ()

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; }
    );
}
```



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]`

```
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; }
    );
}
```

```
bool has_any_equal_to_input(const std::vector<int>& values)
```

```
{
```

```
    int input = 0;  
    std::cin >> input;
```

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

```
}
```

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(); }
```

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;
};
```

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

Type de retour



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`

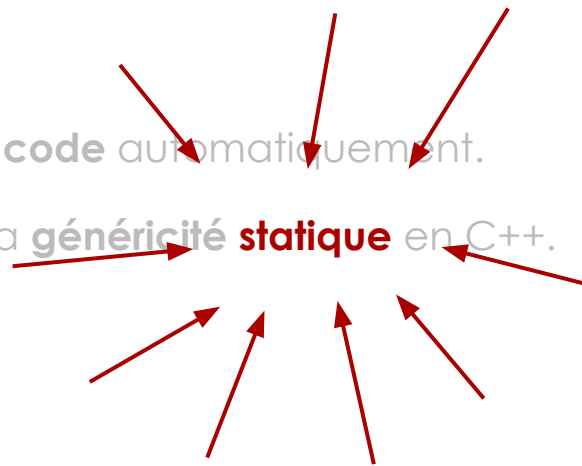
## C'est quoi un template ?

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`



## C'est quoi un template ?

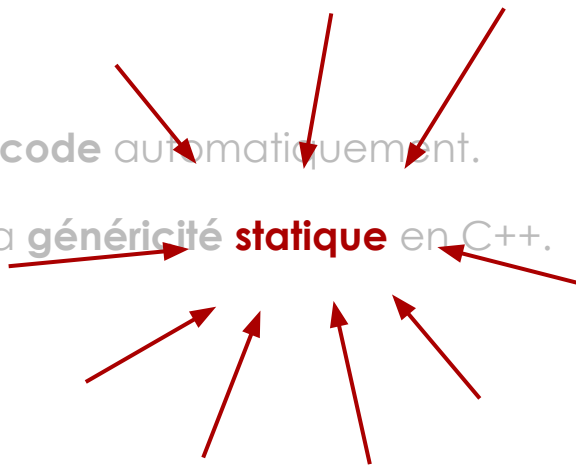
Un template, ou patron, est un modèle qui sert à **générer du code** automatiquement.

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`

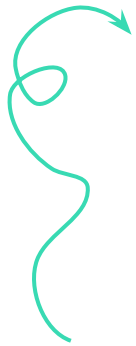
**But:** Faire bosser le compilateur



## 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;
}
```

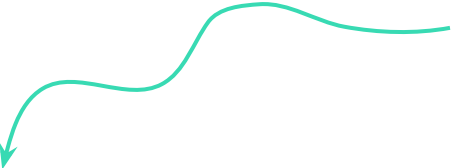
Mot-clé utilisé pour indiquer  
qu'on crée un template.



## Syntaxe

Liste de paramètres du template.

Ici: un paramètre qui est un `typename`



```
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;  
}
```



Nom du template.

## Instanciation

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

Par exemple, `std::min<int>` et `std::min<std::string>` sont deux instances différentes du template `std::min`.

Pour instancier une fonction-template, on peut simplement appeler une instance de la fonction-template. Le compilateur va automatiquement instancier la fonction depuis le modèle lorsqu'il verra la ligne correspondante dans le code.

### **Attention !**

Il faut que le compilateur ait vu le template pour pouvoir en faire une instanciation.

```
#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");
}
```

```
#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");
```

```
}
```

```
#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");
```

```
}
```



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

```
#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");
}
```



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

```
#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**



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

```
#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");  
}
```

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

≡ **Instanciation**

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

```
#include <iostream>
```

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

```
int main()  
{  
    print_between_parentheses  
}
```

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

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

≡ **Instanciation**

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.

```
#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 compilateur peut le  
**déduire**



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.


```
#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?



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.

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

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

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

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

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

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

```
#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 ?

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

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



NON !



```
#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 ?

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

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

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



OUI !

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

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

COMPILO



**template** <typename  
T min(T v1, T v2)

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

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

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

int

int

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

T = int

T = int

COMPILO



OUI !

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

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

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



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

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

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

```
#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 ?

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

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



NON !



```
#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 ?

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

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

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



OUI !

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

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



**template** <typename  
T min(T v1, T v2)

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

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

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

double

int

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

T = double

T = int

COMPILO



NON !

```
#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);  
    |     ~~~~~^~~~~~
```



## Syntaxe

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

## Syntaxe

mot-clef pour indiquer  
qu'on définit un template

```
template <typename T>
```

```
struct Fraction
```

```
{
```

nom du template

```
    idende
```

```
    iseur = {};
```

```
};
```

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, afin d'instancier une classe-template, on peut utiliser une instance du template. 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**.



```
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  
dans sa base de données

```
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);
}
```

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

```
template <typename T>
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

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

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

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

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

*“Faut-il mettre le code dans le hpp ou un cpp séparé?”*

*C'est mieux pour la lisibilité, mais:*

- 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
  - 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;
}
```

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;
}
```

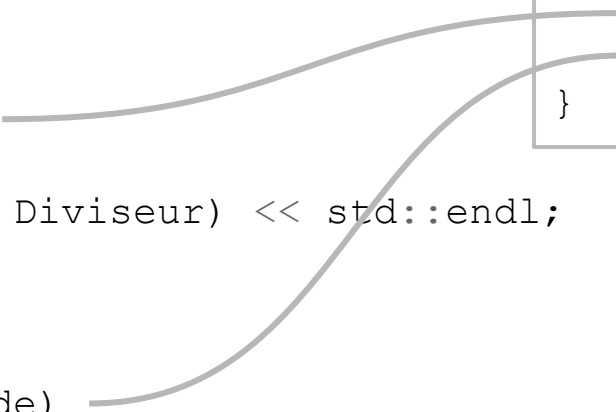
spécialisation pour <0>

Une spécialisation permet, pour une liste d'arguments de template spécifiques, d'utiliser un autre modèle de code que celui par défaut 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;
}
```

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



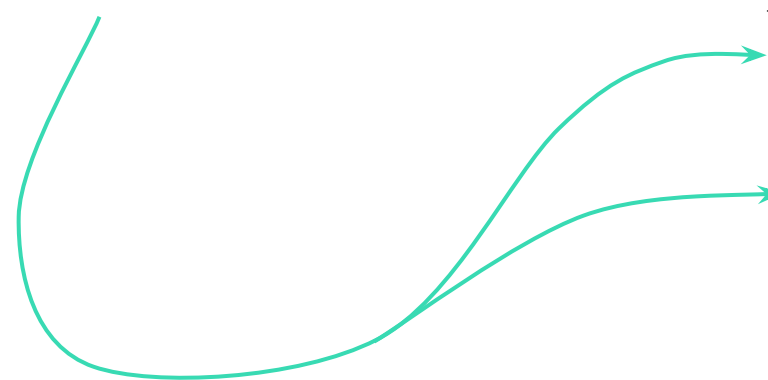
## VS

```
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;
}
```

```
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;
}
```

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;  
}
```

```
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 compilateur
- La valeur de `Diviseur` doit être connue au moment de la compilation

Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1 } + str2;
```

```
}
```

Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

on n'a plus aucun paramètre de template

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1 } + str2;
```

```
}
```



Lorsqu'on spécialise une fonction-template, il faut indiquer les valeurs pour **TOUS** les paramètres de template. On parle de **spécialisation totale**.

```
template <typename R, typename T1, typename T2>
```

```
R add(T1 v1, T2 v2)
```

```
{
```

```
    return static_cast<R>(v1 + v2);
```

```
}
```

```
template <>
```

```
std::string add<std::string, const char*, const char*>(const char* str1,  
                                                         const char* str2)
```

```
{
```

```
    return std::string { str1
```

```
}
```

on a bien spécifié les valeurs  
des 3 paramètres ici

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

- Spécialisation totale de classe-template.
- Spécialisation partielle de classe-template.
- Spécialisation de fonction-membre (forcément totale, puisque c'est une spécialisation de fonction)

Lorsqu'on spécialise une classe-template, il faut réécrire **l'intégralité** de la classe-template, pas uniquement les morceaux qui nous intéressent (attributs + fonctions-membres + implémentation de ces fonctions).

Cela permet d'adapter le type d'attributs à un cas donné (par exemple, pour optimiser le code, ou gérer des cas particulier). `std::vector<bool>` est une spécialisation de `std::vector`, car l'implémentation peut-être optimisée en passant par des masques de bits.

**Exemple :** <https://godbolt.org/z/GG4ed91Kc>

On peut également faire des spécialisations **partielles**.

Contrairement aux spécialisations totales dans lesquelles on s'attend à ce que les premiers chevrons de la spécialisation soient vides, on peut laisser une partie des paramètres non spécifiés, voire en utiliser certains pour construire les paramètres finaux du template.

Par exemple, « je veux que ma spécialisation concerne tous les `std::vector<qqch>` », `qqch` est un paramètre de template de la spécialisation.

**Exemple 1 :** <https://godbolt.org/z/vczrsfeqP>

**Exemple 2 :** <https://godbolt.org/z/TM1W7shGz>

On peut également spécialiser seulement certaines fonctions d'une classe-template. Dans ce cas, la spécialisation doit être totale, puisqu'il s'agit d'une spécialisation de fonctions-template.

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 Is **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
- Implémentation d'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 !

```
struct HasNothing {};
```

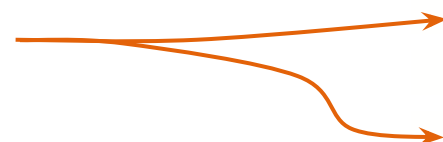
```
struct HasValueType {  
    using value_type = void;};
```

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);  
}
```



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

```
template <typename T>
```

Paramètre template

```
typename T::value_type
```

Type de retour

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

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

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

```
template <typename T>
```

Paramètre template

```
typename T::value_type
```

Type de retour

```
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);  
}
```

```
struct HasNothing {};
```

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

```
template <typename T>
```

```
typename T::value_type
```

```
my_function(const T& object)
```

```
{
```

```
    std::cout << "J'ai un value_type :)" << std::endl;
```

```
}
```

Paramètre template

Type de retour

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);  
}
```

```
struct HasNothing {};
```

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

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

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



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

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

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

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

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

**Voyez-vous le problème ?**

```
struct HasNothing {};
```

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

```
template <typename T>
```

```
void
```

```
my_function(const T& object)
```

```
{
```

```
    std::cout << "Pas de value_type :(" << std::endl;
```

```
}
```

```
template <typename T>
```

```
typename T::value_type
```

```
my_function(const T& object)
```

```
{
```

```
    std::cout << "J'ai un value_type :)" << std::endl;
```

```
}
```

lui  
ou  
lui??

**Ambigu !!**

```
int main() {
```

```
    HasValueType a;
```

```
    my_function(a);
```

```
    HasNothing b;
```

```
    my_function(b);
```

```
}
```

**Voyez-vous le problème ?**



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

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

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

**Astuce sordide pour résoudre le problème**

Ici, on utilise une astuce sordide pour désambiguïser, usuellement, on utilisera plutôt:

- **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

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.

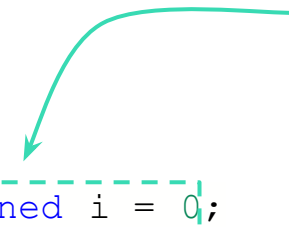
Le mot-clef **constexpr** déclare au compilateur que quelque chose doit être calculé à la compilation.

```
struct MyClass {  
    constexpr static unsigned i = 0;  
  
    inline static unsigned j = 0;  
  
    constexpr unsigned i = 0;  
};
```

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;  
};
```



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

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 !

```
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 (object.trait)  
        std::cout << "J'ai le trait :)" << std::endl;  
    else  
        std::cout << "Je n'ai pas le trait :(" << std::endl;  
}
```

**Instanciation  
par le compilateur**



```
void my_function<WithTrait>(..) {  
    std::cout << "J'ai le trait :)" << std::endl;  
}
```

```
void my_function<NoTrait>(..) {  
    std::cout << "J'ai le trait :)" << std::endl;  
}
```

```
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 (object.trait)  
        std::cout << "J'ai le trait :)" << std::endl;  
    else  
        std::cout << "Je n'ai pas le trait :(" << std::endl;  
}
```

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

- 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 (object.trait)
        object.f();
    else
        object.g(); }
```

- 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 (object.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()`, `plages`, etc.
- Chaque classe a des traits qui indique des propriétés  
Ex: `random access`, `bidirectionnel`
- Spécialisation d'algorithme en fonction de ces traits

- 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