



OH NON ÇA Y EST  
ON EST FOUTUS !!  
QU'EST-CE QUE  
TU VOIS ?!

# Cours de C++ Master 1

2023 / 2024

Y'A ÉCRIT  
"TEMPLATES"...  
"LAMBDA"...  
J'AI PEUR !!!

1. Conteneurs.
2. Plages d'éléments et opérations.
3. Templates.
4. Bonne pratiques.

## 1. Conteneurs

- a. Conteneurs séquentiels
- b. Conteneurs associatifs
- c. Tuples

2. Plages d'éléments et opérations.

3. Templates.

4. Bonne pratiques.

Un **conteneur séquentiel** est un conteneur dans lequel les éléments sont stockés dans un **ordre bien défini**, de telle sorte que les notions de **premier élément** et de **n-ième élément** aient un sens.

Par exemple :

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

Pour accéder à l'élément à la  $i$ -ème position :

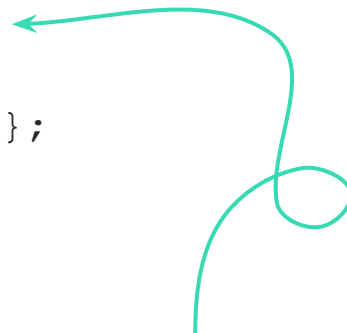
1. via l'opérateur `[]` du conteneur, s'il est disponible

```
auto vec = std::vector<...> { ... };  
std::cout << vec[14] << std::endl;
```

Pour accéder à l'élément à la  $i$ -ème position :

1. via l'operator `[]` du conteneur, s'il est disponible

```
auto vec = std::vector<...> { ... };  
std::cout << vec[14] << std::endl;
```



Un conteneur disposant d'un  
operator[] (entier) est un  
**conteneur à accès aléatoire**

Pour accéder à l'élément à la  $i$ -ème position :

1. via l'opérateur `[]` du conteneur, s'il est disponible

```
auto vec = std::vector<...> { ... };  
std::cout << vec[14] << std::endl;
```

2. via la fonction `std::next` sinon

```
auto list = std::list<...> { ... };  
auto it_12 = std::next(list.begin(), 12);  
std::cout << *it_12 << std::endl;
```

Pour accéder à l'élément à la  $i$ -ème position :

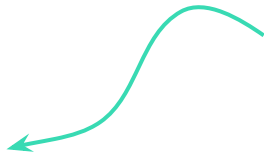
1. via l'operator `[]` du conteneur, s'il est disponible

```
auto vec = std::vector<...> { ... };  
std::cout << vec[14] << std::endl;
```

2. via la fonction `std::next` sinon

```
auto list = std::list<...> { ... };  
auto it_12 = std::next(list.begin(), 12);  
std::cout << *it_12 << std::endl;
```

`std::next` incrémente un itérateur d'une valeur donnée





Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.



cela signifie que les éléments peuvent avoir été déplacés en mémoire

Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

```
➔ auto vec = std::vector<int> { 1, 2, 3, 4 };  
   vec.erase(std::next(vec.begin(), 2));
```



Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

```
➔ auto vec = std::vector<int> { 1, 2, 3, 4 };  
   vec.erase(std::next(vec.begin(), 2));
```



vector::erase invalide l'itérateur courant et tous les suivants

Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

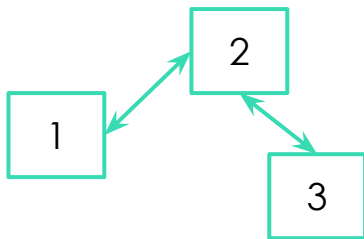
```
➔ auto vec = std::vector<int> { 1, 2, 3, 4 };  
   vec.erase(std::next(vec.begin(), 2));
```



vector::erase invalide l'itérateur courant et tous les suivants

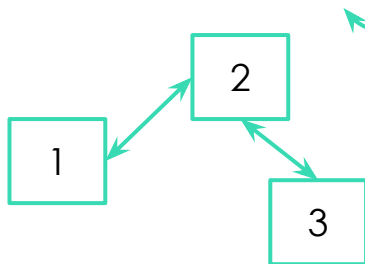
Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

```
➡ auto list = std::list<int> { 1, 2, 3 };  
   list.erase(std::next(list.begin()), 1));
```



Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

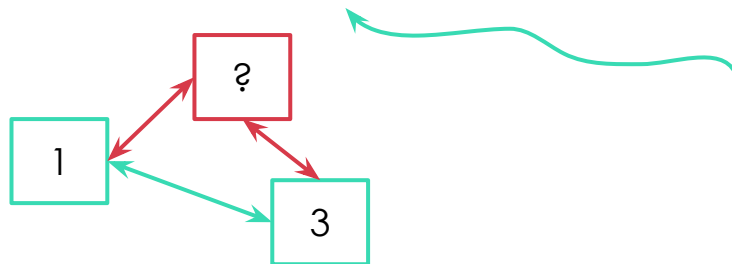
```
➡ auto list = std::list<int> { 1, 2, 3 };  
   list.erase(std::next(list.begin()), 1));
```



list::erase invalide uniquement  
l'itérateur courant

Lorsqu'on réalise des opérations d'**insertion** ou de **suppression** sur un conteneur séquentiel, il faut vérifier si ces opérations **invalident les itérateurs existants**.

```
auto list = std::list<int> { 1, 2, 3 };  
list.erase(std::next(list.begin()), 1);
```



list::erase invalide uniquement  
l'itérateur courant



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

Cette indexation peut-être réalisée soit au moyen du **tri** des clés, soit au moyen de leur **hashage**.

Par exemple :

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

## Indexation par tri

Accès:  $O(\log n)$

Insertion:  $O(\log n)$

Suppression:  $O(\log n)$

Contraintes sur les clés:

- comparables

## Indexation par hashage

Accès:  $O(1)$  amorti

Insertion:  $O(1)$  amorti

Suppression:  $O(1)$  amorti

Contraintes sur les clés:

- équivalences
- hashables

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

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

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

Indexation par tri

Indexation par hashage

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

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

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

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

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

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

Indexation par tri

Indexation par hashage

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

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

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

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

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

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

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

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



1. Conteneurs
- 2. Plages de données et opérations.**
  - a. Plages de données
  - b. Opérations usuelles
  - c. Lambdas
3. Templates.
4. Bonne pratiques.

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());  
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());  
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());  
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());  
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**.

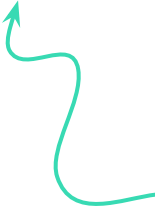
L'élément courant est obtenu en utilisant l'**operator\*** de l'itérateur. On peut aussi appeler une fonction-membre ou accéder à un attribut avec l'**operator->** (comme pour un pointeur).

```
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 bibliothèque 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 ete trouvee  
}
```

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

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

fin de plage

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

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

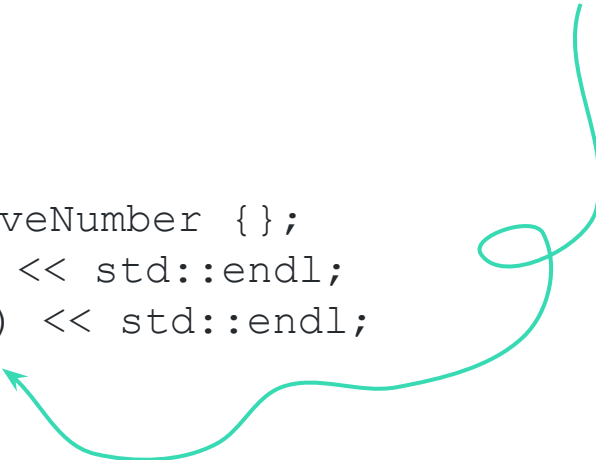
```
auto functor = IsPositiveNumber {};
std::cout << functor(6) << std::endl;
std::cout << functor(-3) << std::endl;
```



```
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.

```
struct EqualValue
{
    int value = 0;

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

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

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

Comment faire pour utiliser le contenu de la variable locale `input` à l'intérieur du prédicat ?

On peut définir un foncteur...

```
struct EqualValue
{
    int value = 0;

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

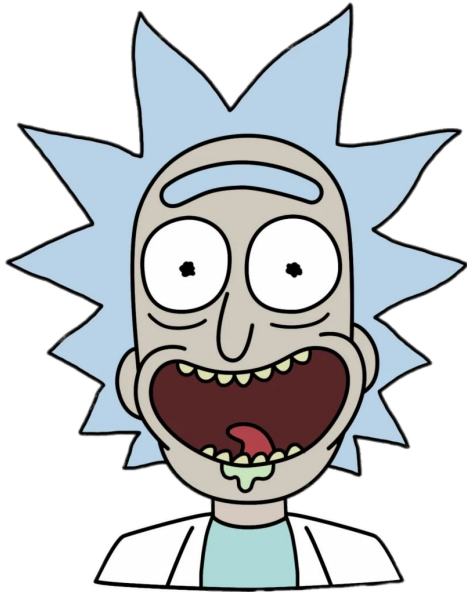
... puis l'instancier pour l'utiliser dans l'appel à `any_of`

```
bool has_any_greater_than_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 sacrément **verbeux** !!!





Heureusement, les **lambdas** sont là !

Une **lambda** est un foncteur instancié en **une seule instruction** via la syntaxe suivante :

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



Une **lambda** est un foncteur instancié en **une seule instruction** via la syntaxe suivante :

Paramètres

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

Une **lambda** est un foncteur instancié en **une seule instruction** via la syntaxe suivante :

Corps

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

Une **lambda** est un foncteur instancié en **une seule instruction** via la syntaxe suivante :

Capture

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

- la capture permet de générer et d'initialiser les attributs du foncteur
- les paramètres permettent de définir la signature de l'`operator()`
- le corps de la lambda produit l'implémentation de l'`operator()`

On peut ainsi réécrire le code suivant :

```
struct EqualValue
{ ... };

bool has_any_greater_than_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_greater_than_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]`

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(); }
```



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 {
    Type de retour 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. Conteneurs
2. Plages d'éléments et opérations.
- 3. Templates.**
  - a. Fonctions-template
  - b. Classes-template
  - c. Spécialisations
4. Bonne pratiques.

## C'est quoi un template ?

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

On a des fonctions-template, qui permettent de créer des fonctions, et des classes-templates, qui permettent de créer des types.

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

Quelques exemples de templates que vous avez déjà rencontrés :


- Les classes-template `std::vector`, `std::map`, et autres conteneurs.
- Les fonctions-template `std::move`, `std::make_unique` OU `std::min`.

## Syntaxe

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



## Syntaxe

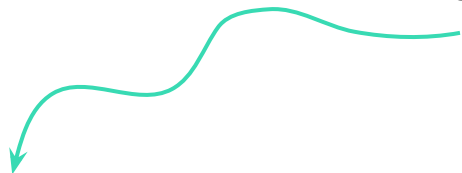


```
template <typename T>  
void print_between_parentheses( const T& value)  
{  
    std::cout << '(' << value << ')' << std::endl;  
}
```

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

## Syntaxe

Liste de paramètres  
du template.



```
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");  
}
```



```
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("Hello");  
}
```

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.

```
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;
}
```



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;
}
```



```
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;
}
```



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  
{  
    idende  
    iseur = 1;  
};
```

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

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);
}
```

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>

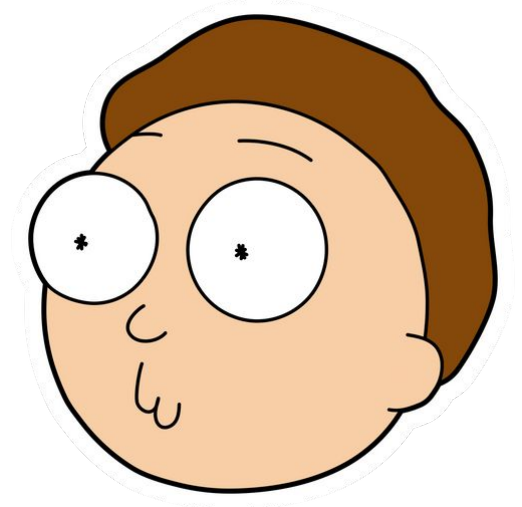
**Exemple 2** : <https://godbolt.org/z/TM1W7shGz>

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. Conteneurs
2. Plages d'éléments et opérations.
3. Templates.
- 4. Bonne pratiques.**
  - a. Expressivité
  - b. Uniformisation
  - c. Compilation
  - d. Autres bonnes pratiques
  - e. Synthèse

Selon vous, que fait ce programme ?

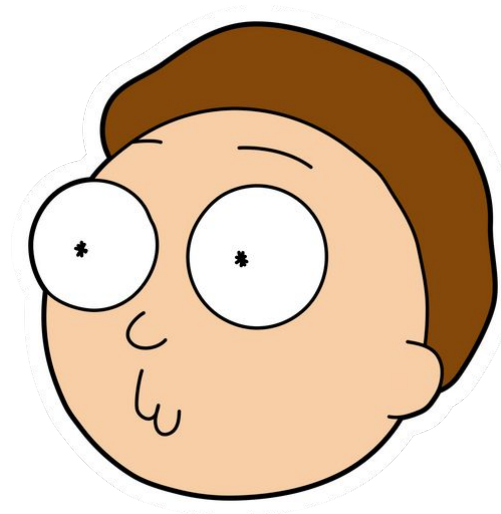
```
int main()  
{  
    auto a = 0;  
    auto b = std::string {};  
  
    std::cin >> b >> a;  
    std::cout << fcn(a, b) << std::endl;  
}
```



Selon vous, que fait ce programme ?

```
int main()
{
    auto a = 0;
    auto b = std::cout >> b >> a;
    std::cout << fcn(a, b) << std::endl;
}
```

**ON SAIT PAS C'EST TOUT MAL NOMME**





## EXPRESSIVITE ++

```
int main()  
{  
    auto repetition_count = 0;  
    auto word_to_repeat = std::string {};  
  
    std::cin >> word_to_repeat >> repetition_count;  
  
    std::cout << repeat_word(word_to_repeat, repetition_count)  
              << std::endl;  
}
```

Améliorer l'**expressivité** du code permet de le comprendre plus rapidement.

3 bonnes pratiques à appliquer :

- découper le code en **fonctions**
- **définir des types**, via des alias ou des classes
- **nommer** explicitement les symboles (variables, types et fonctions)

Autres manières de rendre le code plus expressif :

- définir des **opérateurs** pour les opérations arithmétiques par exemple
- créer des variables pour **nommer des conditions**
- définir des **énumérations** pour nommer des entiers

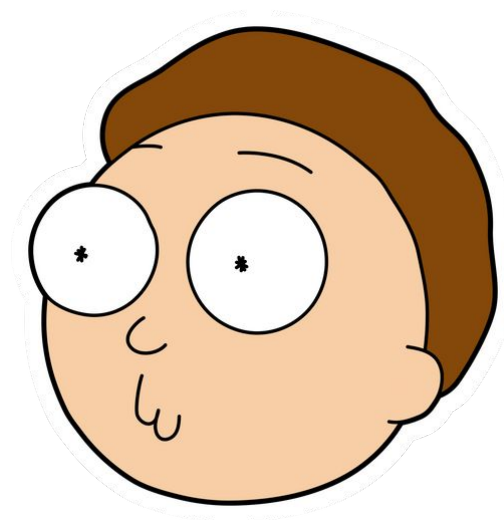
Du code doit pouvoir se comprendre **sans commentaires.**

Si ce n'est pas le cas, vous pouvez le réécrire de manière  
**plus expressive.**

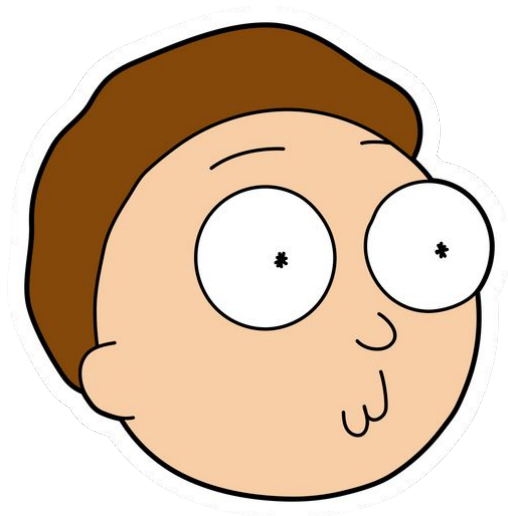
Selon vous, cette fonction fait-elle bien ce qu'elle dit ?

```
int compute Exponent(int nb, int Exp) {  
    if (Exp < 0) return 0;  
    if (Exp == 0) {  
        return 1; }  
  
    int res = nb;  
    while (Exp > 1) {  
        res *= nb;  
    } return res;  
}
```

--Exp;



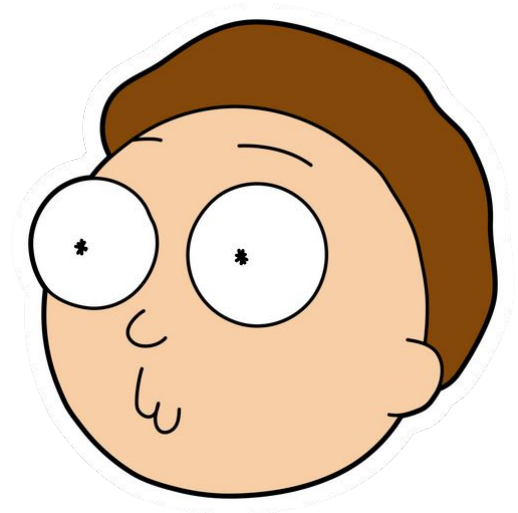
Et celle-ci ?



```
int compute_factorial(int nb) {  
    if (nb < 1) {  
        return 0;  
    }  
  
    int res = nb;  
    while (nb > 1) {  
        --nb;  
        res *= nb;  
    }  
  
    return res;  
}
```

Selon vous, les symboles ci-dessous seraient plutôt des fonctions, des types ou des variables ?

1. `get_value`
2. `Counter`
3. `entity`
4. `create`
5. `animal_3`



Avoir du code écrit de **manière uniforme** permet de le lire et le décrypter **plus rapidement**.

Pour faciliter l'uniformisation du code, on peut mettre en place des **conventions de codage** relative au **style**.



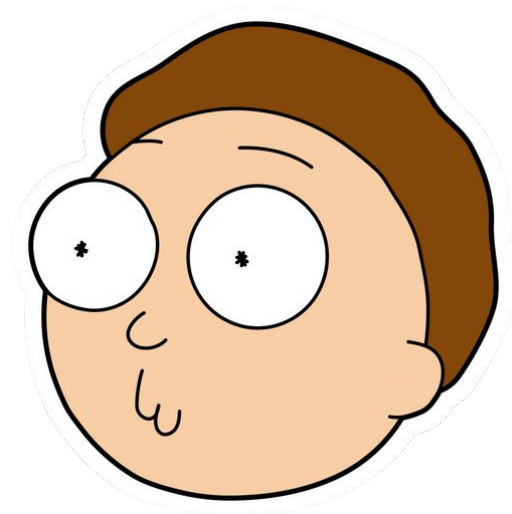
Ce qui définit le **style** :

- PascalCase, camelCase, snake\_case
- Tabs / Spaces
- Saut de ligne
- Indentation
- etc.

Il n'y a pas forcément une convention de style meilleure qu'une autre. Ce qui compte, c'est qu'on utilise les **mêmes conventions** sur **toute la base** de code.

Y a-t-il un problème dans cette fonction ?

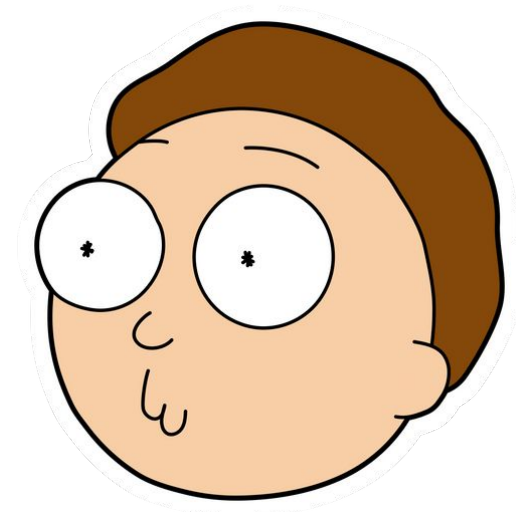
```
void print_reverse(const std::vector<int>& vec)
{
    for (auto i = vec.size(); i >= 0; --i)
    {
        std::cout << vec[i] << std::endl;
    }
}
```



Y a-t-il un problème dans cette fonction ?

```
void print_reverse(const std::vector<int>& vec)
{
    for (auto i = vec.size() - 1; i >= 0; --i)
    {
        std::cout << vec[i] << std::endl;
    }
}
```

**BOUCLE INFINIE A L'EXECUTION**




Un certain nombre d'**erreurs de programmation** peuvent être détectées dès la **compilation**.

Il suffit de fournir les **flags** suivants au compilateur pour qu'il détecte des problèmes potentiels (variables non utilisées, casts implicits, etc) et interrompt la compilation :

```
-Wall -Wextra -Werror
```

Grâce aux flags, le compilateur prévient le programmeur de l'erreur.

```
void print_reverse(const std::vector<int>& vec)
{
    for (auto i = vec.size(); i >= 0; --i)
    {
        std::cout << vec[i] << std::endl;
    }
}
```



**error:** comparison of unsigned expression in '>= 0' is always true

Il existe également des **mots-clés** permettant de s'assurer que le code est utilisé (ou se comportera) comme prévu :  
**const**, **override**, mais aussi **[[nodiscard]]** ou **explicit**.

- Exemple `[[nodiscard]]`  
→ <https://godbolt.org/z/GKsW6xh3z>
- Exemple `explicit`  
→ <https://godbolt.org/z/e79E4389d>



Donnez d'autres exemples de bonnes pratiques



1. 1 fonction = 1 responsabilité
2. écrire les variables toujours en const, et le retirer si besoin ensuite
3. éviter les monolithes: 1 type = 1 responsabilité
4. éviter les fichiers de 10000 lignes
5. éviter les dépendances cycliques
6. const-ref pour éviter les copies
7. ....

Les **bonnes pratiques** permettent de rendre le code :

- plus **lisible**
- plus **compréhensible**
- plus **robuste**
- plus **extensible**
- plus **performant**

Pensez donc bien à :

- améliorer l'**expressivité** de votre code
- définir des **conventions** et les suivre
- utiliser le **compilateur** afin de détecter certaines erreurs de programmation **avant l'exécution**
- appliquer toutes les autres bonnes pratiques que vous connaissez

Vous pouvez consulter les [CppCoreGuidelines](#) qui en référencent beaucoup et les **expliquent** !