

Cours 3

Bonnes Pratiques, entre autres

C++ - Master 1

1. Bonnes Pratiques
2. Exceptions
3. Algorithmes

Donnez quelques exemples de bonnes pratiques



Exemples :

- 1 instr par ligne
- mettre les cas particuliers en valeur avec les indentations
- pas de globales
- bonne nomenclature (bon nommage)
- rajouter la doc
- fonction qui fait peu de choses
- pas trop de caractères sur une même ligne
- override quand on redéfinit
- préconditions
- architecture modulaire

A quoi est-ce que ça sert de les appliquer ?



Les bonnes pratiques permettent de rendre le code :

- plus lisible
- plus compréhensible
- plus robuste
- plus extensible

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 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;  
}
```


Selon vous, que fait ce programme ?

<https://godbolt.org/z/ffxoffqaa>



Selon vous, que fait ce programme ?

<https://godbolt.org/z/PnsPjnr6x>



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

3 bonnes pratiques à appliquer :

- découper le code en petits bouts pour pouvoir les nommer
- 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

Si vous avez l'impression que vous devez commenter votre code pour qu'on le comprenne, c'est que vous pouvez généralement le réécrire de manière plus expressive.

Avoir du code écrit de manière uniforme permet de le lire plus facilement.

Ce qui définit le style :

- PascalCase, camelCase, snake_case
- Tabs / Spaces
- Saut de ligne
- Indentation
- etc.

Pour cela, on met en place des **conventions de style**.

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.

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

Ces erreurs entraînent parfois des bugs très difficiles à identifier et corriger :

- variables non initialisées
- accès à de la mémoire déjà libérée (dangling-ref)
- casts implicites
- etc.

On peut ajouter des mots-clefs dans le code pour forcer le compilateur à vérifier certaines conditions :

- `override` : pour vérifier qu'une fonction est bien une redéfinition d'un membre de la classe de base
- `explicit` : pour s'assurer qu'on ne fait pas de conversion implicite
- `[[nodiscard]]` : pour vérifier que la valeur de retour d'une fonction est utilisée

On a 4 types de casts en C++ :

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `const_cast`

Le `dynamic_cast` permet de convertir un type de base en type dérivé. On parle de **down-cast** : `Base& → Derived&`

```
std::string get_species(const Animal& animal)
{
    if (dynamic_cast<const Dolphin*>(&animal))
    {
        return "Dolphin";
    }

    if (dynamic_cast<const Cow*>(&animal))
    {
        return "Cow";
    }

    return "Unknown";
}
```

Le `static_cast` permet au choix de :

- **up-cast** : `Derived& → Base&`
- ajouter un `const` à un type : `T& → const T&`
- effectuer une conversion explicite valide, par exemple `float → int`

Dans certains cas, on peut également l'utiliser pour réaliser un **down-cast**.

Le `reinterpret_cast` permet d'interpréter la mémoire associée à une variable comme s'il s'agissait d'un autre type.

Quelques cas d'utilisation :

- allouer des pools de mémoire : $\text{byte}^* \rightarrow T^*$
- **sérialisation / désérialisation binaire** : $T \rightarrow \text{char}^* \rightarrow T$
- lire la mantisse d'un float : $\text{float}^* \rightarrow \text{byte}^*$
- ...

Le `const_cast` permet de retirer les `const` : `const T& → T&`

Attention, si une variable a été définie comme étant constante, tenter de la modifier constitue un undefined behavior.

Deux cas d'utilisation :

- éviter la duplication de code lorsqu'on définit des overload non-const de fonctions const
- lorsqu'une fonction d'une librairie externe prend un `T&` ou un `T*`, mais qu'elle garantit qu'elle ne modifie pas la valeur

L'intérêt d'utiliser les casts du C++, c'est que le compilateur effectue un certain nombre de vérifications en amont :

- `static`, `dynamic` et `reinterpret` ne permettent pas de retirer des `const`
- `dynamic_cast` ne peut être appelé que sur des `T*` ou des `T&`
- `dynamic_cast A → B` ne peut être appelé que si `A` hérite de `B` ou que `B` hérite de `A`
- ...

Avant d'implémenter un algorithme ou une structure de données, vérifiez d'abord si la librairie standard le propose déjà.

1. Vous aurez moins de code à écrire.
2. Ça marchera forcément, vu que ça a été utilisé par des milliers de personnes avant vous.
3. Ça sera plus efficace, vu que les personnes qui l'ont codé sont beaucoup plus expérimentées que vous.

Une assertion permet d'interrompre le programme si une condition est fausse :

```
int division(int dividende, int diviseur)
{
    assert(diviseur != 0 && "Le diviseur ne peut pas etre nul !");
    return dividende / diviseur;
}
```

On peut utiliser les assertions pour vérifier les **pré-conditions** et **post-conditions** de fonctions.

Elles servent donc à identifier les erreurs de programmation, pas les erreurs d'utilisation.

En effet, une fois qu'un logiciel est mis à disposition des utilisateurs, on ne veut pas que celui-ci crash si l'utilisateur fait une mauvaise manipulation.

Les assertions sont donc activées lorsqu'on compile en debug, et désactivées une fois que l'on compile en mode release.

Pour traiter les erreurs d'utilisation d'un programme, on peut utiliser des exceptions.

Avant : <https://godbolt.org/z/PnsPjnr6x>

Après : <https://godbolt.org/z/5n7aPrv6x>

Les exceptions permettent d'isoler le code lié à la gestion d'erreurs.

Une **régression** désigne l'introduction d'un bug dans une fonctionnalité suite à un changement dans le code.

Exemple : <https://godbolt.org/z/P68Ehsfz3>

Lorsque les tests sont manuels, vérifier que tout fonctionne comme avant s'avère fastidieux et on oublie souvent de tester une partie des cas d'utilisation.

Les tests unitaires sont donc une très bonne manière de prévenir les régressions, car une fois qu'ils sont codés, on peut les rejouer à l'infini sans aucun effort.

- Améliorer l'expressivité du code
- Définir des conventions et les suivre
- Faciliter la détection des erreurs de programmation (via le compilateur ou les assertions)
- Utiliser les casts du C++ plutôt que le cast C
- Connaître et utiliser ce que la librairie standard propose
- Gérer le traitement des erreurs avec les exceptions
- Implémenter des tests unitaires

Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi(v);  
    std::cout << "Value " << i << " is an integer" << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```

Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi(v);  
    std::cout << "Value " << i << " is an integer" << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```

Si la conversion est impossible,
stoi() lève une exception

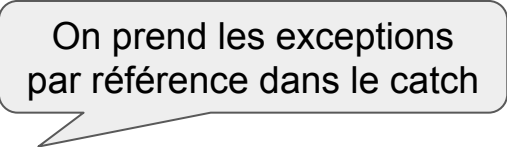
Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi(v);  
    std::cout << "Value " << i << " is an integer" << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```

On sort alors immédiatement du bloc try pour passer dans le bloc catch

Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi  
    std::cout << "Value " << << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```



On prend les exceptions
par référence dans le catch

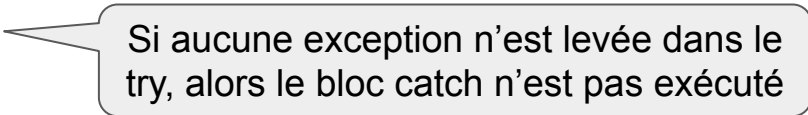
Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi(v);  
    std::cout << "Value " << i << " is an integer" << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```

La fonction what() permet
d'accéder au contenu de l'erreur

Pour intercepter les exceptions levées dans une fonction, il faut placer l'appel à cet fonction à l'intérieur d'un bloc try / catch :

```
auto v = std::string {};  
std::cin >> v;  
  
try  
{  
    const auto i = std::stoi(v);  
    std::cout << "Value " << i << " is an integer" << std::endl;  
}  
catch (const std::exception& e)  
{  
    std::cerr << e.what() << std::endl;  
}
```



Si aucune exception n'est levée dans le try, alors le bloc catch n'est pas exécuté

Pour lever une exception, il faut d'abord la construire.

La librairie standard propose plusieurs types pour cela, qu'elle expose dans le header `<stdexcept>` :

- `std::invalid_argument`
- `std::logic_error`
- `std::runtime_error`
- etc

Toutes les exceptions de `<stdexcept>` attendent une chaîne de caractères à leur construction.

Par exemple, pour instancier une `std::runtime_error`, on peut écrire :

```
std::runtime_error { "le fichier " + path + " n'existe pas" }
```

Une fois l'exception construite, on utilise le mot-clef `throw` pour la lever :

```
int factorial(int n)
{
    if (n < 0)
    {
        throw std::invalid_argument { "factorial expects a positive integer"
    };
    }
    ...
}
```

Lorsqu'une exception est levée, on remonte la pile d'appel jusqu'à trouver un bloc catch qui accepte une exception de ce type :

<https://godbolt.org/z/o19Gq3hh4>

On note que :

- un bloc try peut accepter plusieurs blocs catch
- le polymorphisme fonctionne
- on peut intercepter n'importe quel type avec `catch (. . .)`

Toutes les variablesinstanciées sur la pile sont correctement détruites lorsque l'exception remonte la pile d'appel.

On parle de **stack unwinding**.

```
void test_leak()  
{  
    auto str = std::string { "toto" };  
    throw std::logic_error { "an error occurred" };  
}
```


Toutes les variables instanciées sur la pile sont correctement détruites lorsque l'exception remonte la pile d'appel.

On parle de **stack unwinding**.

```
void {  
  {  
    auto str = std::string { "toto" };  
    throw std::logic_error { "an error occurred" };  
  }  
}
```

Le destructeur de str est bien appelé lorsque l'exception est levée et qu'on sort du scope.

Pour définir un nouveau type d'exception, il suffit d'hériter de `std::exception`, ou bien de l'une de ses sous-classes :

<https://godbolt.org/z/cfb4nfWar>

Si vous décidez d'hériter de `std::exception`, il faut override la fonction `what()`.

Si vous décidez d'hériter de l'une de ses sous-classes `T`, vous aurez besoin de définir un constructeur pour appeler celui de `T`.

Lorsqu'on souhaite implémenter un algorithme, le résultat est souvent peu lisible, et il faut s'y reprendre à plusieurs fois pour s'assurer que tous les cas possibles ont été correctement gérés.

L'efficacité d'un algorithme dépendra également de la structure de données sur laquelle il est appliqué, et il faut donc plusieurs implémentations afin de pouvoir utiliser la meilleur en fonction de l'objet sur lequel on l'applique.

Par exemple, si on essaye de trouver le plus petit élément d'une liste, on ne s'y prendra pas de la même manière que si on essaye de trouver le plus petit élément d'un tas.

Afin de pallier ce problème, la librairie standard propose une interface commune pour exécuter les algorithmes les plus courants sur n'importe quelle structure de données.

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

Les fonctions sont toutes interfacées de la même manière. Elles attendent en entrée :

- un itérateur qui pointe sur le début de la structure de données
- un itérateur qui pointe sur la fin de la structure de données
- les éventuels autres arguments

Exemples : <https://godbolt.org/z/fovDE3dn8>

- Recherche
 - find
 - find_if
 - min_element
 - max_element
 - lower_bound
 - upper_bound
- Accumulation
 - aggregate
 - reduce
- Transformation
 - transform
 - remove_if
 - sort
- Vérification de prédicats
 - all_of
 - any_of
 - none_of

La plupart des algorithmes attendent une “opération” :

- `std::all_of` ou `std::find` attendent un prédicat $T \rightarrow \text{bool}$
- `std::transform` attend une opération $T \rightarrow T'$

L'opération attendue peut être :

- un pointeur de fonction
- un foncteur (ou objet-fonction)
- une lambda

Un foncteur est un objet qui propose un opérateur ().

```
struct Printer
{
    void operator()(const std::string& s) const
    {
        std::cout << s << std::endl;
    }
};
```


Pour invoquer cet opérateur, il faut instancier un objet du type, puis écrire `(p1, p2, ...)` derrière son identifiant :

```
auto printer = Printer {}; // instantiation
printer("toto");           // invoke operator()
```

On peut également l'écrire en une seule instruction :

```
Printer {}("toto"); // instantiation + invocation
```

Une lambda est un outil qui permet, à partir d'une syntaxe concise, de générer automatiquement la classe d'un foncteur et de l'instancier.

Le code ci-dessous est équivalent celui écrit précédemment :

```
auto printer = [] (const std::string& s)
{
    std::cout << s << std::endl;
};

printer("toto");
```

Les crochets `[]` au début de la lambda permettent de spécifier les variables locales que l'on souhaite pouvoir utiliser à l'intérieur de la lambda.

Exemple : <https://godbolt.org/z/95cf7G3bK>

La capture peut être faite :

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

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

Afin de pouvoir stocker une “opération” quelconque dans un champ, la librairie standard définit le type templaté `std::function` dans `<functional>`.

Exemple : <https://godbolt.org/z/344jGs35f>