Conventions en C++

1. Nommage des variables et des fonctions

- **Variables et fonctions** : Utilisez le style **camelCase** ou **snake_case**, selon les conventions de votre projet.
  - CamelCase (souvent utilisé) : la première lettre est en minuscule, et chaque mot suivant commence par une majuscule.

    ```
    cpp
    Copier le code
    int myVariable;
    void processInput();
    ```

  - snake_case : toutes les lettres sont en minuscules, et les mots sont séparés par des underscores.

    ```
    cpp
    Copier le code
    int my_variable;
    void process_input();
    ```

- **Noms de classes** : Utilisez le style **PascalCase** (ou UpperCamelCase) pour les classes, c'est-à-dire que chaque mot commence par une majuscule.

  ```
  cpp
  Copier le code
  class MyClass;
  ```

- **Constantes** : Utilisez des lettres majuscules avec des underscores pour les séparer.

  ```
  cpp
  Copier le code
  ```

```
const int MAX_VALUE = 100;
```

- **Noms significatifs** : Choisissez des noms qui décrivent clairement l'intention de la variable, de la fonction ou de la classe.
  - Mauvais : `int x;`
  - Bon : `int numberOfItems;`

2. Indentation et espacement

- Indentez correctement votre code pour rendre la structure et la hiérarchie visibles.
  - L'indentation standard est de **4 espaces** (ou parfois 2 espaces), évitez d'utiliser des tabulations.

```
cpp
Copier le code
if (condition) {
    doSomething();
} else {
    doSomethingElse();
}
```

- Utilisez des **espaces autour des opérateurs** pour améliorer la lisibilité.

```
cpp
Copier le code
int sum = a + b;  // Bon
int sum=a+b;      // Mauvais
```

- Mettez des **espaces après les virgules** dans les listes d'arguments ou les déclarations de variables.

```
cpp
Copier le code
int a, b, c;  // Bon
```

```
int a,b,c;    // Mauvais
```

3. Commentaires

- **Commentaires ligne simple** : Utilisez `//` pour un commentaire sur une seule ligne.

```
cpp
Copier le code
// Calcule la somme des deux nombres
int sum = a + b;
```

- **Commentaires multi-lignes** : Utilisez `/* ... */` pour les commentaires multi-lignes, surtout pour des explications plus détaillées.

```
cpp
Copier le code
/*
 * Cette fonction calcule la somme de deux entiers.
 * @param a : le premier entier
 * @param b : le second entier
 * @return : la somme des deux entiers
 */
int add(int a, int b) {
    return a + b;
}
```

- **Documentation des fonctions** : Utilisez des commentaires Doxygen pour documenter les fonctions et les classes dans des projets plus complexes.

```
cpp
Copier le code
/**
 * @brief Ajoute deux entiers.
 * @param a Le premier entier.
```

```
 * @param b Le deuxième entier.
 * @return La somme des deux entiers.
 */
int add(int a, int b) {
    return a + b;
}
```

4. Organisation des fichiers

- **Fichiers d'en-tête (.h/.hpp)** : Utilisez-les pour déclarer les interfaces (classes, structures, fonctions). Évitez de mettre des implémentations dans ces fichiers.

```
cpp
Copier le code
// Fichier .h
class MyClass {
public:
    void doSomething();
};
```

- **Fichiers source (.cpp)** : Utilisez-les pour implémenter les méthodes et fonctions déclarées dans les fichiers d'en-tête.

```
cpp
Copier le code
// Fichier .cpp
#include "MyClass.h"void MyClass::doSomething() {
    // Implémentation
}
```

- **Include Guards** : Utilisez des gardes d'inclusion pour éviter les inclusions multiples d'un fichier d'en-tête.

```
cpp
Copier le code
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
    // Déclaration
};

#endif // MYCLASS_H
```

5. Gestion des espaces de noms

- **Utilisez des namespaces** pour organiser votre code et éviter les conflits de noms.

```
cpp
Copier le code
namespace MyNamespace {
    class MyClass {
        // Déclaration
    };
}
```

- **Évitez** `using namespace std;` **dans les fichiers d'en-tête** pour prévenir les conflits. Privilégiez le fait de spécifier l'espace de noms complet.

```
cpp
Copier le code
std::string myString;  // Bon
using namespace std;    // À éviter dans les headers
```

6. Gestion de la mémoire et pointeurs intelligents

- **Évitez les fuites de mémoire** en utilisant des **pointeurs intelligents** comme `std::unique_ptr` et `std::shared_ptr` au lieu de gérer manuellement la mémoire

avec des pointeurs bruts (`new` / `delete`).

```
cpp
Copier le code
std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>
();
```

- **Préférez les références** aux pointeurs si vous n'avez pas besoin d'une gestion dynamique de la mémoire.

```
cpp
Copier le code
void process(const MyClass& obj);
```

7. Const-correctness

- Utilisez le mot-clé `const` autant que possible pour indiquer que les valeurs ne changeront pas.
  - **Variables constantes** :

```
cpp
Copier le code
const int maxLimit = 100;
```

  - **Méthodes constantes** : Les méthodes qui ne modifient pas l'état d'un objet doivent être marquées comme `const`.

```
cpp
Copier le code
int getValue() const;
```

  - **Pointeurs constants** :

```
cpp
Copier le code
const int* ptr;   // Un pointeur vers une valeur cons
tante (la valeur pointée ne peut pas être modifiée).
int* const ptr;   // Un pointeur constant (le pointeu
r ne peut pas changer, mais la valeur pointée peut).
```

8. Gestion des erreurs

- Utilisez **les exceptions** pour gérer les erreurs de manière appropriée, surtout dans les cas où la fonction ne peut pas terminer son travail.

```
cpp
Copier le code
try {
    // Code pouvant lancer une exception
} catch (const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

- **Préférer les exceptions** aux codes d'erreur pour indiquer des erreurs. Cela rend le code plus lisible et sépare la logique métier de la gestion des erreurs.

9. Respect de la RAII (Resource Acquisition Is Initialization)

- Suivez le principe RAII en C++ pour la gestion des ressources (mémoire, fichiers, connexions réseau, etc.). RAII garantit que les ressources sont automatiquement libérées lorsqu'elles ne sont plus nécessaires.

```
cpp
Copier le code
std::ofstream file("output.txt");
if (!file) {
    throw std::runtime_error("Failed to open file");
}
```

```
// Le fichier sera automatiquement fermé à la destructio
n de l'objet `file`
```

10. Tests

- Écrivez des **tests unitaires** pour assurer la qualité et la fiabilité de votre code. En C++, des frameworks comme **Google Test** ou **Catch2** sont populaires pour écrire des tests.

```
cpp
Copier le code
TEST(MyClassTest, HandlesZeroInput) {
    MyClass obj;
    EXPECT_EQ(obj.doSomething(0), 0);
}
```

11. Clarté du code

- **KISS (Keep It Simple, Stupid)** : Évitez les constructions complexes ou déroutantes. Un code simple est plus facile à maintenir.
- **DRY (Don't Repeat Yourself)** : Évitez de dupliquer du code. Utilisez des fonctions ou des classes réutilisables lorsque cela est possible.
- **YAGNI (You Aren't Gonna Need It)** : N'implémentez pas de fonctionnalités inutiles ou anticipées sans un besoin concret.