4. Programmation générique avec la bibliothèque Qt

4.1. Introduction

La bibliothèque Qt propose un ensemble de conteneurs et d'outils conçus pour gérer efficacement les données. Ces conteneurs sont optimisés pour fonctionner avec les types spécifiques de Qt, comme les objets dérivés de QObject, et s'intègrent bien avec les mécanismes de Qt, comme les signaux/slots et la gestion de la mémoire. Les conteneurs Qt sont regroupés dans l'espace de nommage Qt et sont conçus pour une compatibilité accrue avec les API de Qt.

4.2. Les conteneurs Qt

Les conteneurs Qt sont des structures de données abstraites permettant de stocker et de manipuler des collections d'objets. Ils se divisent en plusieurs catégories : conteneurs séquentiels, conteneurs associatifs, et des conteneurs plus spécialisés comme les piles et les files. Ils diffèrent principalement par la manière dont ils gèrent les données et les performances qu'ils offrent.

Tous les conteneurs fournis par Qt sont basés sur le concept des **templates** (modèles) en C++. Les templates permettent de définir des classes ou des fonctions génériques pouvant fonctionner avec n'importe quel type de données. Cela signifie qu'un conteneur Qt peut être utilisé pour stocker des éléments de n'importe quel type, qu'il s'agisse de types de base comme `int`, `double...` ou de types plus complexes comme `QString` ou des objets définis par l'utilisateur.

Les méthodes suivantes sont communes à la plupart des conteneurs Qt :

- `isEmpty()` : Retourne true si le conteneur est vide.
- `size()` : Retourne le nombre d'éléments dans le conteneur.
- `clear()` : Supprime tous les éléments.
- `contains(const T& value)` : Retourne `true` si l'élément est présent.
- `remove(const T& value)` : Supprime la première occurrence de l'élément spécifié.

Un certain nombre d'opérateurs sont surchargés pour simplifier l'écriture :

- Opérateur `=` : L'opérateur d'affectation est utilisé pour assigner une valeur à un conteneur. Les conteneurs sont conçus pour copier ou déplacer leur contenu en fonction du type de l'objet.
- Opérateur `==` et `!=` : Ces opérateurs permettent de comparer deux conteneurs pour vérifier s'ils contiennent les mêmes éléments.
- Opérateur `<<` et `>>` : Ces opérateurs sont surchargés pour ajouter ou extraire des éléments, principalement lors des opérations d'entrée/sortie.
- Opérateur `+` et `+=` : L'opérateur `+` est utilisé pour concaténer deux conteneurs, tandis que `+=` permet d'ajouter des éléments ou de fusionner des conteneurs.

4.2.1. Conteneurs séquentiels

Les conteneurs séquentiels stockent les données de manière contiguë, les uns après les autres. Qt propose plusieurs conteneurs de ce type avec chacun des caractéristiques propres :

- **`Qlist` :** C'est un conteneur généraliste largement utilisé dans les versions précédentes de Qt. `QList` permet de stocker des éléments de taille variable avec un accès rapide en temps constant à la majorité des éléments. Cependant, à partir de Qt 6, `QList` est déconseillé au profit de `QVector` pour la plupart des types de données.
- **`QVector` :** Similaire à `QList`, mais optimisé pour les tableaux contigus en mémoire. Il est particulièrement adapté à des opérations nécessitant une gestion rapide de grandes quantités de données et est le choix recommandé dans Qt 6 pour remplacer `QList` dans la plupart des cas.

Pour accéder à un élément, on utilise l'opérateur `[]` ou la méthode `at(int index)`. Les autres méthodes communes à ces conteneurs sont :

- `append(const T& value)` : Ajoute un élément à la fin.
- `prepend(const T& value)` : Ajoute un élément au début.
- `insert(int index, const T& value)` : Insère un élément à un index donné.
- `removeAt(int index)` : Supprime l'élément à un index donné.
- `replace(int index, const T& value)` : Remplace l'élément à un index donné.
- `first()` et `last()` : Retourne le premier ou le dernier élément.

Exemples d'utilisation

Voici un exemple qui illustre l'utilisation de QList.

Exemple d'utilisation de QList — ExempleQList.cpp

```cpp
#include <QList>
#include <QDebug>
int main() {

    QList<int> numbers;

    numbers << 10 << 20 << 30;
    numbers.append(40);
    numbers.prepend(5);

    qDebug() << "Liste des nombres :" << numbers;
    qDebug() << "Premier élément :" << numbers.first();
    qDebug() << "Dernier élément :" << numbers.last();

    return 0;
}
```

Voici un exemple qui illustre l'utilisation de QVector.

Exemple d'utilisation de QVector — ExempleQVector.cpp

```cpp
#include <QVector>
#include <QDebug>
int main() {

    QVector<int> numbers = {1, 2, 3, 4, 5};

    numbers.replace(2, 10); // Remplace l'élément à l'index 2
    numbers.append(6); // Ajoute un élément à la fin

    qDebug() << "Vérification du vecteur :" << numbers;
    qDebug() << "Élément à l'index 2 :" << numbers[2];

    return 0;
}
```

La bibliothèque propose des conteneurs dérivés apportant des fonctionnalités spécifiques, par exemple :

- **`QStringList`** : C'est une classe dérivée de `QList<QString>`, optimisée pour gérer des collections de chaînes de caractères. Elle hérite des fonctionnalités de `QList` et ajoute des méthodes spécifiques pour manipuler des listes de chaînes de caractères plus facilement :
  - `join(const QString &separator)` : Concatène tous les éléments de la liste en une seule chaîne, avec un séparateur entre chaque élément.
  - `filter(const QString &str)` : Retourne une nouvelle liste contenant uniquement les chaînes qui contiennent le texte spécifié.
  - `replaceInStrings(const QString &before, const QString &after)` : Remplace toutes les occurrences d'une sous-chaîne par une autre dans chaque élément de la liste.
  - `indexOf(const QString &str, int from = 0)` : Retourne l'index de la première occurrence de la chaîne donnée.

Exemple d'utilisation de QStringList — ExempleQStringList.cpp

```cpp
#include <QStringList>
#include <QDebug>
int main() {
    QStringList names;
    names << "Alice" << "Bob" << "Charlie";

    qDebug() << "Liste originale :" << names;
    qDebug() << "Concatenation avec virgule :" << names.join(", ");
    qDebug() << "Filtrer les éléments contenant 'o' :" << names.filter("o");

    return 0;
}
```

- **QByteArray :** `Optimisé pour gérer des données binaires ou des chaînes ASCII. Contrairement à QString, qui gère du texte Unicode, QByteArray est conçu pour manipuler des séquences d'octets, souvent utilisées dans la gestion des flux d’entrée/sortie, comme les fichiers ou les communications réseau.`
  - `append(const QByteArray &ba)` ou `append(char c)` : Ajoute des octets à la fin du tableau.
  - `toHex()` : Convertit le tableau en une chaîne représentant les données en hexadécimal.
  - `fromHex(const QByteArray &hex)` : Convertit une chaîne hexadécimale en tableau d'octets.
  - `left(int n)` : Retourne les n premiers octets du tableau.
  - `right(int n)` : Retourne les n derniers octets.
  - `mid(int pos, int len = -1)` : Retourne une portion du tableau, à partir de `pos` pour `len` octets.

Exemple d’utilisation de QByteArray **ExempleQByteArray.cpp**

```cpp
#include <QByteArray>
#include <QDebug>

int main() {

    QByteArray byteArray("Hello World");
    byteArray.append("!");

    qDebug() << "ByteArray avec un point d'exclamation :" << byteArray;
    qDebug() << "Représentation hexadécimale :" << byteArray.toHex();

    return 0;
}
```

4.2.2. Conteneurs associatifs

Les conteneurs associatifs stockent des paires clé-valeur, ce qui permet un accès rapide aux éléments en fonction de leur clé.

- **QMap :** Conteneur associatif ordonné qui associe des clés uniques à des valeurs. Les éléments sont stockés triés selon les clés.
- **QHash :** Semblable à `QMap`, mais les éléments sont stockés de manière non ordonnée pour des performances accrues lors des recherches.

Méthodes communes :

- `insert(const Key& key, const T& value)` : Insère une paire clé-valeur.
- `value(const Key& key)` : Retourne la valeur associée à une clé.
- `remove(const Key& key)` : Supprime la paire clé-valeur correspondant à la clé.
- `contains(const Key& key)` : Vérifie si la clé est présente.
- `keys()` et `values()` : Retourne une liste des clés ou des valeurs.

Exemples d’utilisation

Exemple d’utilisation de QMap **ExempleQMap.cpp**

```cpp
#include <QMap>
#include <QDebug>

int main() {

    QMap<QString, int> ageMap;
    ageMap.insert("Alice", 25);
    ageMap.insert("Bob", 30);

    qDebug() << "Âge d'Alice :" << ageMap.value("Alice");
    qDebug() << "Toutes les clés :" << ageMap.keys();

    return 0;
}
```

De manière similaire, voici un exemple d'utilisation de QHash.

Exemple d'utilisation de QHash **ExempleQHash.cpp**

```cpp
#include <QHash>
#include <QDebug>

int main() {
    QHash<QString, QString> country;
    country.insert("FR", "France");
    country.insert("DE", "Germany");
    qDebug() << "Nom de pays pour 'FR' :" << country.value("FR");
    return 0;
}
```

Variantes spécifiques :

- **QMultiMap** et **QMultiHash** : Versions de `QMap` et `QHash` qui permettent de stocker plusieurs valeurs pour une même clé.
    - `insertMulti(const Key& key, const T& value)` : Insère une nouvelle paire clé-valeur, même si la clé existe déjà.
    - `values(const Key& key)` : Retourne une liste de toutes les valeurs associées à une clé donnée.

Pour les deux types de conteneurs, l'exemple est similaire, QMultiHash peut être remplacé par QMultiMap.

Exemple d'utilisation de QHash **ExempleQHash.cpp**

```cpp
#include <QMultiHash>
#include <QDebug>

int main() {
    QMultiHash<QString, QString> emailList;
    emailList.insert("Bob", "bob@example.com");
    emailList.insert("Bob", "bob@work.com");
    qDebug() << "Tous les e-mails de Bob :" << emailList.values("Bob");
    return 0;
}
```

Remarque

Le conteneur le plus approprié pour maintenir une liste d'éléments ordonnés sans nécessiter de gestion manuelle est **QMap**, ou par extension **QMultiMap**. En effet, **QMap<Key, T>** est un conteneur qui stocke des paires clé-valeur, où les clés sont automatiquement triées. **QMultiMap** fonctionne de manière similaire, mais permet d'associer plusieurs valeurs à une même clé, tout en maintenant l'ordre des clés.

Exemple fabrication d'une liste ordonnée **ListeOrdo.cpp**

```cpp
QMap<int, QString> map;
map.insert(3, "Trois");
map.insert(1, "Un");
map.insert(2, "Deux");

// Les éléments seront ordonnés par la clé entière : 1 -> "Un", 2 -> "Deux", 3 -> "Trois"
```

Dans cet exemple, les chaînes de caractères sont stockées dans un **QMap** associées à des clés entières. Le conteneur trie automatiquement les éléments par ordre croissant de clé. Lorsqu'on parcourt le conteneur de manière standard, les éléments sont retournés dans cet ordre croissant. Si on souhaite obtenir les éléments en ordre décroissant, il suffit de commencer depuis la fin du conteneur et de revenir vers le début.

L'exemple sera repris dans le chapitre sur les itérateurs.

Enfin, ce dernier type de conteneur associatif représente un ensemble.

- **QSet :** Conteneur d'éléments uniques, non ordonnés. Il est basé sur une table de hachage, tout comme `QHash`, et offre une gestion rapide des éléments uniques.

Les méthodes spécifiques à ce type de conteneur sont :

- `insert(const T& value)` : Ajoute un élément à l'ensemble.
- `remove(const T& value)` : Supprime un élément de l'ensemble.
- `intersect(const QSet<T>& other)` : Crée une intersection avec un autre ensemble.
- `unite(const QSet<T>& other)` : Crée une union avec un autre ensemble.

Exemple d'utilisation de Qset **ExempleQSet.cpp**

```cpp
#include <QSet>
#include <QDebug>

int main() {
    QSet<int> uniqueNumbers;
    uniqueNumbers << 1 << 2 << 3 << 4;
    qDebug() << "Contient 3 ? :" << uniqueNumbers.contains(3);
    uniqueNumbers.insert(5);
    qDebug() << "Union avec un autre ensemble :" << uniqueNumbers.unite(QSet<int>() << 6 << 7);
    return 0;
}
```

De même, on trouve également dans la bibliothèque

- **QMultiSet** : C'est une variante de `QSet` qui permet de stocker des éléments dupliqués. `QMultiSet` accepte les doublons.

Les méthodes spécifiques associées à ce type de conteneur sont par exemple :

- `insert(const T& value)` : Ajoute un élément, même s'il est déjà présent dans l'ensemble.
- `count(const T& value)` : Retourne le nombre d'occurrences d'un élément donné.

Exemple d'utilisation de QMultiSet **ExempleQMultiSet.cpp**

```cpp
#include <QMultiSet>
#include <QDebug>

int main() {
    QMultiSet<int> numberSet;
    numberSet << 1 << 1 << 2 << 3;
    qDebug() << "Nombre d'occurrences de 1 :" << numberSet.count(1);
    numberSet.insert(2);
    qDebug() << "Ensemble mis à jour :" << numberSet;
    return 0;
}
```

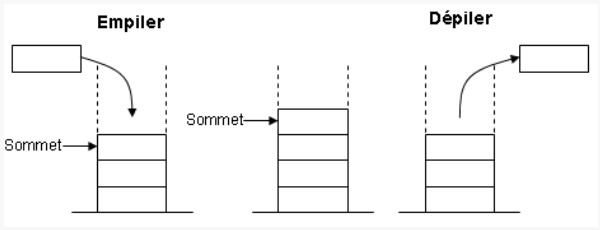
4.2.3. Autres conteneurs spécifiques

Les piles et les files sont implémentées de base dans la bibliothèque Qt pour plus d'efficacité et une utilisation plus intuitive.

- **QStack** : Implémentation d'une pile (LIFO – Last In, First Out). Bien que dérivé de `QVector`, `QStack` fournit une interface intuitive pour les opérations spécifiques à une pile :
  - `push(const T& value)` : Ajoute un élément au sommet de la pile.
  - `pop()` : Retire l'élément au sommet de la pile.
  - `top()` : Retourne l'élément au sommet de la pile sans le retirer.

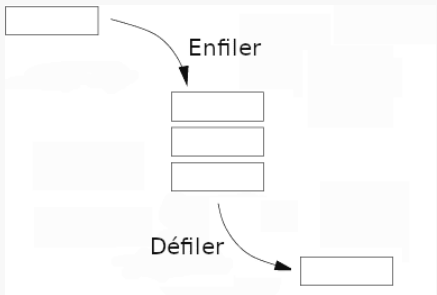
Exemple d'utilisation de Qstack **ExempleQStack.cpp**

```
#include <QStack>
#include <QDebug>
int main() {
    QStack<int> stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);
    qDebug() << "Élément au sommet :" << stack.top();
    stack.pop();
    qDebug() << "Sommet après suppression :" << stack.top();
    return 0;
}
```



- **QQueue** : Implémentation d'une file d'attente (FIFO – First In, First Out). Dérivé de `QList`, `QQueue` simplifie les opérations spécifiques à une file :
  - `enqueue(const T& value)` : Ajoute un élément à la fin de la file (l'équivalent de `append()` dans `QVector` ou `QList`).
  - `dequeue()` : Retire et retourne l'élément au début de la file (équivalent à `removeFirst()`).
  - `head()` : Retourne l'élément en tête de la file sans le retirer (similaire à `first()`).

```
#include <QQueue>
#include <QDebug>
int main() {
    QQueue<int> queue;
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
    qDebug() << "Élément en tête :" << queue.head();
    queue.dequeue();
    qDebug() << "Élément après suppression :" << queue.head();
    return 0;
}
```



Attention

Même si `QStack` et `QQueue` dérivent respectivement de `QVector` et `QList`, l'opérateur `[]` n'est pas disponible pour parcourir les éléments ou accéder directement à des éléments arbitraires. Ils se concentrent uniquement sur l'accès aux éléments au sommet (pour `QStack`) ou en tête (pour `QQueue`).

4.3. Les itérateurs

Les itérateurs sont des objets qui permettent de parcourir des collections de manière séquentielle, élément par élément, sans exposer leur structure interne. Dans Qt, comme en C++, les itérateurs jouent un rôle essentiel dans la manipulation efficace des données contenues dans des collections telles que **QList**, **QVector**, **QMap**, et bien d'autres.

4.3.1. Types d'itérateurs dans Qt

Qt propose plusieurs types d'itérateurs, chacun adapté à des besoins particuliers. Voici les principaux types d'itérateurs que vous pouvez rencontrer dans les conteneurs Qt :

- **Itérateurs explicites** : Ces itérateurs sont des classes spécifiques fournies par Qt pour parcourir les conteneurs. Ils offrent des méthodes claires pour parcourir les conteneurs de manière directe. Il existe deux variantes principales : les itérateurs en lecture seule et les itérateurs modifiables. Par exemple :
  - `QListIterator<T>` et `QMutableListIterator<T>`
  - `QMapIterator<Key, T>` et `QMutableMapIterator<Key, T>`
  - `QVectorIterator<T>` et `QmutableVectorIterator<T>`

Méthodes des itérateurs explicites :

- `hasNext()` : Renvoie `true` si l'itérateur peut avancer vers l'élément suivant.
- `next()` : Avance l'itérateur et retourne l'élément suivant.
- `peekNext()` : Renvoie l'élément suivant sans avancer l'itérateur.
- `remove()` : Supprime l'élément actuel (disponible uniquement pour les itérateurs modifiables).
- `insert(T value)` : Insère un élément avant l'élément actuel.

Exemple d'utilisation d'itérateurs en lecture seule, ils ne permettent que de parcourir les conteneurs sans pouvoir les modifier :

Exemple d'itérateur en lecture seule — IterateurlectureSeule.cpp

```
QList<int> list = {1, 2, 3, 4};
QListIterator<int> it(list); // Itérateur en lecture seule
while (it.hasNext()) {
    int value = it.next(); // Accède au prochain élément
    qDebug() << value;
}
```

Les itérateurs modifiables (comme `QMutableListIterator<T>`) permettent non seulement de parcourir, mais aussi de modifier la collection durant le parcours.

```
QList<int> list = {1, 2, 3, 4};
QMutableListIterator<int> it(list);
while (it.hasNext()) {
    int value = it.next();
    if (value % 2 == 0) {
        it.remove(); // Supprime les éléments pairs
    }
}
qDebug() << list; // list contient maintenant {1, 3}
```

- **Itérateurs implicites (STL-like)** : Qt supporte également des itérateurs similaires à ceux présent dans la STL (Standard Template Library), qui se comportent comme des pointeurs. Les itérateurs implicites fonctionnent avec des méthodes comme `begin()`, `end()`, `rbegin()` (pour un parcours à l'envers), etc. Ils peuvent être utilisés de manière similaire à des pointeurs dans des boucles, comme les itérateurs classiques de C++ :
  - `QList<T>::iterator` et `QList<T>::const_iterator`
  - `QMap<Key, T>::iterator` et `QMap<Key, T>::const_iterator`

Ces deux types d'itérateurs offrent des fonctionnalités similaires, mais le choix entre eux dépend souvent des préférences de l'utilisateur et des cas d'utilisation spécifiques.

Méthodes des itérateurs implicites :

- `begin()` : Retourne un itérateur au début de la collection.
- `end()` : Retourne un itérateur à la fin de la collection (après le dernier élément).
- `rbegin()` et `rend()` : Pour un parcours inversé (du dernier au premier élément).
- `const_iterator` : Permet un parcours en lecture seule.

```
QList<int> list = {1, 2, 3, 4};
for (QList<int>::iterator it = list.begin(); it != list.end(); ++it) {
    if (*it % 2 == 0) {
        *it = *it * 2; // Double les nombres pairs
    }
}
qDebug() << list; // list contient maintenant {1, 4, 3, 8}
```

Pour reprendre l'exemple de la liste de données ordonnée en utilisant le conteneur QMap vue précédemment, l'utilisation des itérateurs facilite le parcours du conteneur.

```
#include <QCoreApplication>
#include <QMap>
#include <QString>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QMap<int, QString> map;
    map.insert(3, "Trois");
    map.insert(1, "Un");
    map.insert(2, "Deux");

    // Affichage en ordre croissant
    qDebug() << "Ordre croissant :";
    for (auto it = map.cbegin(); it != map.cend(); ++it) {
        qDebug() << it.key() << "->" << it.value();
    }

    // Affichage en ordre décroissant
    qDebug() << "\nOrdre décroissant :";
    for (auto it = map.crbegin(); it != map.crend(); ++it) {
        qDebug() << it.key() << "->" << it.value();
    }

    return a.exec();
}
```

L'ordre des éléments affichés sera donc :

- **Ordre croissant** : `1 -> "Un"`, `2 -> "Deux"`, `3 -> "Trois"`.
- **Ordre décroissant** : `3 -> "Trois"`, `2 -> "Deux"`, `1 -> "Un"`.

L'utilisation de **cbegin()/cend()** pour l'ordre croissant et **crbegin()/crend()** pour l'ordre décroissant permet de naviguer facilement dans les deux sens.

4.3.2. Différences entre itérateurs explicites et implicites

Caractéristiques	Itérateurs explicites	Itérateurs implicites
Syntaxe	Utilise des méthodes comme `next()`, `hasNext()`	Utilise des opérateurs comme ++, *, ->
Modifiabilité	Les itérateurs non mutables ne peuvent pas modifier la collection	Peut modifier la collection avec des itérateurs non-const
Clarté	Syntaxe explicite pour traverser et modifier	Syntaxe plus succincte, similaire à la STL
Performances	Légèrement plus lourd en raison des appels de méthode	Plus léger, équivalent aux itérateurs STL

4.3.3. Avantages des itérateurs

Les itérateurs offrent plusieurs avantages :

- **Abstraction** : Ils masquent la structure interne des conteneurs, permettant de les parcourir sans se soucier des détails d'implémentation.
- **Flexibilité** : Ils permettent de manipuler des collections de manière générique, indépendamment du type de conteneur (liste, tableau, carte, etc.).
- **Modifiabilité** : Avec les itérateurs modifiables, vous pouvez parcourir et modifier une collection en même temps.
- **Performances** : Les itérateurs, en particulier ceux qui sont implicites, sont conçus pour être efficaces en termes de parcours, en minimisant la surcharge.

4.4. La macro foreach

La bibliothèque Qt offre la possibilité d'utiliser la macro `foreach`. C'est une manière pratique et concise de parcourir les éléments d'un conteneur sans avoir à utiliser explicitement des itérateurs. Elle simplifie l'itération sur les conteneurs tels que **QList**, **QVector**, **QMap**, **QSet**, et autres.

4.4.1. La syntaxe de la macro `foreach` est la suivante :

Syntaxe macro foreach

```
foreach (type variable, container) {
    // Code à exécuter pour chaque élément
}
```

- **type** : Le type de l'élément dans le conteneur (par exemple, `int`, `QString`, `QPair<Key, Value>`, etc.).
- **variable** : Une variable qui représentera chaque élément lors de l'itération.
- **container** : Le conteneur à parcourir (par exemple, `QList<int>`, `QMap<QString, int>`, etc.).

Exemple simple avec `QList`

```
QList<int> list = {1, 2, 3, 4, 5};
foreach (int value, list) {
    qDebug() << value; // Affiche 1, 2, 3, 4, 5
}
```

Dans cet exemple, la macro foreach itère automatiquement sur chaque élément de la liste list et assigne chaque élément à la variable value.

Exemple avec un conteneur associatif (`QMap`)

```
QMap<QString, int> map;
map["Apple"] = 50;
map["Banana"] = 30;
map["Cherry"] = 20;
foreach (const QString &key, map.keys()) {
    qDebug() << key << ":" << map.value(key);
}
```

Dans cet exemple, on utilise `map.keys()` pour obtenir une liste de clés, puis on itère sur ces clés pour accéder aux valeurs correspondantes.

4.4.2. Limites de la macro `foreach`

Bien que la macro `foreach` soit pratique, elle présente quelques limites qu'il est important de garder à l'esprit :

Copie des éléments :

- La macro `foreach` crée par défaut une **copie** des éléments du conteneur. Cela signifie que les modifications faites à la variable dans la boucle ne sont pas répercutées dans le conteneur original.
- Pour éviter cela et améliorer les performances, il est recommandé d'utiliser des références (avec `&`) si vous n'avez pas besoin de modifier les éléments, ou des itérateurs si vous souhaitez les modifier.

Exemple avec référence :

```
QList<QString> list = {"Alice", "Bob", "Charlie"};
foreach (const QString &name, list) {
    qDebug() << name;
}
```

Pas de modification directe du conteneur :

- Il n'est pas possible de modifier directement le conteneur (ajouter, supprimer des éléments) dans une boucle `foreach`. Pour cela, il faut utiliser des itérateurs explicites ou des boucles `for` classiques.

Exemple où l'on ne peut pas supprimer un élément :

```
QList<int> list = {1, 2, 3, 4, 5};
foreach (int value, list) {
    if (value % 2 == 0) {
        list.removeOne(value); // Ce n'est pas autorisé
    }
}
```

Utilisation déconseillée dans les nouveaux projets :

- Avec les versions modernes de Qt et C++, la boucle `foreach` a été progressivement remplacée par la boucle `for` basée sur la plage introduite avec C++11. Cette dernière est plus efficace et ne copie pas les éléments par défaut.

```
QList<int> list = {1, 2, 3, 4, 5};
for (int value : list) {
    qDebug() << value;
}
```

Conclusion

La macro `foreach` de Qt est un moyen rapide et pratique d'itérer sur les éléments d'un conteneur sans avoir à écrire des boucles ou à manipuler des itérateurs directement. Cependant, elle copie les éléments du conteneur, ce qui peut poser des problèmes de performance et de modification. Dans les projets modernes, il est souvent préférable d'utiliser la boucle `for` basée sur la plage (`range-based for loop`) introduite avec C++11, qui offre une alternative plus performante et expressive.