Les fonctionnalités de base des collections

Le chapitre précédent présentait chaque collection et ses spécificités, mais sans entrer dans les détails. Certaines fonctionnalités étant commune a plusieurs collections, ces fonctionnalités ne vont pas être décrites par collection, mais par catégories de fonctionnalités. N'hésitez pas a consulter la documentation pour savoir si une collection en particulier permet d'utiliser une fonctionnalité ou non.

Certaines fonctionnalités ont déjà étaient décrites dans les chapitres précédents, ce chapitre ne sera qu'un rappel dans ce cas.

Sémantique de collection et des éléments

Vous avez déjà utilise a plusieurs reprises des collections dans les chapitres précédents. Vous avez par exemple vu qu'une collection pouvait être comparée par égalité (`std::equal`), être triée (en utilisant la comparaison "plus petit que" `std::less`) ou être copiée (`std::copy`).

En fait, les fonctionnalités proposées par les collections dépendent des fonctionnalités possibles des éléments qu'elle contient. Par exemple, copier une collection signifie en fait copier les éléments un par un. Pour copier une collection, il faut donc que les éléments soient copiables.

main.cpp

```
#include <vector>
#include <memory>

int main() {
    using copiable_collection = std::vector<std::shared_ptr<
int>>;
    copiable_collection cc1 {};
    copiable_collection cc2 = cc1;
```

```
    using noncopiable_collection = std::vector<std::
unique_ptr<int>>;
    noncopiable_collection uc1 {};
    noncopiable_collection uc2 = uc1;
}
```

(Ne vous inquiétez pas pour `std::shared_ptr` et `std::unique_ptr`, vous verrez ces classes plus tard. Elles sont utilisées dans cet exemple uniquement parce que la premiere est copiable et pas la seconde.)

Dans ce code, avec une classe copiable (`std::shared_ptr`), la copie sera autorisée (`cc2 = cc1`). Au contraire, avec une classe non copiable (`std::unique_ptr`), essayer de copier de collection (`uc2 = uc1`) produira une erreur de compilation.

```
/usr/local/include/c++/5.3.0/bits/unique_ptr.h:356:7: note:
declared here
       unique_ptr(const unique_ptr&) = delete;
       ^
```

(Le message d'erreur ne dit pas explicitement que `std::unique_ptr` n'est pas copiable, il indique qu'une fonction particulière, le constructeur par copie, est supprimée. Les messages d'erreur en C++ sont parfois difficile a comprendre, cela fait partie de l'apprentissage du C++ d'apprendre a les comprendre.)

N'oubliez pas que les fonctionnalités décrites dans ce chapitre dépendent donc du type d'élément que vous utiliserez dans une collection.

Créer une collection

Construction par defaut

Toutes les collections de la bibliothèque standard sont constructibles par défaut (DefaultConstructible), c'est a dire sans aucun parametre ou de liste de valeurs.

```cpp
std::vector<int> v {};
std::list<double> l {};
std::map<char, std::string> m {};
```

Syntaxe alternative (rappel)
Pour rappel, il existe d'autres syntaxes possibles pour créer une variable par défaut, qui ne sont pas recommandées, mais que vous pouvez rencontrer dans un ancien code C++.

```cpp
std::vector<int> v; // sans accolades
```

Autre rappel, il est classique de faire l'erreur d'utiliser des parentheses. Cependant, cela ne permet pas de créer une collection par défaut. (Cela permet en fait de déclarer une fonction.)

```cpp
std::vector<int> v(); // erreur, ce code declare une
fonction
```

Copie et déplacement

Les collections sont également copiables (*Copy*) et déplaçables (*Move*), par construction (*Constructible*) et assignation (*Assignable*). Ce qui fait que les collections peuvent être CopyConstructible, MoveConstructible, CopyAssignable ou MoveAssignable.

Lorsqu'une collection est copiée dans une autre collection, cela implique que chaque élément de la première collection va être copie dans la seconde collection. Au final, les deux collections contiendront la même liste d'éléments.

Lorsqu'une collection est déplacée dans une autre collection (par exemple en utilisant la fonction `std::move`, qui se traduit par "déplacer"), cela signifie que les éléments sont retires de la première collection pour être déplacer dans la seconde. Au final, la première collection sera vide et la seconde contiendra les éléments qui se

trouvaient précédemment dans la première collection.

main.cpp

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v1 { 1, 2, 3 };

    std::vector<int> v2 = v1;             // copie
    std::cout << v1.size() << ' ' << v2.size() << std::endl;

    std::vector<int> v3 = std::move(v1);  // deplacement
    std::cout << v1.size() << ' ' << v3.size() << std::endl;
}
```

affiche :

```
3 3
0 3
```

Apres la copie, les collections `v1` et `v2` contiennent tous les deux trois éléments. Apres le déplacement, la collection `v1` ne contient plus d'élément, alors que `v3` contient trois éléments.

Les notions de copie et déplacement d'objets sont très importantes en C++, en particulier pour la gestion de la durée de vie des objets. Cela sera approfondi dans un chapitre dans la suite de ce cours.

assignation

assign, paire d'iterateurs

Liste de valeurs

std::vector<int> v { 1, 2, 3, 4 };

Taille initiale

std::vector<int> v (5); std::vector<int> v (5, 123);

Attention entre v(5) et v{5}.

resize + reserve

Taille et capacite

Size, capacité, empty, shrink_to_fit. swap

Ajouter et supprimer des éléments

Ajout et suppression d'éléments. L'idiome remove-erase. emplace vs push/insert

Complexité algo

Accéder aux éléments

Accès aux éléments (random access, front, back)

Les autres fonctions membres

Autres fonctions membre (find, count, etc)

Itérateurs

allocator, data()

Chapitre précédent	Sommaire principal	Chapitre suivant