Le concept de collections de données

L'une des difficultés principales que vous aurez à résoudre en tant que développeur est de définir comment les données doivent être organisées en mémoire (structures de données) et comment ces données doivent être traitées (algorithmes). Il est intéressant, dès que cela est possible, de séparer ces deux aspects du problème dans des classes différentes, pour renforcer la réutilisabilité du code que vous écrivez. La bibliothèque standard du C++ est organisée de cette façon, avec d'un côté des classes de structures de données (`std::string` pour les chaînes, `std::vector` pour les tableaux, etc.) et des fonctions libres pour le traitement des données (regroupées dans le fichier d'en-tête `<algorithms>`).

Pour qu'un code soit réutilisable au maximum, l'idéal serait que n'importe quelle structure de données soit compatible avec n'importe quel algorithme. Dit autrement, cela veut dire que si vous créez une structure de données, elle doit être utilisable avec n'importe quel algorithme de la bibliothèque standard et que si vous créez un algorithme, il doit être utilisable avec n'importe quelle structure de données de la bibliothèque standard. Pour cela, les structures de données sont conçues autour du concept de collection, que vous allez voir dans ce chapitre.

Les chaînes comme collection de caractères

Les chaînes de caractères `string`, que vous avez manipulées dans les chapitres précédents, sont un exemple de collection. Une collection est un ensemble d'éléments (`string` est un ensemble de caractères) qui respecte les propriétés suivantes :

- accéder au début de la collection en utilisant la fonction `begin` (en fonction libre ou en fonction membre) ;

- accéder à la fin de la collection en utilisant la fonction `end` (également en fonction libre ou en fonction membre) ;
- respecter la notion “élément suivant”, c'est-à-dire que chaque élément d'une collection possède un et un seul élément suivant. Cette élément est accessible en utilisant l'opérateur `++` ou la fonction libre `next`.

Il est alors possible de parcourir une collection du début à la fin en passant d'un élément au suivant. On parle d'accès séquentiel dans ce cas.

Pour rappel, une fonction membre s'écrit en utilisant l'opérateur `.` entre le nom d'une variable et la fonction membre. Une fonction libre s'applique sur une variable en la donnant en argument entre parenthèses. Les deux syntaxes sont identiques en termes de comportement du programme.

```
std::string const s {};

// début de la collection
s.begin();      // fonction membre
std::begin(s);  // fonction libre

// fin de la collection
s.end();        // fonction membre
std::end(s);    // fonction libre
```

Les algorithmes de la bibliothèque standard s'appliquent sur une collection en donnant en argument le premier et le dernier élément de la collection sur laquelle on souhaite appliquer l'algorithme. Par exemple, pour appliquer l'algorithme `std::sort` (qui permet de trier les éléments d'une collection), on écrira :

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string s { "azerty" };
    std::sort(std::begin(s), std::end(s));
```

```
    std::cout << s << std::endl;
}
```

affiche :

```
aertyz
```

L'inclusion de l'en-tête `<algorithm>` permet d'utiliser les algorithmes de la bibliothèques standard.

On peut se demander pourquoi passer en argument des fonctions le début et la fin d'une collection, au lieu de passer la collection complète en argument (en écrivant par exemple `std::sort(s)`). La raison est que cela permet d'utiliser les algorithmes également sur une partie d'une collection au lieu de la collection complète.
En effet, les algorithmes prennent en paramètre le début et la fin sur lesquels s'appliquent l'algorithme, qui ne sont pas forcément les premier et dernier éléments de la collection. Il est ainsi possible de trier une chaîne à partir du troisième élément en écrivant `std::sort(std::begin(s)+3, std::end(s))`.

Attention, cette syntaxe nécessite que la chaîne contienne au moins trois caractères, sous peine d'obtenir un comportement indéterminé. La manipulation des collections élément par élément sera vu dans un prochain chapitre.

Notez bien que même si `std::string` est une collection, c'est un cas particulier lorsque vous l'utilisez avec `std::cout`. Dans ce cas, `std::string` n'est pas affichée comme un collection, mais bien comme une chaîne de caractères. Ce qui veut dire que dans le cas général de n'importe quelle collection, vous ne pouvez pas afficher directement son contenu avec `std::cout`. Un code d'exemple est donné a la fin de ce chapitre pour afficher le contenu d'une collection.

```
std::vector<int> v { 1, 2, 3 };
std::cout << v << std::endl; // erreur
```

Templates et généricité

Les données que vous aurez à manipuler ne se limiteront pas aux chaînes de caractères. Heureusement, la bibliothèque standard fournit d'autres structures de données, qui permettent de manipuler n'importe quel type de données dans une collection.

La page de documentation des conteneurs de la bibliothèque standard liste une quinzaine de types différents de structures de données. Nous n'allons voir dans ce chapitre que les tableaux de type `std::vector` et `std::array`, les autres conteneurs seront vus dans des prochains chapitres. La différence entre ces deux types de tableau est que `std::array` représente un tableau dont le nombre d'éléments (on parle de la “taille du tableau”) est fixé à la compilation, tandis que le nombre d'éléments de `std::vector` peut être modifié durant l'exécution.

Les conteneurs de données sont des classes, au même titre que `std::string`. Pour déclarer un tableau, on utilisera donc la même syntaxe que pour déclarer une chaîne. Par exemple, pour créer un tableau vide d'entiers, on écrira :

```
std::vector<int> integers {};
```

Dans ce code, `integers` est le nom de la variable qui contient le tableau d'entiers (*integers* signifie “des entiers” - prenez l'habitude d'écrire vos codes en anglais, en particulier pour nommer vos variables, fonctions et classes). Le type de tableau est `vector<int>`, que l'on peut lire directement comme étant un tableau (`vector`) d'entiers (`int`).

La syntaxe de `std::vector` est un peu particulière, c'est ce que l'on appelle une classe template (ou classe générique). Vous avez déjà rencontré cette syntaxe dans le chapitre Obtenir des informations sur les types, pour la classe `std::numeric_limits`. La syntaxe générale de `std::vector` est la suivante :

```
std::vector<TYPE>
```

Imaginons que l'on vous demande de créer un telle classe, qui permette

de manipuler des tableaux de données. Une première approche serait d'écrire une classe représentant un tableau d'entier `vector_int`, puis de la dupliquer et de modifier cette classe pour manipuler un tableau de doubles `vector_double`. Et ainsi de suite pour chaque type que l'on souhaite manipuler.

Cependant, ce type d'approche est très problématique. En premier lieu, cela implique de copier-coller plusieurs fois le code et de le modifier selon le type de données contenu dans le tableau. Si vous corrigez un problème dans l'un des types de tableau, il ne faut pas oublier de corriger également les autres tableaux. C'est un risque d'erreur, le code est moins maintenable.

Le second problème est qu'il ne sera pas possible de prendre en charge tous les types. Un utilisateur de votre code qui crée ses propres types devra copier votre code et le modifier pour prendre en charge ses propres types. C'est une autre source d'erreur importante, le code n'est pas évolutif.

Les templates permettent d'écrire des classes et fonctions dont le comportement dépendra d'informations données à la compilation. Les informations données peuvent être un type ou une valeur entière. Les classes `vector` et `std::array` sont des exemples de template. Ces classes sont conçues pour accepter n'importe quel type de données, en particulier les types que vous créerez. Ce type d'approche présente de nombreux avantages :

- maintenabilité : le code n'est pas dupliqué. S'il doit être corrigé, il suffit de corriger une seule implémentation et tous les codes utilisant cette classe seront corrigés en même temps.
- évolutivité : si vous créez un nouveau type, vous pouvez l'utiliser dans un template (à partir du moment où vous respectez les conditions d'utilisation de ce template).

Les arguments template sont donnés entre chevrons `<>` après le nom de la classe ou de la fonction template. S'il y a plusieurs arguments, ils sont séparés par des virgules.

```
template_class<int>                 // classe template avec 1 argument
another_template_class<int, float> // classe template avec 2 arguments

template_function<int>()            // une fonction template
```

Attention aux arguments passés dans les appels de classes et fonctions. Vous avez donc deux types d'arguments (par exemple pour une fonction) :

```
foo<arg1, arg2>(arg3, arg4);
```

Arguments	Type	Valeurs possible	Évaluation
`arg1` et `arg2`	arguments de template	Types et valeurs entières	Lors de la compilation
`arg3` et `arg4`	arguments de fonction	Variables et littérales	Lors de l'exécution

On voit bien que ces deux types d'arguments répondent à des besoins différents, acceptent des valeurs différentes et ne sont pas évalués en même temps. Il faut faire attention de ne pas les confondre. L'utilisation de deux types d'arguments peut sembler être une complexité inutile, mais en fait, c'est l'une des forces du C++. Cela permet de penser les problèmes à résoudre en termes de ce qui peut être évalué à la compilation ou ce qui peut l'être à l'exécution. Et donc de pouvoir écrire des abstractions très complexes (mais très simples à utiliser), sans perte de performance lors de l'exécution.

Vous apprendrez par la suite différentes approches pour écrire du code générique performant, mais dans un premier temps, voyons comment utiliser de telles classes template.

Les tableaux comme collections

Les tableaux étant des collections, il est possible d'utiliser les fonctions `begin` et `end` pour obtenir respectivement le début et la fin d'un tableau.

Ces fonctions sont également utilisables comme fonctions membres ou fonctions libres. Par exemple, avec la fonction de tri `sort` :

```
std::sort(a.begin(), a.end());          // fonctions membres
std::sort(std::begin(a), std::end(a)); // fonctions libres
```

Au final, vous n'avez besoin que de connaître ces deux fonctions, `begin` et `end`, pour utiliser une collection avec les algorithmes de la bibliothèque standard. Il existe d'autres types de collection que `std::array` et `std::list`, n'hésitez pas à parcourir la page de documentation correspondante aux conteneurs de données et à en tester quelques uns.

Déclarer et initialiser des tableaux

La classe `std::vector` prend un seul argument template, qui est le type de données que le tableau doit contenir. La classe `std::array` prend deux arguments : le type de données et le nombre d'éléments que doit contenir le tableau. Leur syntaxe est donc la suivante :

```
std::vector<TYPE>
std::array<TYPE, TAILLE>
```

Par exemple, pour créer des tableaux :

```
// tableau d'entiers, la taille peut changer à l'exécution
std::vector<int> const integers {};

// tableau de 5 nombres réels, la taille est fixe
std::array<float, 5> const floats {};
```

Par défaut, `std::vector` ne contient pas d'éléments lors de l'initialisation de la variable `integers`. Comme `std::vector` est un tableau de taille redimensionnable, vous pourrez ajouter des éléments par la suite.

Au contraire, `std::array` est initialisé avec cinq éléments dans le code précédent. Il est possible de créer une `std::array` avec aucun élément,

mais comme il n'est pas possible d'ajouter des éléments, l'intérêt est limité.

Remarque : pour rappel, le but de ce cours n'est pas de vous présenter toutes les syntaxes possibles, mais celles qui sont utiles à connaître pour comprendre les bases du C++. Il est possible d'utiliser d'autres syntaxes pour les classes `std::vector` et `std::array`, mais la compréhension de ces syntaxes nécessite des connaissances plus avancées en C++.
Il est possible d'initialiser `std::vector` avec un nombre déterminé d'éléments, comme pour `std::array`. Pour cela, il faut donner ce nombre entre parenthèses :

```
std::vector<int> const integers(5); // crée un tableau contenant 5 éléments
```

On retrouve ici la différence de syntaxe entre argument *template* et argument de fonction. Pour `std::array`, la taille du tableau (nombre d'éléments) est fixée à la compilation, c'est donc un argument *template* (entre chevrons). Pour `std::vector`, la taille est variable durant l'exécution, c'est donc un argument de fonction (entre parenthèses).

Accolades et parentheses
Remarquez bien ici l'utilisation des parenthèses au lieu des accolades, ce n'est pas une erreur de frappe.

Les accolades permettent de définir une liste de valeurs, qui seront insérées directement dans une collection. Les parenthèses permettent d'appeler un constructeur (une fonction spéciale permettant d'initialiser une classe).

Dans les deux cas, il est possible de connaître la taille d'un tableau en utilisant la fonction membre `size`.

main.cpp

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

int main() {
    std::vector<int> const integers(5);
    std::array<float, 3> const floats {};
    std::cout << "Size of vector is " << integers.size() <<
std::endl;
    std::cout << "Size of array is " << floats.size() << std
::endl;
}
```

affiche :

```
Size of vector is 5
Size of array is 3
```

Vous pouvez donner des valeurs entre les accolades pour initialiser le tableau. Une liste de valeurs (*initializer-list*) s'écrit entre accolades, avec des virgules comme séparateurs.

```cpp
std::vector<int> const integers { 0, 1, 2, 3 };
std::array<int, 5> const floats { 0, 1, 2, 3 };
```

Remarque : pour rappel, l'utilisation des espaces est libre en C++. Le critère principal doit être la lisibilité du code. Il est tout à fait possible d'écrire les listes de valeurs sous forme compacte `{0,1,2,3}`. Peu importe comment vous souhaitez présenter vos codes, mais essayez de respecter des conventions d'écriture (même informelles) dans vos codes, pour que les syntaxes soient homogènes.

Dans ce code, les deux tableaux sont initialisés avec les valeurs 0 à 3. `std::vector` étant un tableau dynamique, sa taille est initialisée en fonction du nombre d'éléments donné dans la liste. Ce qui signifie que `integers` contient 4 éléments dans ce code. Au contraire, la taille de `std::array` est fixée par le second argument template et non par le nombre d'éléments dans la liste. `floats` contient donc 5 éléments, les 4 premiers correspondant aux valeurs données dans la liste, le dernier est initialisé par défaut (avec la valeur 0 donc).

Modifier la taille d'un tableau

Comme `std::array` est un tableau de taille fixée à la compilation, il n'est pas possible d'ajouter ou de supprimer des éléments lors de l'exécution du programme. Cette partie ne s'applique donc qu'aux tableaux de type `vector`.

Les tableaux de type `std::vector` peuvent être manipulés comme des piles. Une pile est une collection qui permet de lire, supprimer (*pop*) ou ajouter (*push*) des éléments à la fin de la collection (*back*). Les noms des fonctions membres sont assez simples à comprendre (même si vous ne comprenez pas très bien l'anglais) :

- `back` permet d'accéder au dernier élément, pour le lire ou le modifier ;
- `push_back` permet d'ajouter (*push*) un élément à la fin de la pile ;
- `pop_back` permet de retirer (*pop*) l'élément qui se trouve à la fin de la pile.

Attention, il est nécessaire que la collection ne soit pas vide pour appeler `pop_back`, sinon cela provoque un comportement indéfini (*undefined behavior*). Aucun message d'erreur (à la compilation ou à l'exécution) n'est affiché si vous utilisez `pop_back` sur un tableau vide.

Notez aussi qu'il est possible de lire et modifier le premier élément d'un tableau en utilisant la fonction membre `front`.

main.cpp

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

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

    v.push_back(4);
    std::cout << v.back() << std::endl;
```

```
    // v contient { 1, 2, 3, 4 }

    v.back() = 5;
    std::cout << v.back() << std::endl;
    // v contient { 1, 2, 3, 5 }

    v.pop_back();
    v.pop_back();
    std::cout << v.back() << std::endl;
    // v contient { 1, 2 }
}
```

affiche :

```
4
5
2
```

Il existe également une collection qui permet d'être agrandie ou rétrécie depuis le premier et le dernier élément : std::deque. Son utilisation est assez similaire à `std::vector`, vous verrez dans les exercices d'application comment l'utiliser.

Afficher le contenu d'une collection

Pour afficher le contenu d'une collection, il n'est pas possible d'utiliser directement `std::cout` dessus. Il faut afficher individuellement chaque élément avec `std::cout`, ce qui implique de voir comment parcourir une collection. Vous verrez cela dans la suite du cours, pour le moment, voici deux syntaxes pour faire cela.

Avec un algorithme

Une première solution est en fait de copier (avec l'algorithme `std::copy`) chaque élément dans le flux de sortie standard (`std::cout`).

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> const i { 1, 2, 3, 4 };
    copy(begin(i), end(i), std::ostream_iterator<int>(std::
cout, "\t"));
    std::cout << std::endl;

    std::vector<double> const d { 1.2, 3.4, 5.6, 7.8 };
    copy(begin(d), end(d), std::ostream_iterator<double>(std
::cout, "\t"));
    std::cout << std::endl;

    std::vector<std::string> const s { "un", "deux", "trois",
"quatre" };
    copy(begin(s), end(s), std::ostream_iterator<std::string
>(std::cout, "\t"));
    std::cout << std::endl;
}
```

affiche :

```
1   2    3    4
1.2 3.4    5.6    7.8
un  deux    trois    quatre
```

Avec une boucle

Pour accéder à chaque élément individuellement, il est nécessaire de faire intervenir la notion d'itérateur et de boucles, ce qui sera vu dans la suite de ce cours. Cependant, il est intéressant de voir rapidement la syntaxe permettant d'afficher le contenu d'une collection. Pour cela, vous pouvez utiliser la syntaxe suivante :

```
for (auto const value: integers )
    std::cout << value << std::endl;
```

Ce code se lit de la façon suivante : “pour (*for*) chaque élément d'une collection (`integers` dans ce code), afficher la valeur avec `std::cout`”. Ce code est relativement générique, vous pouvez l'utiliser avec n'importe quelle collection de données dont le type peut être affiché avec `std::cout`.

main.cpp

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<int> const i { 1, 2, 3, 4 };
    for (auto const value: i)
        std::cout << value << '\t';
    std::cout << std::endl;

    std::vector<double> const d { 1.2, 3.4, 5.6, 7.8 };
    for (auto const value: d)
        std::cout << value << '\t';
    std::cout << std::endl;

    std::vector<std::string> const s { "un", "deux", "trois",
"quatre" };
    for (auto const value: s)
        std::cout << value << '\t';
    std::cout << std::endl;
}
```

affiche :

```
1   2    3    4
1.2 3.4    5.6    7.8
un  deux    trois    quatre
```

Bien sûr, comme une chaîne de caractères `std::string` peut être vue comme une collection, il est également possible d'utiliser cette syntaxe

pour afficher les caractères individuellement.

main.cpp

```cpp
#include <iostream>
#include <string>

int main() {
    std::string const s { "hello, world!" };
    for (auto const value: s)
        std::cout << value << ' ';
}
```

affiche :

```
h e l l o ,   w o r l d !
```

Chapitre précédent	Sommaire principal	Chapitre suivant