Les collections de données

L'une des difficultés principales que vous aurez a résoudre en tant que développeur est de définir comme les données doivent être organisées en mémoire (structures de données) et comment ces données doivent être traités (algorithmes). Il est intéressant, que cela est possible, de séparer ces deux aspects du problèmes dans des classes différentes, pour renforcer la réutilisabilité du code que vous écrivez. La bibliothèque standard du C++ est organisé de cette façon, avec d'un côté des classes de structures de données (`string` pour les chaînes, `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 une algorithme, il doit être utilisable avec n'importe quelle structure de donnés 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 manipulé dans les chapitres précédents, sont un exemple de collection. Une collection est une ensemble d'éléments (`string` est un ensemble de caractères) qui respecte les propriétés suivantes :

- avoir un premier élément, accessible en utilisant la fonction `begin` (en fonction libre ou en fonction membre) ;
- avoir un dernier élément, accessible 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 depuis le premier élément jusqu'au dernier en passant d'un élément au suivant.

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 {};

// premier élément
s.begin(); // fonction membre
begin(s);  // fonction libre

// dernier élément
s.end();   // fonction membre
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 `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(begin(s), 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 premier et le dernier élément 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 premier et le dernier élément sur lesquels s'appliquent l'algorithme, qui ne sont pas forcement les premier et dernier élément de la collection. Il est ainsi possible de trier une chaîne à partir du troisième élément en écrivant `std::sort(begin(s)+3, 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.

Généralisation des collections : les tableaux

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 quelle type de données dans une collection.

La page de documentation des conteneurs de la bibliothèques standard liste une quinzaine de types différents de structures de données. Nous allons voir dans ce chapitre que les tableaux de type `vector` et `array`, les autres conteneurs seront vu dans des prochains chapitres. La différence entre ces deux types de tableau est que `array` représente un

tableau dont le nombre d'éléments (on parle de “taille du tableau”) est fixé à la compilation, tandis que le nombre d'élément de `vector` peut être modifié durant l'exécution.

Templates et généricité

Les conteneurs de données sont de 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 la noms 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 comment étant un tableau (`vector`) d'entiers (`int`).

La syntaxe de `vector` est un peu particulier. 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`.

Imaginons que l'on vous demander de créer un telle classe, qui permet de manipuler des tableaux de données. Une première approche serait d'écrire une classe représentant un tableau d'entier `vector_int`, puis dupliquer et 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 la 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 une 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 `array` sont un exemple 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 respecter les conditions d'utilisation de ce template).

Les arguments template sont donnés entre chevrons `<>` après le noms 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 évaluer 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 évaluer à la compilation ou ce que peut l'être à l'exécution. Et donc de pouvoir écrire des abstractions très complexes (mais très simple à utiliser), sans perte de performance lors de l'exécution.

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

Déclarer et initialiser des tableaux

La classe `vector` prend un seul argument template, qui est le type de données que le tableau doit contenir. La classe `array` prend deux arguments : le type de données et le nombre d'éléments que doit contenir le tableau. Par exemple, pour créer des tableaux vides :

```
std::vector<int> integers {};
std::array<int, 5> floats {};
```

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 `vector` et `array`, mais la compréhension de ces syntaxes nécessites des connaissances plus avancées en C++.

Il est possible d'initialiser `vector` avec un nombre déterminé d'éléments, comme pour `array`. Pour cela, il faut donner ce nombre entre parenthèses :

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

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

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

main.cpp

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

int main() {
    std::vector<int> integers(5);
    std::array<float, 3> 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 crochets pour initialiser le tableau. La liste des valeurs est séparées par des virgules. Remarquez que la syntaxe est un peu différente de `string`, il faut utiliser un signe `=` dans ce cas.

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

ou syntaxe avec double bracket ?

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 sous forme compact `{0,1,2,3}`. Essayer de respecter des règles d'écriture (même informelles) dans vos code , pour que les syntaxes soient homogènes.

Dans ce code, les deux tableaux sont initialisés avec les valeurs 0 à 3. `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 `array` est fixé par le second argument template et pas 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).

afficher un tableau :

```
for (auto e: a)
    std::cout << a << std::endl;
```

Modifier un tableau

push, pop, etc

Les tableaux comme collection

begin et end

sort(a.begin(), a.end());

Chapitre précédent	Sommaire principal	Chapitre suivant