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.

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.

problèmes : duplication du code (moins maintenable) et pas prise en charge de n'importe quel type (par évolutif). Solution, écrire une classe qui prend en paramètre un type. Donné en <>. Ainsi, vector<int>, tableau d'entiers, vector<double> tableau de doubles, etc. Utilisable avec n'importe quel type

begin/end