Le chapitre précédent présentait le premier type de structure de contrôle permettant de ne pas avoir une séquence linéaire d'instructions. Ce chapitre présente le second type de structure de contrôle, permettant de répéter plusieurs fois une même séquence d'instructions.

La notion de “répéter une tâche” vous est familier dans la vie courante. Par exemple, compter de un à dix sur ses doigts peut être considéré comme une boucle, dans laquelle vous comptez un doigt supplémentaire à chaque itération et vous vous arrêtez lorsque vous avez comptez tous vos doigts. Pour lire un livre, vous allez lire une page, passer à la suivante et recommencer, jusqu'a la derniere pages. Si vous jouez à un jeu vidéo, vous allez avancer votre personnage, évaluer la situation, agir en conséquence, puis recommencer jusqu'à arriver à la fin du jeu.

En programmation, une boucle est similaire : répéter une tâche (une séquence d'instructions) jusqu'à atteindre le but recherché.

Les boucles sont particulièrement utiles avec les collections, pour réaliser une tâche quelconque sur chaque élément de la collection. Pour illustrer cela, le cas du tableau `std::vector` sera detaillé comme exemple.

Les instructions iteratives

Classiquement, une boucle peut etre decomposer en trois parties :

- l'initialisation, c'est-à-dire ce qui sera réalisé avant de démarrer la boucle ;
- les instructions à réaliser a chaque itération de la boucle ;
- une condition à vérifier pour terminer la boucle.

Par exemple, pour compter de un à dix, vous devrez :

- initialiser un compteur avec la valeur `1` ;
- a chaque itération, incrémenter le compteur ;
- lorsque le compteur est égal à la valeur `10`, la boucle s'arrête.

Chaque étape est importante pour que la tâche soit correctement réalisée. Si vous réinitialisez le compteur avec la valeur `5`, vous aurez compte que cinq fois lorsque la boucle s'arretera. Idem si incrément de deux le compteur a chaque itération. Et la condition d'arrêt n'est pas correcte, le compteur ne sera pas non plus correct.

Cependant, les erreurs mentionnées précédemment produisent “simplement” un résultat faux. Il existe en réalité un problème plus grave : les boucles infinies, qui ne se terminent jamais. Une telle boucle mène au blocage du programme, puis à son crash (automatique lorsque la mémoire est saturée, ou provoquée par l'utilisateur).

Il n'est pas possible de proposer une syntaxe, quelque soit le langage de programmation, qui empêcherait celui qui écrit le code de ne pas faire d'erreur de logique et d'ecrire une boucle qui ne realise pas la tache accomplie. (C'est le rôle des tests unitaires de vérifier cela, vous verrez cela dans un prochain chapitre).

Par contre, il est possible de proposer des syntaxes qui limiteront le risque d'ecrire des boucles infinies. Pour cette raison, il existe plusieurs syntaxes possible en C++ pour ecrire des boucles. Dans ce chapitre, les syntaxes seront présentées dans l'ordre du plus sécurisé au moins sécurisé.

Les boucles range-for

Syntaxe de base

Les boucles *for* sur un intervalle (*range-based for loop*, qui est souvent simplifie en *range-for*) permettent de parcourir la totalité d'une collection. La syntaxe est la suivante :

```
for(TYPE ELEMENT: COLLECTION) {
    ...
}
```

Dans cette syntaxe, `COLLECTION` correspond à la collection sur laquelle la boucle est appliquée. `ELEMENT` est une variable locale que vous pouvez utiliser dans la boucle et qui correspond à chaque élément de la collection. Par exemple, si vous avec une collection qui contient les valeurs `1`, `2`, `3`, la variable `ELEMENT` aura la valeur `1` lors de la premiere iteration, puis la valeur `2` lors de la deuxième itération, puis pour terminer la valeur `3` lors de la troisième itération.

Par exemple, pour parcourir un tableau :

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

int main() {
    std::vector<int> v { 1, 2, 3 };
    for(int i: v) {
        std::cout << i << std::endl;
    }
}
```

affiche :

```
1
2
3
```

Le type de l'élément

Le type `TYPE` correspond au type de la variable représentant chaque élément dans la boucle. Cela peut être le type de l'élément dans la collection ou n'importe quel type convertible.

Par exemple, pour une collection de type `std::vector<int>`, il est possible d'utiliser le type `int` (comme dans l'exemple précédent), mais également `int&` (référence sur un entier), `long int` (conversion vers un type plus large).

Note : il est egalement possible d'utiliser un type moins large (par

exemple `short int`), mais cela peut produire un message d'avertissement dans ce cas.

```
warning: implicit conversion loses integer precision: 'int'
to 'short' [-Wconversion]
    for (short int i: v) {
                  ~^
```

Sauf cas particulier, il n'est pas utile de réaliser une conversion de type et il est plus simple d'utiliser exactement le même type dans la boucle *range-for* que le type de l'élément dans la collection (donc pour une collection de type `std::vector<int>`, utiliser `int`). De fait, le plus simple est d'utiliser la déduction de type dans ce cas, ce qui permet en plus d'avoir un code plus évolutif (si vous changez le type de la collection, vous n'avez pas besoin de changer le type utilisé dans la boucle *range-for*).

Avec la déduction de types, le code précédent devient :

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

int main() {
    std::vector<int> v { 1, 2, 3 };
    for(auto i: v) {
        std::cout << i << std::endl;
    }
}
```

Pour rappel, le mot-clé `auto` déduit uniquement le type (`int` dans ce cas), sans conserver les modificateurs de types (`const` ou les references par exemple). Il peut donc être intéressant d'ajouter des modificateurs de types, en fonction du type d'éléments et de ce que vous souhaitez faire. Le choix des modificateurs est la même problématique que celle que vous avez vu pour les parametres de fonctions :

- si vous souhaitez modifier la valeur, il faut utiliser une référence ;
- si vous ne souhaitez pas modifier la valeur et que le type est un

type fondamentale (la copie est peu coûteuse), il faut utiliser un passage par valeur ;
- si vous ne souhaitez pas modifier la valeur et que le type est complexe (la copie est coûteuse), il faut utiliser une référence constante.

```
const vector<int> v { 1, 2, 3, 4 };
for (auto i: v) {
    std::cout << i << std::endl;
}

vector<int> v { 1, 2, 3, 4 };
for (auto & i: v) {
    ++i; // v contient { 2, 3, 4, 5 } apres la boucle
}

const vector<std::string> v { "hello", "world" };
for (auto const& s: v) {
    std::cout << s << std::endl;
}
```

Reference universelle
Une autre solution est d'utiliser une référence universelle. Ce type de référence s'adapte en fonction du type de collection et de l'élément. La syntaxe est la suivante :

```
for (auto && x: v) ...
```

La syntaxe est similaire à une référence sur une *rvalue*, mais en utilisant `auto`.

Avec les collections associatives

Pour rappel, une collection associative est une collection qui associe une valeur a une cle, comme par exemple `std::map`. Ce type de collection ne contient pas des valeurs, mais des paires de clés et valeurs. Il faut donc

adapter l'utilisation des boucles *range-for* en consequence. Pour cela, il faut utiliser les accesseurs `first` et `second` correspondant au type `std::pair` utilise dans une `std::map`.

Par exemple :

main.cpp

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<int, std::string> m { { 1, "hello" }, { 2,
"world" } };
    for (auto const& p: m) {
        std::cout << p.first << ": " << p.second << std::
endl;
    }
}
```

affiche :

```
1: hello
2: world
```

L'utilisation de la déduction de type est particulièrement intéressant dans ce cas, pour éviter de devoir utiliser explicitement `std::pair` (ce qui alourdi la syntaxe). Sans la déduction de types, il faudrait ecrire :

```
for (std::pair<int, std::string> const& p: m) ...
```

Les collections de la bibliothèque standard qui nécessitent l'utilisation de `std::pair` sont celle contenant “map” : `std::map`, `std::multimap`, `std::unordered_map` et `std::unordered_multimap`.

Avec d'autres types de conteneurs

Les boucles *range-for* sont utilisables avec d'autres types de conteneurs

que les collections de la bibliothèque standard :

- les tableaux hérités du C (qui sont à éviter en C++ moderne, sauf cas d'utilisation spécifiques) ;
- les chaines de caracteres (qui sont assimilables à des collections de caractères) ;
- les listes de valeurs (*initializer-list*).

Utiliser une boucle *range-for* avec une chaîne de caractères permet d'accéder directement a chaque caractère.

```
#include <iostream>

int main() {
    for(auto c: "hello") {
        std::cout << c << std::endl;
    }
}
```

affiche :

```
h
e
l
l
o
```

Les listes de valeurs sont particulierement interessantes, puisque cela permet d'ecrire directement une liste de valeur dans une boucle *range-for*. Par exemple :

main.cpp
```
#include <iostream>

int main() {
    for(auto i: { 1, 2, 3, 4 }) {
        std::cout << i << std::endl;
    }
}
```

affiche :

```
1
2
3
4
```

Les vues [C++17]
Dans la prochaine norme du C++ (le C++17) est ajouté un nouveau concept : les vues. Les vues sont des collections qui permettent d'accéder aux éléments d'une autre collection. Cela permet de créer une sous-collection, sans copier les donnees.

```
std::string      s1 { "hello, world" };
std::string      s2 { std::begin(s), std::begin(s) + 5 }; //
copie contenant "hello"
std::string_view s3 { std::begin(s), std::begin(s) + 5 }; //
vue sur la sous-chaîne "hello"

for (auto & c: s2) {
    c = std::to_upper(c);
}
std::cout << s1 << std::endl; // affiche "hello, world"

for (auto & c: s3) {
    c = std::to_upper(c);
}
std::cout << s1 << std::endl; // affiche "HELLO, world"
```

Interlude : les algorithmes standards

Les algorithmes de la bibliothèques standard **ne sont pas** des instructions iteratives. Mais beaucoup utilisent en interne des boucles et peuvent être utilisés à leur place, lorsque vous travaillez sur des collections.

En termes de sécurité du code, les algorithmes prennent généralement

en paramètre des paires d'itérations. Vous avez donc la garantie qu'ils ne seront pas appelés sur des collections de types differents.

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

int main() {
    std::vector<int> v;
    std::vector<double> w;
    std::sort(std::begin(v), std::end(w));
}
```

affiche un message indiquant qu'il y a un conflit pour la déduction du type d'itérateur (*deduced conflicting types*) :

```
main.cpp:8:5: error: no matching function for call to 'sort'
    std::sort(std::begin(v), std::end(w));
    ^~~~~~~~
(...) note: candidate template ignored: deduced conflicting
types for parameter '_RandomAccessIterator'
```

Par contre, les algorithmes ne garantissent pas :

- que les itérateurs proviennent de la même collection ;
- que les itérateurs sont dans le bon ordre.

Par exemple :

```
std::vector<int> v;
std::vector<int> w;

std::sort(std::begin(v), std::end(w));  // (#1)

std::sort(std::end(v), std::begin(v));  // (#2)
```

Le premier appel a la fonction `std::sort` est invalide du fait que les deux itérateurs proviennent de collections différentes. Le second appel à la fonction `std::sort` est invalide du fait que le premier itérateur correspond a un élément qui se trouve après l'élément correspondant au

second itérateur.

Les ranges [C++2x]
Dans une prochaine norme du C++, il sera possible d'utiliser le concept de *range*, qui sont sémantiquement des paires d'itérations, ce qui apportent les deux garanties manquantes aux algorithmes.

Dans de nombreux cas, il sera plus intéressant et sécurisé d'utiliser les algorithmes standards (avec ou sans prédicat personnalisé) que les syntaxes itératives qui sont décrits dans la suite de ce chapitre.

L'intérêt des algorithmes par rapport à la boucle *range-for* est qu'ils permettent de travailler sur des sous-ensembles d'une collection. En particulier, l'algorithme `std::for_each` peut remplacer une boucle *range-for* sur un sous-collection (en attendant l'arrivée de vues en C++).

```
#include <iostream>
#include <array>
#include <numeric>
#include <iterator>

int main() {
    std::array<char, 26> alphabet;
    std::iota(std::begin(alphabet), std::end(alphabet), 'a');
    std::copy(std::begin(alphabet), std::end(alphabet),
              std::ostream_iterator<char>(std::cout, " "));
}
```

affiche :

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

La boucle for

Les syntaxes précédentes sont les plus sûres, mais si vous recherchez des exemples de code sur Internet, vous les rencontrerez moins souvent que les syntaxes qui suivent. Cela s'explique par le fait que les boucles

range-for sont un ajout “récent” du C++11 (donc plus de 5 ans...), que les algorithmes standards s'utilisent régulièrement avec les fonctions *lambdas* (qui sont egalement un ajout du C++11), et que ces deux syntaxes n'existent pas en C.

Dans les syntaxes précédentes, les trois composantes d'une itération (initialisation, incrémentation et condition d'arrêt) sont implicites, ce qui évite de faire des erreurs en les écrivant. La boucle *for* permet d'ecrire explicitement ces trois composantes. La syntaxe générale est la suivante :

```
for (INITIALISATION; TEST_CONTINUATION; INCREMENTATION) {
    ...
}
```

Une particularité de la boucle *for* : il s'agit d'un test de continuation, et non un test d'arret. C'est-à-dire que la boucle continuera tant que la condition est vraie (au lieu de s'arrêter dès que la condition est vraie).

Par exemple, pour compter de 1 à 10, il faut :

- initialiser une variable avec la valeur 1 ;
- incrementer de 1 cette variable ;
- tant que la variable est inférieure ou égale à la valeur 10, continuer la boucle.

Chacune de ces composants sont facilement traduit en C++ :

- initialisation : `int i { 1 }` ;
- incrementation : `++i` ;
- condition d'arret : `i <= 10`.

En utilisant ces syntaxes dans un boucle *for*, vous obtenez donc :

```
#include <iostream>

int main() {
    for (int i { 1 }; i <= 10; ++i) {
        std::cout << i << std::endl;
    }
```

```
}
```

affiche :

```
1
2
3
4
5
6
7
8
9
10
```

La syntaxe de la boucle *for* est relativement compacte, et donc il est assez facile de se tromper. Vous devez vous habituer à lire correctement cette syntaxe, en identifiant bien chaque composant de la boucle (initialisation, incrémentation, condition d'arrêt). N'hésitez pas à bien aérer votre code, pour le rendre le plus lisible possible.

Parcourir une collection

Un cas d'utilisation classique de la boucle *for* est de parcourir une collection (lorsque la boucle *range-for* n'est pas utilisable). Pour cela, la méthode la plus générique est d'utiliser les itérateurs. Les trois composants de la boucle peuvent s'écrire :

- initialisation : `auto it = std::begin(v)` (ou `std::cbegin`, `std::rbegin`, `std::crbegin`) ;
- incrementation : `++it` ;
- condition d'arret : `it != std::end(v)` (ou `std::cend`, `std::rend`, `std::crend`).

Notez l'utilisation de la déduction de type avec `auto` pour éviter de devoir ecrire le type complet (et long) de l'itérateur. Pour rappel :

```
std::vector<int> v;
std::vector<int>::iterator it { std::begin(v) };
std::vector<int>::const_iterator cit { std::cbegin(v) };
```

Faites bien attention que `std::begin` retourne `iterator` et `std::cbegin`' retourne `const_iterator`.

Le code complet pour parcourir une collection avec des itérateurs est donc le suivant :

main.cpp

```
#include <iostream>
#include <list>

int main() {
    std::list<int> l { 1, 2, 3, 4 };
    for (auto it = std::cbegin(l); it != std::cend(l); ++it)
{
        std::cout << (*it) << std::endl;
    }
}
```

affiche :

```
1
2
3
4
```

Notez que la boucle *range-for* fait exactement la même chose en interne.

Parcourir un tableau

Les tableaux sont des collections et peuvent donc être parcouru en utilisant des itérateurs. Mais il est également possible de les parcourir en utilisant l'indice des éléments et l'opérateur d'indexation `[]`. Dans ce cas, il faut ecrire une boucle *for* basé sur une variable, comme vu précédemment, et l'utiliser pour acceder aux elements.

main.cpp

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

int main() {
    std::vector<char> v { 'a', 'b', 'c', 'd' };
    for (std::size_t i { 0 }; i < v.size(); ++i) {
        std::cout << v[i] << std::endl;
    }
}
```

affiche :

```
a
b
c
d
```

Choix du type pour l'indice

Le type à utiliser pour l'indice devrait être le même type que celui retourné par la fonction `size()`. En toute rigueur, ce type est `std::vector<int>::size_type`.

Cependant, ce type est un peu long à ecrire et équivalent en general a `std::size_t`. Vous verrez donc souvent ce type dans les codes.

Il est possible d'utiliser la déduction de type, mais il faut faire attention. Si vous écrivez `auto i = 0`, la littérale `0` est de type `int` et la variable `i` sera donc aussi de type `int`. Ce qui est une erreur ! (La comparaison d'un type signé et d'un type non signé produit des messages d'avertissement et des comportements parfois inattendus).

Il faut donc utiliser la syntaxe avec `decltype` : `decltype(v.size()) i = 0`. Mais cela ne simplifie pas beaucoup la lecture par rapport à l'écriture explicite du type.

L'utilisation de `std::size_t` est un bon compromis entre lisibilité et rigueur.

Les boucles while et do-while

Les deux dernières syntaxes itératives sont relativement simples à comprendre. Par rapport à la boucle *for*, seule la condition de continuation a une place definie dans ces syntaxes. L'initialisation et l'incrémentation sont placées librement dans le code (voire sont omis).

```
while (TEST_CONTINUATION) {
    ...
}

do {
    ...
} while (TEST_CONTINUATION);
```

La difference entre les boucles *while* et *do-while* est que dans la premiere, la condition de continuation est exécutée avant le corps de la boucle. Il est donc possible de sortir tout de suite de la boucle *while* sans entrer dans la boucle. Dans la seconde syntaxe, le corps de la boucle est exécutée avant le test de continuation. La boucle sera donc toujours exécutée au moins une fois.

En pratique, les boucle *while* et *do-while* sont souvent equivalentes a la boucle *for*, si l'initialisation et l'incrémentation est explicite :

```
INITIALISATION;
while (TEST_CONTINUATION) {
    ...
    INCREMENTATION;
}
```

Programmation structurée

Toutes les syntaxes du C++ ne sont pas présentées dans ce cours. En particulier, concernant le contrôle des flux de donnees, une instruction n'est volontairement non presentee : l'instruction `goto`. Cette instruction permet de passer d'un endroit quelconque d'un programme à un autre

endroit quelconque.

L'utilisation de cette instruction pose un gros problème de lecture du code, il devient rapidement difficile de suivre le flux d'instruction (“code spaghetti”).

Au contraire, n'utiliser que les structures de contrôles permet d'améliorer la lisibilité du code et facilite donc la conception et la maintenance des programmes (en C++ ! L'utilisation de `goto` ou d'un équivalent peut être une pratique acceptable dans d'autres langages de programmation, par exemple en C).

Ce paradigme s'appelle la programmation structurée. Concretement, cela signifie que le code se limite :

- aux séquences d'instructions (une série d'instructions, qui s'exécutent les unes après les autres) ;
- aux instructions conditionnelles (qui permet de choisir entre deux flux d'instructions) ;
- aux instructions itératives (qui permettent de répéter des séquences d'instructions).

Paradigmes
La programmation structurée est un sous-ensemble de la programmation impérative. Pour rappel des paradigmes que vous avez déjà rencontrés :

- la programmation impératives, qui décrit un programme comme un ensemble d'instructions ;
- la programmation procédurale, qui utilise des fonctions (procedure).

Un point intéressant avec la programmation structurée est qu'elle se prete tres bien aux représentation graphiques. Cela permet de représenter un algorithme sous forme visuelle, ce qui facilite la compréhension. Il existe plusieurs représentations graphiques dédiées aux langages de programmation, par exemple les organigrammes (

flowchat en anglais, en vert dans la figure suivante) et les structogrammes (*Nassi–Shneiderman diagram* en anglais, en violet dans la figure suivante).

(Image provenant de l'article de Wikipedia sur la programmation structurée).

Pour aller plus loin, vous pouvez consulter l'article de Wikipedia sur la programmation structurée.

Chapitre précédent	Sommaire principal	Chapitre suivant