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 etre interessant d'ajouter des modificateurs de types, en fonction du type d'elements et de ce que vous souhaitez faire. Le choix des modificateurs est la meme problematique que ce

Syntaxe de base

Directement utilisation sur un liste de valeurs (*initializer-list*) ou une chaine de caracteres (qui peut etre vu comme un tableau de caractères) :

```cpp
#include <iostream>

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

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

affiche :

```
1
2
3
h
e
l
l
o
```

Pour le type `TYPE`, le plus simple est d'utiliser l'inférence de type, ce qui vous permet de changer plus facilement le type de collection sans devoir réécrire la boucle *range-for*. Le plus simple est d'utiliser les références universelles `auto&&`.

Note : futur C++ (ou a implementer soi meme) les vues qui sont un sous-collection d'une collection.

Les algorithmes de la bibliothèque standard

Les algos ne sont pas des instruction iteratives, mais il est bien de les rappeler.

Permet d'appliquer une tache sur chaque element d'une collection. En interne, cela utilise donc bien des iterations. Relativement safe, puisque l'on passe des iterateurs sur le premier element a traiter et le premier element qui ne dois plus l'etre.

```
#include <algorithm>

int main() {
    const std::vector<int> v { 1, 2, 3 };
    const auto sum = std::accumulate(std::begin(v), std::end
(v), 0);
}
```

Toujours un risque d'erreur (se tromper entre begin/end, iterateurs sur collections differentes). Note : futur C++ avec Ranges.

```
std::accumulate(std::end(v), std::begin(v), 0);
std::accumulate(std::begin(v), std::end(w), 0);
```

Utilisation avec range-for, par exemple `iota` pour generer des series, ou `generate` pour generer des nombres aleatoires.

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

int main() {
    std::array<char, 26> alphabet;
    std::iota(alphabet.begin(), alphabet.end(), 'a');

    for(auto&& c: alphabet) {
        std::cout << c << ' ';
    }
    std::cout << std::endl;
}
```

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
```

instruction for

3 éléments, tous optionnels :

- initialisation
- test de continuation
- incrémentation

syntaxe :

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

Par exemple, pour compter de 1 à 10, on utilise un compteur (variable entière) :

- initialisation à 1
- s'arrête lorsque == à 10
- incrémente à chaque boucle

devient :

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

Parcourir un tableau

Avec des iterateurs :

```
for (auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << (*it) << std::endl;
}
```

Pars du premier element (begin). A chaque boucle, passe a l'element suivant (++, next aurait pu convenir aussi). S'arrete lorsque arrive a end.

Condition d'arret = n'est pas executee. Donc quand it == end, la boucle n'est pas executee et (*it) non plus (ca serait invalide).

Pour cela que end(v) ne correspond pas au dernier element d'une collection, mais a l'element “fictif” suivant.

Avec l'operateur [] :

while et do while

Permet de répéter un bloc d'instructions tant qu'une condition est vérifiée. Syntaxe :

```
while (condition) {
    ...
}

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

Dans les 2 cas, exécute le bloc tant que la condition est vraie. Différence entre les 2 : avec do while, bloc exécuté au moins 1 fois avant de test la condition ; avec while, commence par tester

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