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.

Les instructions iteratives

point important : les conditions d'arret. Une boucle se repete jusqu'a ce que la condition d'arret soit valide. Risque de boucle infinie en cas d'erreur sur la condition d'arret.

Utiliser des syntaxes qui limite les risques, en determinant automatiquement les conditions d'arret.

Range-for

Va parcourir tous les elements d'une collection. Aucun risque sur la condition d'arret, puisque determinee directement par l'instruction `for-rnage`.

Syntaxe :

```
for(auto&& ELEMENT: COLLECTION) {
    ...
}
```

“extrait” les elements d'une collection un par un. `auto&& ELEMENT` est la declaration d'une variable “ELEMENT”, dont le type est deduit automatiquement par inference de type. Par exemple, `vector<int>` sera `int` et `list<string>` sera `string`.

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

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

affiche :

```
1
2
3
```

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

```
#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
```

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.

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

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;
}
```

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