Les itérateurs

L'un des points forts des collections et algorithmes de la bibliothèque standard est qu'il est possible d'utiliser n'importe quelle collection avec n'importe quel algorithme (avec quelques restrictions, dues à la nature des algorithmes. Par exemple, `binary_search` ne s'applique par définition que sur des collections triées). Il est même possible d'écrire ses propres collections ou ses propres algorithmes et les utiliser avec ceux de la bibliothèque standard.

Cette flexibilité est obtenue grâce au concept d'itérateur. Le principe est très simple : un itérateur représente un élément dans n'importe quelle collection. En écrivant des algorithmes qui manipulent des itérateurs au lieu de collections particulières, il est ainsi possible d'utiliser les algorithmes sur n'importe quelle collection.

Note : créer un algorithme ne nécessite que d'apprendre la programmation générique, vous verrez cela dans la partie de ce cours “Créer ses algorithmes”. Créer une collection est un peu plus complexe, puisque cela nécessite de connaître la programmation orientée objet. Vous verrez cela plus tard dans le cours.

Manipuler un itérateur

Même si vous n'en aviez pas conscience, vous avez déjà manipuler des itérateurs. En effet, les fonctions `std::begin` et `std::end` permettent d'obtenir un itérateur sur le début et la fin d'une collection. Jusque maintenant, vous utilisiez les itérateurs retournés par ces fonctions directement dans des algorithmes, mais rien n'interdit de créer des variables contenant ces itérateurs.

```
const auto first = std::begin(v);
const auto last = std::end(v);
```

Pour éviter de devoir rappeler à chaque fois la déclaration d'une collection, il est classique d'utiliser des noms pour certains types de variables. Dans ce cours, vous verrez par exemple 'v' pour désigner un `std::vector` ou `it` pour désigner un itérateur (peut importe le type exacte).
Pour déclarer une variable de type itérateur, il est beaucoup plus simple d'utiliser l'inférence de type, comme dans le code précédent. Si vous souhaitez écrire explicitement le type d'itérateur, la syntaxe à utiliser est la suivante :

```
std::vector<int>::iterator first { std::begin(v) };
```

Vous avez déjà vu des syntaxes similaires dans le début de ce cours (par exemple `std::numeric_limits<int>::max()` dans le chapitre Obtenir des informations sur les types). Le code `std::vector<int>::iterator` signifie simplement “le type `iterator` provenant de la classe `vector<int>`”.

Notez bien qu'il faut respecter les collections de provenance des itérateurs. Si vous appeler un algorithme qui attend deux itérateurs d'une même collection, mais que vous lui donnez des itérateurs provenant de deux collections différentes, cela produira un comportement indéterminé (un crash en général dans ce cas).

```
std::sort(std::begin(v1), std::end(v2)); // erreur
```

Reverse et Const-correctness

Il existe en fait plusieurs versions des fonctions `std::begin` et `std::end` :

- les itérateurs “reverse”, qui permettent de parcourir une collection de la fin vers le début ;
- les itérateurs “constants”, qui interdisent de modifier les éléments d'une collection.

Les fonctions “reverse” commencent par “r”, les fonctions “constantes” commencent par “c” et il est possible d'avoir n'importe quelles combinaisons :

- `std::begin` et `std::end` ;
- `std::rbegin` et `std::rend` ;
- `std::cbegin` et `std::cend` ;
- `std::crbegin` et `std::crend`.

Avec le C++14, ces fonctions sont disponibles comme fonctions libres et fonctions membres, les compilateurs plus anciens ne proposeront pas forcement les fonctions libres. Dans ce cas, vous pouvez utiliser les fonctions membres, qui seront toujours disponibles.

```
std::begin(v)  // fonction libre
v.begin() // fonction membre
```

Les itérateurs “reverse” permettent de parcourir une collection depuis la fin vers le début.

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string s { "azerty" };
    std::sort(std::begin(s), std::end(s));
    std::cout << s << std::endl;
    std::sort(std::rbegin(s), std::rend(s));
    std::cout << s << std::endl;
}
```

affiche :

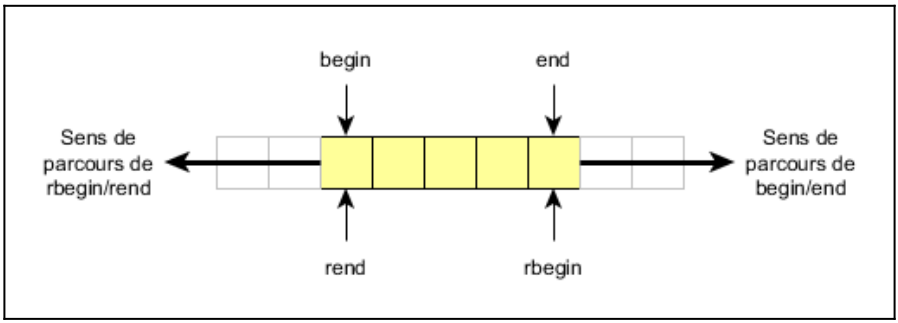
```
aertyz
zytrea
```

L'effet est le même que si vous aviez trié la collection en utilisant l'opérateur “plus grand que” (`std::greater`) au lieu de l'opération par défaut “plus petit que”.

Vous pourriez être tenté, pour parcourir une collection depuis la fin vers le début, d'utiliser un algorithme en donnant `std::end` avant `std::begin`.

```
std::sort(std::end(s), std::begin(s));
```

Vous pouvez essayer ce code : cela produira une erreur à l'exécution. Voici un schéma permet de bien comprendre ce qu'il se passe.



En passant `std::end` puis `std::begin` en argument, l'algorithme va commencer à parcourir la collection à `end`, mais va aller vers la droite. Il ne rencontrera jamais `std::begin` et ne s'arrêtera que lorsque votre programme va crasher. En utilisant `std::rbegin` et `std::rend`, l'algorithme va aussi commencer vers la fin de la collection, mais va la parcourir vers la gauche. Il rencontrera `std::rend` et s'arrêtera donc correctement au début de la collection, après l'avoir parcouru.

La “const-correctness” (qui signifie “avoir un code qui respecte l'utilisation des constantes”) est un peu plus complexe à comprendre, puisqu'il peut y avoir deux types de constances avec les itérateurs : sur l'itérateur et sur l'élément qu'il représente.

Comme vous l'avez vu, vous pouvez utiliser le mot-clé `const` pour indiquer d'une variable ne sera pas modifiée. Avec un itérateur, cela

signifie que la variable représentera toujours le même élément dans une collection.

```
const auto first { std::begin(v); };
first = std::end(v);       // modification de l'itérateur : erreur
do_something(first);  // modification de l'élément : ok
```

Dans ce code, `first` représente toujours le premier élément, il n'est pas possible de changer cette variable pour affecter un autre itérateur. Par contre, l'élément de la collection peut être modifié si vous souhaitez.

Un itérateur constant représente un élément d'une collection, mais interdit de modifier cet élément. Il est possible de changer l'itérateur, mais pas la valeur de l'élément.

```
auto first { std::cbegin(v); };
first = std::cend(v);      // modification de l'itérateur : ok
do_something(first);  // modification de l'élément : erreur
```

Pour garantir un maximum la qualité du code, il faut utiliser les constantes dès que possible. Par exemple, vous avez vu dans le chapitre précédent le code suivant :

```
const std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
std::string s2       {
"......................................." };
std::copy(std::cbegin(s1), std::cend(s1), std::begin(s2));
```

Pour rappel, ce code copie les éléments de la première collection (`s1`) dans la seconde (`s2`). La première collection n'est pas modifiée, elle est donc constante. Dans le code, cette constance est exprimée par le mot-clé `const` dans la déclaration de la collection `s1` et dans l'utilisation des fonctions retournant des itérateurs constants `std::cbegin` et `std::cend`.

La seconde collection est modifiée, elle n'est donc pas déclarée comme constante et la fonction `std::begin` est utilisée.

Créer une sous-collection

L'utilisation des itérateurs ne se limite pas aux algorithmes, il est également possible de les utiliser dans d'autres contexte. Par exemple, beaucoup de collections (en particulier `std::vector` et `std::string`) peuvent être initialisées en utilisant une paire d'itérateurs. La nouvelle collection créés contiendra une copie des éléments contenus entre ces deux itérateurs.

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

int main() {
    std::string s1 { "azerty" };

    std::string s2(std::begin(s1), end(s1));
    std::cout << s2 << std::endl;

    std::string s3(std::begin(s1), std::begin(s1) + 3);
    std::cout << s3 << std::endl;
}
```

affiche :

```
azerty
aze
```

La variable `s2` correspond à une copie complète de la variable `s1`. Pour un code aussi simple que celui-là, cette syntaxe est équivalente à une initialisation directe :

```
std::string s2(s1);
```

Pour rappel, ce type d'initialisation peut poser des problèmes, par exemple lorsque les deux collections ne sont pas strictement identiques (par exemple `vector<float>` et `vector<double>`, voir Introduction aux algorithmes standards). L'utilisation de `std::copy` peut résoudre ce problème, mais il faut d'abord initialiser la seconde collection avec une

taille correcte, puis utiliser la copie.

```
std::string s2(s1.size());
std::copy(std::begin(s1), std::end(s1), std::begin(s2));
```

L'initialisation avec une paire d'itérateurs permet de simplifier le code et d'utiliser `const` plus facilement dans ce cas.

La déclaration de la variable `s3` utilise une opération arithmétique sur un itérateur : `std::begin(s1) + 3`. Vous allez voir plus en détail cette syntaxe par la suite, mais il suffit simplement de comprendre pour le moment que la syntaxe `std::begin(s1), std::begin(s1) + 3` permet de prendre les trois premiers éléments.

Intervalle semi-ouvert

Les fonctions `std::begin` et `std::end` ne sont pas les seules fonctions à retourner un itérateur. Plusieurs algorithmes retournent également un itérateur, comme `std::find` pour rechercher une valeur dans une collection ou `std::partition` pour séparer une collection en deux sous-collections.

Par exemple, avec `std::find` pour trouver un élément puis créer deux sous-collections :

main.cpp

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    const std::string s {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    const auto it = std::find(std::begin(s), std::end(s),
'8');
    std::cout << std::string(std::begin(s), it) << std::endl;
    std::cout << std::string(it, std::end(s)) << std::endl;
```

```
}
```

affiche :

```
f9c02b6c9da
8943feaea4966ba7417d65de2fe7e
```

Dans ce code, la fonction `std::find` recherche le caractère `8` et le trouve à la douzième position. L'itérateur correspondant à cette position est ensuite utilisé pour créer une première chaîne contenant les caractères avec cet élément, puis une seconde chaîne contenant les caractères suivant cet élément.

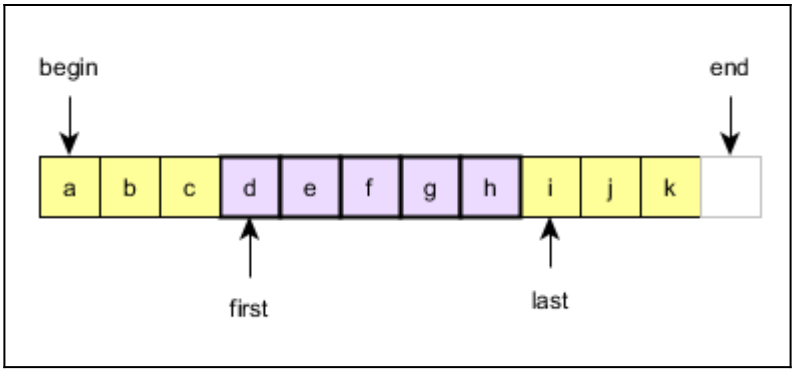
Vous pouvez remarquer un point important sur le fonctionnement des itérateurs : le caractère trouvé est dans la seconde sous-chaîne, pas dans la première. Une paire d'itérateurs fonctionne comme un intervalle semi-ouvert : l'élément correspondant au premier itérateur est inclut, l'élément correspondant au second itérateur est exclut.

Si vous recherchez les caractères 'd' et 'i' dans la chaîne `“abcdefghijk”` et que vous créez la chaîne correspondante à ces deux itérateurs, le résultat sera la chaîne `“defgh”`.

main.cpp

```cpp
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    const std::string s { "abcdefghijk" };
    const auto first = std::find(std::begin(s), std::end(s),
'd');
    const auto last = std::find(std::begin(s), std::end(s),
'i');
    std::cout << std::string(first, last) << std::endl;
}
```



Il se pose alors une question : que représente l'itérateur retourné par `std::end` ? Si `std::end` retourne un itérateur sur le dernier élément (par exemple 'k' dans le dernier exemple), alors la chaîne créée en utilisant les itérateurs retournés par `std::begin` et `std::end` correspondrait à la chaîne `“abcdefghij”` et non à la chaîne `“abcdefghijk”` (la chaîne sans le dernier caractère).

La fonction `std::end`, qui correspond à la fin de la collection, ne correspond pas en fait au dernier élément d'une collection, mais à un élément virtuel, qui n'existe pas. Cet élément virtuel peut être vu comme un élément supplémentaire après le dernier élément de la collection.

Cet élément n'existe pas réellement. Tenter d'y accéder ou de le manipuler comme s'il existait produirait un comportement indéterminé.

Et plus généralement, essayer d'utiliser un itérateur invalide (donc non compris entre `std::begin` inclut et `std::end` exclut) produit un comportement indéterminé. C'est au développeur de vérifier la validité des itérateurs qu'il manipule.

Pour les fonctions `std::rbegin` et `std::rend`, les itérateurs correspondent respectivement au dernier élément de la collection et un élément virtuel qui serait situé avant le premier élément.

Cet itérateur invalide retourné par `std::end` a une utilisation particulière avec les algorithmes de recherche, il est retourné lorsque la recherche

échoue. Par exemple, si vous essayez de rechercher le caractère “1” dans la chaîne “abcdefghijk”, le recherche échoue et retourne `std::end`.

main.cpp

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    const std::string s { "abcdefghijk" };
    const auto it = std::find(std::begin(s), std::end(s),
'1');
    std::cout << std::boolalpha << (it == std::end(s)) <<
std::endl;
}
```

affiche :

```
true
```

Avant d'utiliser un itérateur retourné par `std::begin` ou un algorithme, il est nécessaire de tester la validité avant utilisation.

```
assert(it != std::end(s));
```

Assertion
`assert` est une fonction qui vérifie qu'une condition est valide et arrête le programme si ce n'est pas le cas. Ce comportement est bien sûr un comportement qui ne doit pas arriver à l'utilisateur final de vos logiciels. C'est un outil qui sert à vérifier s'il y a des erreurs de programmation, qui doivent être corrigées avant de publier un logiciel.

Cette fonction est disponible dans le fichier d'en-tête `<cassert>`.

Une collection vide (par exemple `{}` ou une chaîne vide `""`) ne contient aucun élément. Dans ce cas, `std::begin` retourne `std::end` (vous pouvez donc tester si une collection est vide avec le code `std::begin == std::end`, mais il est plus logique d'utiliser la fonction membre

`empty()` si elle est disponible.

Intervalle (`range`)

Une paire d'itérateurs permet de définir un intervalle (*range*). Vous verrez parfois la notation `[first, last)` pour désigner un intervalle fermé à gauche et ouvert à droite. Cette notation mathématique permet d'écrire une collection, avec les crochets droits pour inclure les bornes et les parenthèses pour les exclure. Ainsi, l'ensemble [0, 1) correspond à l'ensemble des réels compris entre 0 inclus et 1 exclu, (0, 1) exclu les valeurs 0 et 1, (0, 1] exclu 0 et inclus 1, etc.

Notion d'indirection

Lorsque vous créez une variable, cela permet de créer un objet en mémoire, qui sera accessible directement avec cette variable. Chaque variable que vous créez correspond à un objet différent en mémoire, si vous créez une nouvelle variable à partir d'une variable existante, cela copie le premier objet. Si vous modifiez ensuite le second objet, le premier ne sera pas modifié.

Il existe plusieurs types d'indirection en C++ : les références, les pointeurs et les itérateurs. Les syntaxes présentent quelques différences, mais également des principes communs. Les itérateurs sont utilisés avec les collections, les références et pointeurs seront utilisés avec les fonctions (dans la partie “Créer ses algorithmes”) et l'allocation dynamique (dans la partie Programmation Orientée Objet).

Une indirection permet d'accéder indirectement à un autre objet, pour pouvoir lire la valeur ou la modifier. Accéder à l'élément correspondant à une indirection s'appelle “déréférencer une indirection”. Pour les itérateurs, l'accès se fait en utilisant l'opérateur `*` unaire préfixé.

- unaire signifie que cette opérateur ne prend qu'un argument, l'itérateur dans ce cas ;
- préfixé signifie que l'opérateur s'écrit avant son argument.

Plus concrètement, il faut donc écrire :

```
*it
```

Écris de cette façon, vous pouvez penser que cette syntaxe ne pose pas particulièrement de problème. Mais il ne faut pas oublier que l'opérateur `*` possède plusieurs signification en C++ : cela représente aussi la multiplication (et également les pointeurs, que vous verrez plus tard). Il faut donc être particulièrement vigilant pour éviter la confusion. N'hésitez pas à entourer cette syntaxe de parenthèses, dès que le code risque d'être ambiguë. (Voire vous pouvez mettre tout le temps les parenthèses, cela ne coûte rien).

main.cpp

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

int main() {
    const std::vector<int> v { 12, 23, 34 };
    const auto it = std::cbegin(v);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
12
```

Modifier un itérateur après déréférencement permet de modifier l'élément correspondant dans la collection.

main.cpp

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s { "abcdef" };
    const auto it = std::begin(s);
    std::cout << s << std::endl;
    (*it) = 'A';
    std::cout << s << std::endl;
```

```
}
```

affiche :

```
abcdef
Abcdef
```

N'oubliez pas la différence entre un itérateur constant et non constant. Essayer de modifier un itérateur constant produira une erreur de compilation :

main.cpp

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

int main() {
    std::string s { "abcdef" };
    const auto it = std::cbegin(s);  // cbegin au lieu de
begin
    std::cout << s << std::endl;
    (*it) = 'A';
    std::cout << s << std::endl;
}
```

affiche :

```
main.cpp:8:11: error: cannot assign to return value because
function 'operator*' returns a const value
    (*it) = 'A';
    ~~~~~ ^
```

L'erreur signifie que la valeur retournée par l'opérateur `*` est une valeur constante et ne peut être assignée.

Il est possible d'avoir plusieurs indirections sur un objet. Modifier l'une des indirections va modifier l'élément correspondant, ce qui implique que toutes les accès via les indirections seront modifiées.

main.cpp

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

```
int main() {
    std::string s { "abcdef" };
    const auto it1 = std::begin(s);
    const auto it2 = std::begin(s);
    std::cout << (*it2) << std::endl;
    (*it1) = 'A';
    std::cout << (*it2) << std::endl;
}
```

affiche :

```
a
A
```

Inférence de type après déréférencement

Si vous essayez de créer une variable avec `auto` pour contenir l'élément après déréférencement, vous perdez l'indirection :

main.cpp

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

int main() {
    std::string s { "abcdef" };
    const auto it = std::begin(s);
    auto c = (*it);
    std::cout << s << std::endl;
    c = 'A';
    std::cout << s << std::endl;
}
```

La ligne modifiant la variable `c` ne modifie pas la collection, ce n'est plus une indirection.

La raison est que `auto` ne conserve pas les modificateurs de type, en particulier les indirections. Ce code est donc équivalent au code suivant :

```
char c = (*it);
```

Ce qui permet de copier l'élément correspondant à l'itérateur dans une nouvelle variable, pas de créer une indirection sur cet élément.

Pour régler ce problème, vous pouvez utiliser des références (que vous verrez ensuite) ou utiliser `decltype(auto)` pour l'inférence de type. (Pour rappel, `decltype` et `decltype(auto)` conservent les modificateurs de type, au contraire de `auto`).

main.cpp
```cpp
#include <iostream>
#include <string>

int main() {
    std::string s { "abcdef" };
    const auto it = std::begin(s);
    decltype(auto) c = (*it);
    std::cout << s << std::endl;
    c = 'A';
    std::cout << s << std::endl;
}
```

affiche :

```
abcdef
Abcdef
```

Faites bien attention au type d'inférence de type lorsque vous utiliser `auto` et `decltype`.

Validité d'une indirection

Les indirections sont des outils très puissant en C++ (et d'autres langages bas niveau). Cependant, elles ont une contrainte d'utilisation qui peut être très problématique : elles ne sont valides que tant que l'élément correspondant est accessible. Si la collection est déplacée ou supprimée, une indirection va correspondre à un élément invalide et

tenter d'y accéder produira un comportement indéterminé.

Et il y a un problème encore plus important :

Il n'est pas possible de tester une indirection pour vérifier qu'elle est valide ou non !

Cette phrase peut paraître étrange, puisque vous avez vu qu'il était possible de tester un itérateur pour savoir s'il correspond à `std::end` ou non, pour savoir s'il est valide ou non. Pourquoi n'est-il pas possible de tester la validité d'une indirection dans ce cas ?

Pour commencer, un exemple de code montrant le problème :

main.cpp

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

int main() {
    std::string s { "abcdef" };
    const auto it = std::begin(s);
    std::cout << s << std::endl;
    s.clear();
    (*it) = 'z'; // à quel élément correspond (*it) ?
}
```

Il est facile de comprendre que puisque la chaîne `s` ne contient plus aucun élément, accéder au premier élément n'a aucun sens.

Notez bien que ce code ne produit pas d'erreur ! Le code s'exécute sans problème et ne semble pas présenter d'erreur. C'est le principe des comportements indéfinis (*Undefined Behavior*, UB) : cela ne produit pas forcement une erreur, cela peut sembler fonctionner et il est difficile de détecter (et donc de corriger) ce type d'erreur.

C'est de la responsabilité du développeur de prendre les précautions nécessaires pour ne pas avoir ce type d'erreur. Heureusement, il existe des bonnes pratiques et des outils d'analyse statique pour aider à limiter ces erreurs.

Ne sous-estimez pas la difficulté que représente ce type d'erreur. C'est une problématique qui n'a pas été résolue de façon sûre depuis la création du C++ (et du C). De nombreuses évolutions récentes du langage C++ vise à limiter ces problématiques.

Pour terminer, une précision sur pourquoi cela pose problème de corriger les itérateurs lorsqu'ils deviennent invalides. Pour cela, il faudrait que à chaque fois qu'une collection fait une opération qui invalide des itérateurs, il faut parcourir l'ensemble des itérateurs et les mettre à `std::end`. Faire cela est possible, mais peut être coûteux, sans que cela soit indispensable (si le développeur à correctement sécurisé son code).

La philosophie du C++ est de ne pas payer pour ce dont on n'a pas pas besoin. Sécuriser les collections pour ce problème ne peut donc pas être proposé par défaut. (Mais cela est possible, vous pourrez faire cela en exercice après la partie sur la programmation orientée objet).

Chapitre précédent	Sommaire principal	Chapitre suivant