Chapitre 5 :
Classes et conteneurs STL

Algorithmique et programmation 420-JWA-BT

Chapitre 5 : Classes et conteneurs STL

- `std::array`
- `std::vector`
- `std::deque`
- `std::list`
- `std::set`
- `std::map`

std::array

`std::array`

`#include <array>`

- La class **`std::array`** représente un tableau de taille fixe comme un tableau régulier C++. Cette classe a tous les avantages et désavantages des tableaux C++ sauf pour:
  - il est possible d'avoir un **`std::array`** avec une taille de 0;
  - la taille de **`std::array`** est toujours connue (avec la méthode **`size()`**);
  - il peut être passé en paramètre à une fonction sans dégénération;
  - il peut être copié avec l'opérateur d'assignation;
  - il peut utiliser toutes les fonctionnalités des conteneurs STL.

`std::array`

`#include <array>`

- La déclaration d'une variable **`std::array`** nécessite 2 informations, comme pour un tableau C++: le type d'éléments, et la nombre d'éléments:

```cpp
#include <array>

int main()
{
    std::array<int, 5> foo;
}
```

- L'initialisation est comme pour un tableau régulier:

```cpp
#include <array>

int main()
{
    std::array<int, 5> foo = { 1, 2, 3, 4, 5 };
}
```

std::array

#include <array>

- Comme pour les tableaux réguliers, si vous ne faites que déclarer, les éléments seront des valeurs inconnues. Et si le nombre d'éléments de la liste est inférieur à la taille, le reste sera 0.

- Si vous voulez un tableau à plusieurs dimensions, vous devez littéralement faire un tableau de tableaux:

```cpp
#include <array>

int main()
{
    // foo est un tableau de 3 tableau de 5 int
    std::array<std::array<int, 5>, 3> foo;
}
```

std::array

#include <array>

- Vous pouvez les utiliser comme un tableau régulier pour accéder aux éléments et boucler:

```cpp
#include <array>

int main()
{
    std::array<int, 5> foo;
    foo[0] = 1;
    foo[1] = 2;
    foo[2] = 3;
    foo[3] = 4;
    foo[4] = 5;

    for (int i = 0; i < foo.size(); ++i)
    {
        std::cout << foo[i] << std::endl;
    }

    for (int i: foo)
    {
        std::cout << i << std::endl;
    }
}
```

std::array

#include <array>

- Faites attention à le passer en paramètre à une fonction! Contrairement aux tableaux réguliers, il sera passé par copie de son contenu et non de son adresse!

```
void MaFonction(std::array<int, 1000000> foo)
{
    // ...
}
```

- Il est donc très important de les passer par référence:

```
void MaFonction(std::array<int, 1000000>& foo)
{
    // ...
}
```

`std::array`

`#include <array>`

- Il est aussi possible d'accéder aux éléments avec la méthode "**`at(size_type pos)`**". Ceci était très courant dans du code pré-C++11 puisque l'opérateur "`[]`" ne pouvait pas s'appliquer sur des références **`const`**.

```
void MaFonction(const std::array<int, 1000000>& foo)
{
    std::cout << foo.at(0) << std::endl;
}
```

- À partir de C++11, ceci n'est plus le cas, et la différence entre "**`at`**" et "`[]`" est que "**`at`**" vérifie que la position demandée est valide, contrairement à "`[]`" qui permet d'accéder n'importe où en mémoire.

`std::array`

`#include <array>`

- Si nous avons besoin d'accéder directement au tableau caché par **`std::array`**, il est possible de le faire à l'aide de la méthode **`data()`**.

```cpp
void MaFonction(int* foo)
{
    // ...
}


int main()
{
    std::array<int, 5> foo;

    MaFonction(foo); // erreur de compilation!
}
```

```cpp
void MaFonction(int* foo)
{
    // ...
}


int main()
{
    std::array<int, 5> foo;

    MaFonction(foo.data()); // OK!
}
```

std::vector

`std::vector`

`#include <vector>`

- La classe **`std::vector`** est un tableau à taille dynamique. Puisqu'elle peut changer, la taille ne fait pas partie du type de la variable, seulement le type d' éléments:

```cpp
#include <vector>

int main()
{
    std::vector<int> foo;
}
```

- Sans initialisation, l'objet **`std::vector`** a une constructeur qui l'initie avec une taille de 0. Il n'est donc pas nécessaire d'avoir une initialisation. Par contre, vous pouvez lui donner une taille de départ en le spécifiant au constructeur:

```cpp
#include <vector>

int main()
{
    std::vector<int> foo(5);
}
```

`std::vector`

`#include <vector>`

- Vous pouvez également lui donner une liste de valeurs comme pour un tableau:

```cpp
#include <vector>

int main()
{
    std::vector<int> foo = { 1, 2, 3, 4, 5 };
}
```

- Après initialisation, un **`std::vector`** se comporte de façon pratiquement identique à un **`std::array`**. La plus grande différence est qu'il est possible de modifier sa taille. **`std::vector`** permet ceci en allouant un nouveau tableau et y copiant tous les éléments de l'ancien. Normalement, il prévoit le coup et s'alloue un tableau plus grand que nécessaire. Vous pouvez obtenir la taille cachée de ce tableau avec la méthode **`capacity()`**.

`std::vector`

`#include <vector>`

- **`clear()`**: Vide le **`std::vector`** et retourne sa taille à 0.
- **`insert(pos, value)`**: Insert la valeur “**`value`**” à la position “**`pos`**”, augmentant la taille de 1. Plus la position est près du début, plus l’opération est coûteuse.
- **`erase(pos)`**: Efface la valeur à position “**`pos`**”, réduisant la taille de 1. Plus la position est près du début, plus l’opération est coûteuse.
- **`push_back(value)`**: Ajoute un nouvel élément à la fin. Ceci peut être rapide si la capacité est suffisante, sinon ceci pourrait demander une nouvelle allocation de tableau.
- **`pop_back()`**: Enlève le dernier élément. Normalement ceci est très rapide.
- **`resize(count)`**: Change la taille, en réduisant ou augmentant.

`std::vector`

`#include <vector>`

- On peut également éviter des réallocations inutiles en réservant d'avance une capacité avec la méthode **`reserve(new_cap)`**.

- Puisque **`std::vector`** utilise un tableau C++ en arrière-plan, la performance de toutes les opérations n'affectant pas la taille est identique à l'utilisation d'un tableau. Il est commun de voir **`std::vector`** utilisé à la place de **`std::array`** ou d'un tableau C++ même si la taille ne change pas. D'ailleurs, il est même recommandé comme conteneur si dans le doute.

- Le coût d'une réallocation est souvent moindre que l'on peut l'imaginer. L'implémentation essaie d'éviter la copie le plus possible en utilisant **`realloc`** si possible. Cette fonction garde le même tableau si la mémoire désirée était immédiatement disponible après le tableau en mémoire. Par contre, on ne peut pas prendre ceci pour acquis, et il est toujours préférable de réserver l'espace et d'ajouter les éléments en groupes.

`std::vector`

`#include <vector>`

- Faites attention aux références aux éléments d'un **`std::vector`**! Puisqu'il est possible que le tableau soit réalloué après une modification, si vous aviez une référence à un élément existant, il est possible que cette référence ne soit plus valide!

```cpp
#include <vector>

int main()
{
    std::vector<int> foo = { 1, 2, 3, 4, 5 };

    // référence au premier élément
    int& element = foo[0];

    // ajout d'un élément, possible que la référence soit maintenant invalide!
    foo.push_back(6);

    // pourrait faire planter le programme!
    std::cout << element << std::endl;
}
```

std::deque

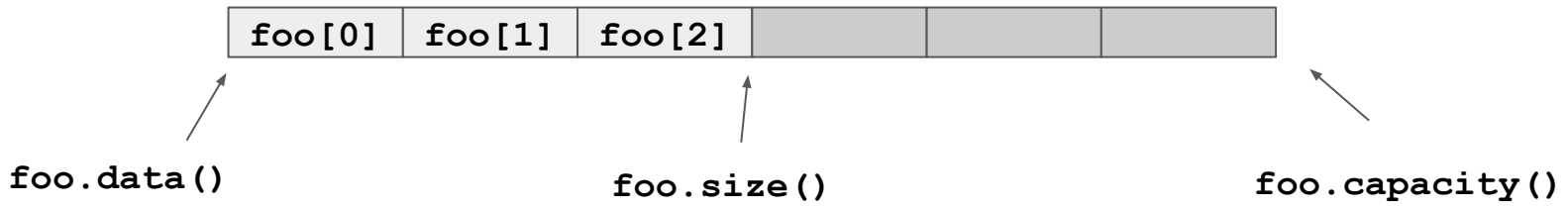
`std::deque`

`#include <deque>`

- Puisque `std::vector` est considéré une classe par excellence, tant que les modifications sont faites par la fin, il y a une demande pour une classe permettant de modifier aussi efficacement par le début.

- `std::deque` veut dire "***D**ouble-**E**nded **QUE**ue*", et est pratiquement aussi efficace mais permet d'insérer au début en faisant plusieurs tableaux en mémoire.

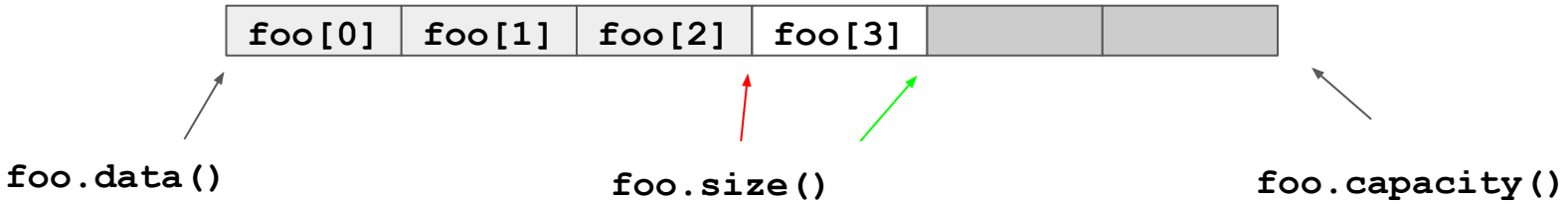
- En comparaison, un `std::vector` a généralement un tableau fixe avec une capacité supérieure à son contenu:



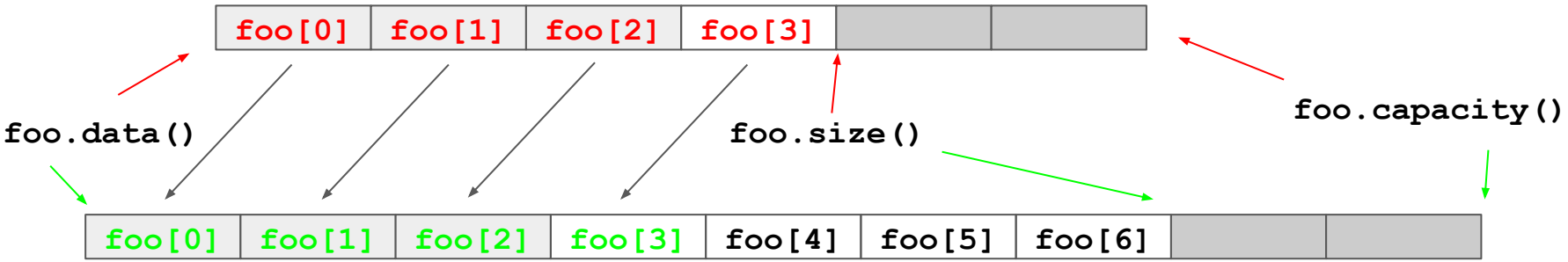
`std::deque`

```
#include <deque>
```

- Lorsque vous insérez des éléments à la fin d'un `std::vector`, s'il a une capacité suffisante, ceci n'a pas de coût additionnel:



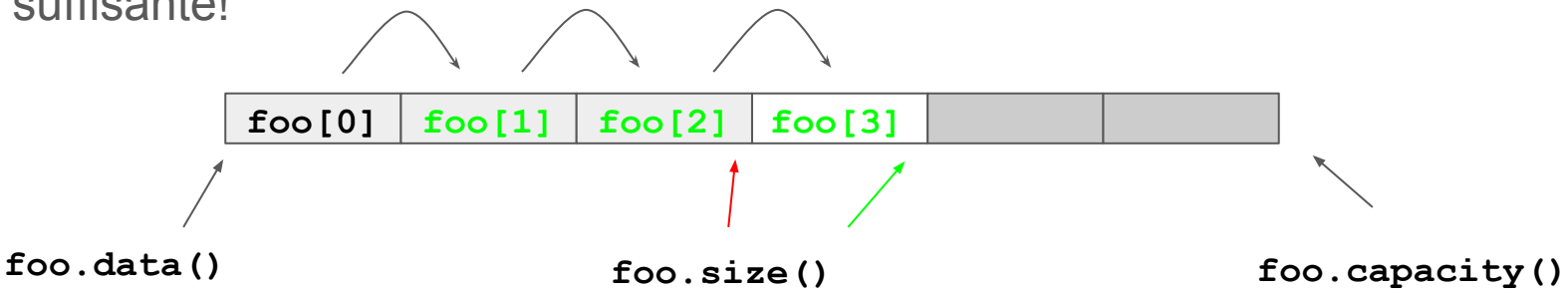
- Mais si la capacité est insuffisante, il crée un nouveau tableau avec une plus grande capacité et copie le tout:



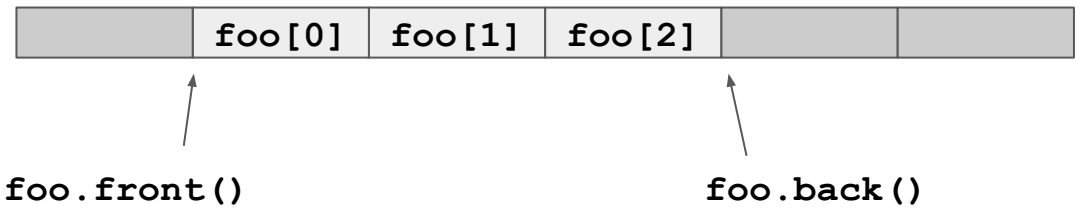
`std::deque`

`#include <deque>`

- Lorsque l'on insère au début, cela est coûteux même si la capacité est suffisante!



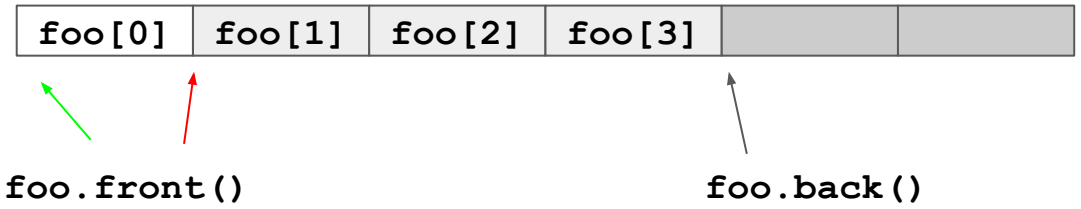
- La classe `std::deque` ne fait jamais de déplacement et/ou copie d'élément lorsque modifiée par une des deux extrémités. Elle utilise la même technique que `std::vector` pour déplacer la fin lorsque la capacité est suffisante, mais le fait aussi pour le début.



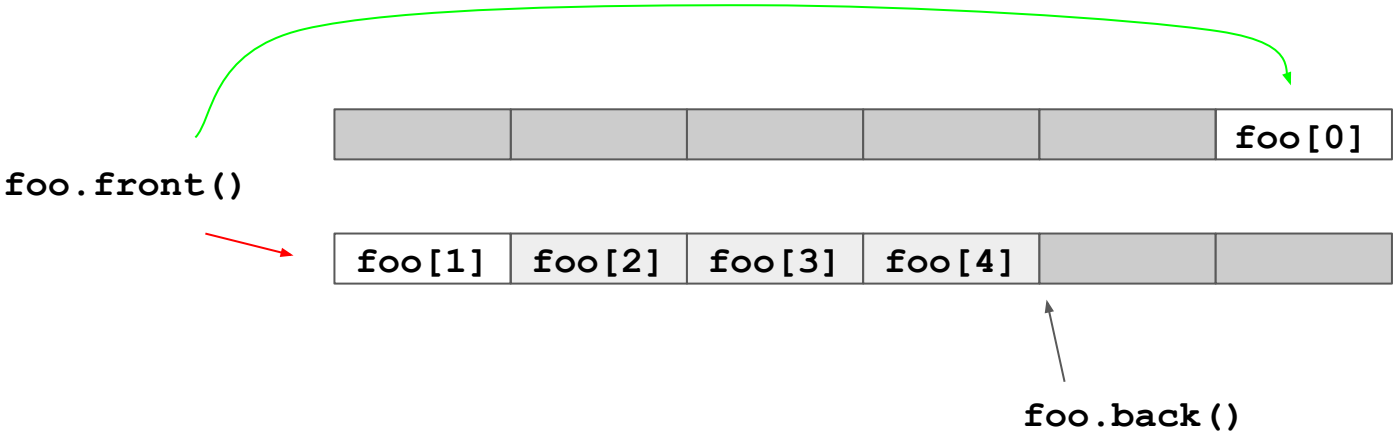
std::deque

```
#include <deque>
```

- L'insertion au début est donc sans coût lorsque la capacité est suffisante:



- Lorsque la capacité est insuffisante, le comportement est différent. `std::deque` créera aussi un nouveau tableau, mais il ne copiera pas les éléments existants.



`std::deque`

`#include <deque>`

- Ce comportement veut dire que la mémoire n'est pas strictement contiguë. Nous avons une liste de tableaux, donc chaque accès doit maintenant passer par deux adresses pour accéder son contenu. **`std::deque`** n'est donc pas un remplacement sans coût pour un tableau, et ne devrait être utilisé que lorsque le coût d'effectuer des copies ou de modifier par le début sont trop considérables.

- Lorsque les éléments sont insérés au milieu, il n'y a aucun avantage d'utiliser **`std::deque`**, il doit copier tous les éléments déplacés comme un **`std::vector`**.

- Puisque les éléments d'un **`std::deque`** ne sont pas tous dans un même tableau, on ne peut pas obtenir celui-ci avec une méthode "**`data()`**".

`std::deque`

`#include <deque>`

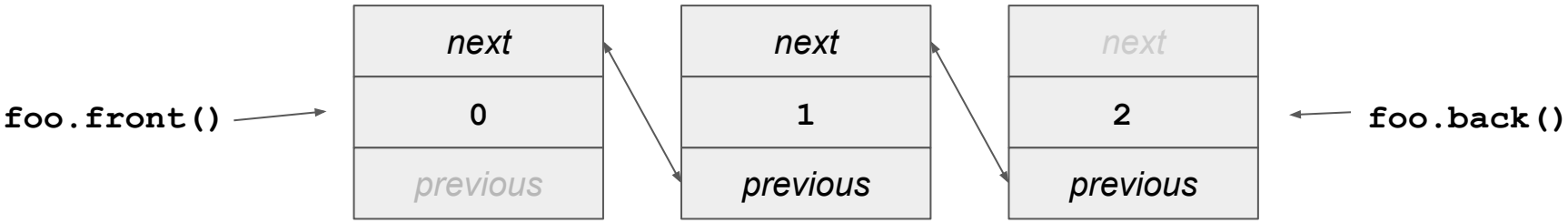
- En plus des méthodes de modification qu'offre **`std::vector`**, **`std::deque`** offre également:
  - **`push_front(value)`** : insère un élément au début, augmentant la taille de 1.
  - **`pop_front()`** : enlève l'élément au début, réduisant la taille de 1.
- La déclaration, l'initialisation, et le reste des méthodes sont pratiquement identiques à **`std::vector`**.

std::list

`std::list`

`#include <list>`

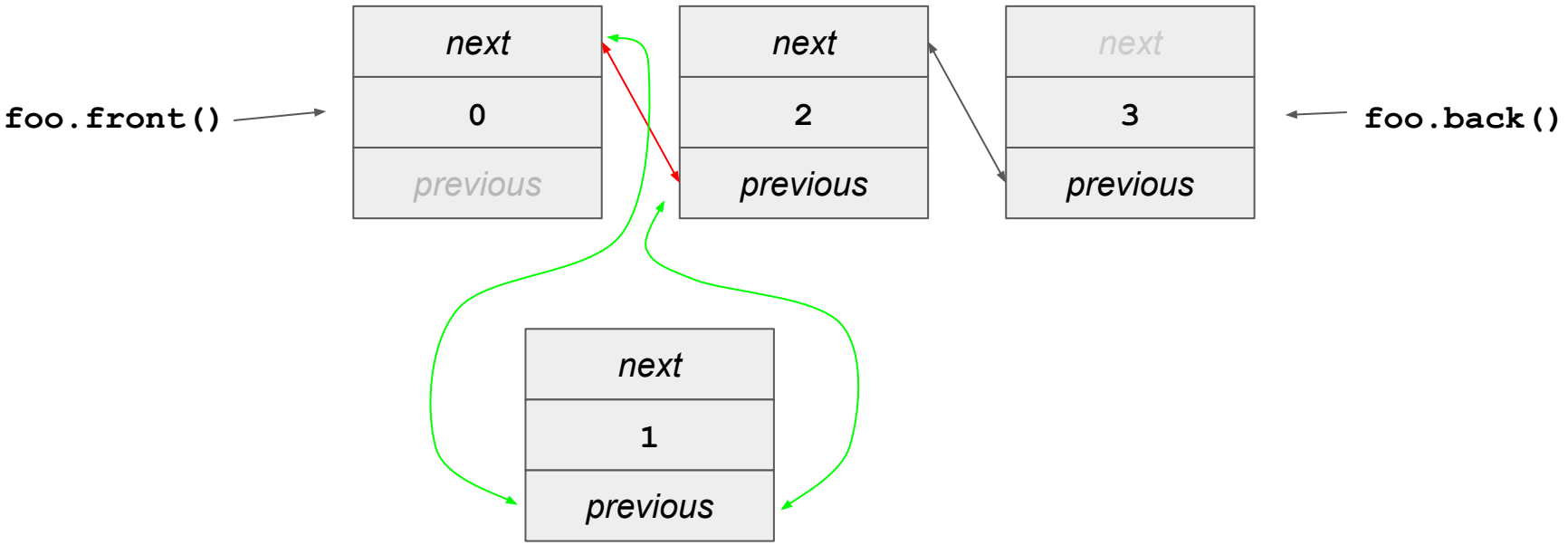
- **`std::list`** est l'implémentation d'une liste chaînée en C++. Contrairement à un tableau, une liste chaînée n'offre pas un accès direct à ses éléments puisqu'ils ne sont pas en mémoire contiguë. Pour accéder à un élément, on doit naviguer la liste, chaînon par chaînon, jusqu'à ce qu'on ait atteint notre objectif. Il n'est donc pas possible d'utiliser "**`at`**" ou "**`[]`**" pour accéder des éléments.



`std::list`

```
#include <list>
```

- Malgré que ceci peut sembler très coûteux, `std::list` permet d'insérer à n'importe quelle position sans avoir besoin de copier des éléments! On ne fait que créer un nouveau chaînon et ajuster les références:



`std::list`

`#include <list>`

- Les méthodes de modification de **`std::list`** sont les mêmes que pour **`std::deque`**.

- **`std::list`** devrait seulement être utilisée si les modifications à la liste sont souvent au milieu et que le coût de copie de ses éléments est élevé. Dans la majorité des cas, le ralentissement d'accès aux éléments autre que le premier et le dernier est beaucoup plus considérable les avantages par rapport à **`std::deque`** ou **`std::vector`**.

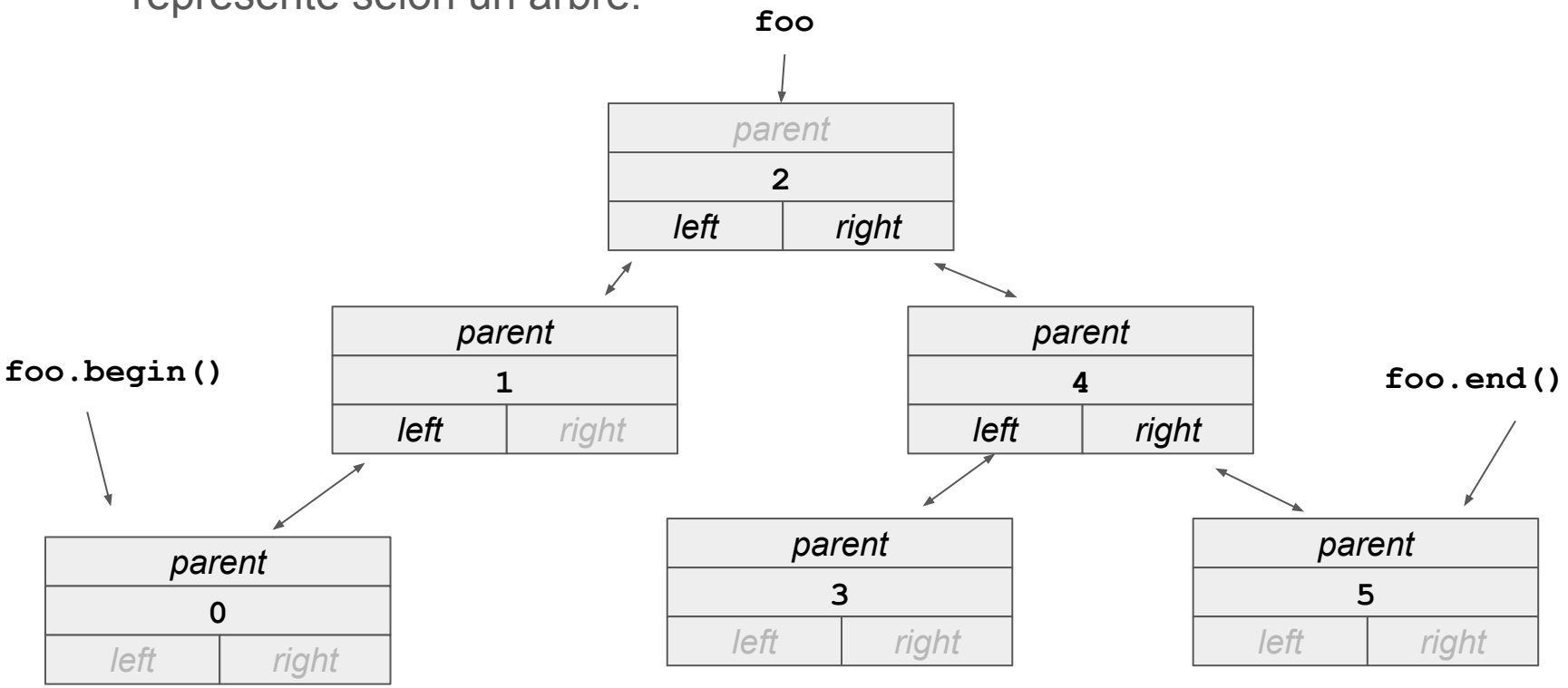
- Si nous n'avons besoin de naviguer que dans une direction, il est possible d'utiliser **`std::forward_list`** pour n'avoir besoin qu'une seule référence par chaînon.

std::set

`std::set`

`#include <set>`

- `std::set` représente un ensemble d'éléments unique trié en ordre. Il a l'avantage que tous les éléments s'insèrent automatiquement à leur position selon l'ordre de tri, et la recherche d'élément est rapide. Il est généralement représenté selon un arbre:



`std::set`

```
#include <set>
```

- Le fonctionnement complet d'un arbre binaire est au delà des objectifs du cours, mais vous pouvez en lire davantage du Wikipédia: https://fr.wikipedia.org/wiki/Arbre_bicolore

- Contrairement aux autres conteneurs, on ne peut pas insérer un élément à une position précise. La position est toujours déterminé par son ordre de tri. Tout comme **`std::list`**, nous n'avons pas accès aux éléments par leur position avec "**`at`**" ou "**`[]`**". Le but d'un **`std::set`** est de représenter un ensemble et non pas d'avoir une liste séquentielle d'éléments. On ne parle généralement plus *d'accéder* à un élément, mais plutôt *de trouver* un élément.

`std::set`

`#include <set>`

- **`clear()`** : vide le contenu, comme pour les autres conteneurs.
- **`insert(value)`** : insère un élément à la position selon son ordre de tri.
- **`erase(value)`** : efface un élément à la position selon son ordre de tri.
- **`count(value)`** : retourne le nombre d'éléments avec la valeur "**`value`**" (0 ou 1).

`std::set`

`#include <set>`

- Si on veut pouvoir avoir plus d'une fois le même élément, on peut utiliser la classe **`std::multiset`**. Dans ce cas, la méthode **`count(value)`** pourrait retourner un nombre plus grand que 1. Faites attention à l'utilisation de **`erase(value)`**, car elle effacera TOUS les éléments avec cette valeur, et non pas uniquement un élément. Pour effacer un seul élément, on doit utiliser une combinaison des méthodes **`find`** et **`erase`**.

std::map

`std::map`

`#include <map>`

- **`std::map`** est fonctionne de la même façon que **`std::set`**, mais chaque élément se fait également associer une valeur autre que sa clé de tri. On doit donc déclarer une variable **`std::map`** en spécifiant deux types: la clé et la valeur associée.

```cpp
#include <map>

int main()
{
    std::map<std::string, int> eleves;
}
```

`std::map`

`#include <map>`

- Puisqu'il y a deux types, lorsque l'on veut initialiser la **`std::map`**, on doit le faire à l'aide de paires:

```cpp
#include <map>

int main()
{
    std::map<std::string, int> eleves =
    {
        { "Matthew", 1810022 },
        { "Charles", 1810051 },
        { "Guillaume", 1810080 },
        { "Mathieu", 1810042 },
        { "Jimmy", 1810027 },
        { "Ivann", 1810043 },
        { "Alexandre", 1810078 },
        { "Raphael", 1810028 },
        { "Patrick", 1810057 }
    };
}
```

`std::map`

`#include <map>`

- **`std::map`** ré-introduit l'opérateur "**`[]`**" pour ajouter et extraire des éléments selon leur clé:

```
// ajout de Donovan
eleves["Donovan"] = 1810009;

// affichage du numéro de Jimmy
std::cout << eleves["Jimmy"] << std::endl;
```

- La majorité des méthodes fonctionnent avec uniquement la clé de tri (**`erase`**, **`count`**, …). Par contre, **`insert`** prend maintenant une paire au lieu d'une seule valeur:

```
eleves.insert({ "David", 1810031 });
```

`std::map`

`#include <map>`

- Itérer sur **`std::map`** se fait également avec des paires. Celle-ci doivent avec les mêmes types que la **`std::map`**.

```
for (const std::pair<std::string, int>& eleve : eleves)
{
    std::cout << "Le numéro de " << eleve.first << " est " << eleve.second << "." << std::endl;
}
```

  À noter que les paires de cette boucle doivent être const puisqu'il n'est pas permis de modifier les paires internes de **`std::map`**. Souvent, on utilise le type "**`auto`**" pour ne pas avoir à déterminer quels étaient les types de la **`std::map`**.

```
for (auto& eleve : eleves)
{
    std::cout << "Le numéro de " << eleve.first << " est " << eleve.second << "." << std::endl;
}
```

std::map

#include <map>

- Lorsque l'on itère de cette façon, l'ordre sera toujours l'ordre de tri des clés de notre ensemble. Par exemple, avec le code précédant, les étudiants sortiront en ordre alphabétique:

```
Le numéro de Alexandre est 1810078.
Le numéro de Charles est 1810051.
Le numéro de David est 1810031.
Le numéro de Donovan est 1810009.
Le numéro de Guillaume est 1810080.
Le numéro de Ivann est 1810043.
Le numéro de Jimmy est 1810027.
Le numéro de Mathieu est 1810042.
Le numéro de Matthew est 1810022.
Le numéro de Patrick est 1810057.
Le numéro de Raphael est 1810028.
```

`std::map`

```
#include <map>
```

- Si les clés ne sont pas unique, on peut utiliser **`std::multimap`** qui permet d'avoir plus d'un élément avec la même clé. Ceci pourrait être pratique si on gardait plutôt les notes d'un élève au lieu de son numéro!

- Lorsque les deux éléments sont uniques (par exemple, ici le nom et le numéro), on doit déterminer à la création ce qui servira pour trouver un élément le plus souvent. Dans les exemples précédents, trouver un élève par nom est rapide, mais trouver un élève par numéro pourrait devoir rechercher à travers la totalité de l'ensemble.