Les catégories de collections

La bibliothèque standard propose de nombreux conteneurs de données. Pour le moment, vous avez vu principalement `std::vector` et `std::string`. Ce chapitre présente les autres collections de la bibliothèque standard, en particulier les listes doublement chaînées `std::list` et les conteneurs associatifs `std::map` et `std::set`.

Les collections se distinguent par leur gestion interne des données et par les fonctionnalités qu'elles proposent. Cependant, il est habituel que plusieurs collections proposent les mêmes fonctionnalités.

Par exemple, plusieurs collections permettent d'ajouter un élément à la fin d'une collection avec la fonction `push_back` (“pousser à la fin”), comme `std::vector`, `std::list`, ou `std::deque`. Ces différentes collections ne gèrent pas les données de la même façon en interne, ce qui a un impact sur les performances. La complexité algorithmique de chaque fonction est indiquée dans la documentation.

(Si vous ne connaissez pas ce qu'est la complexité algorithmique, ce concept sera détaillé dans les chapitres sur la création d'algorithmes).

Note : la manipulation plus poussée des collections avec `begin` et `end` passe par l'utilisation des itérateurs. Ce concept est suffisamment important pour faire l'objet d'un chapitre dédié. Ce chapitre se focalise uniquement sur les fonctionnalités spécifiques des conteneurs.

Collection et conteneur
En toute rigueur, les concepts de collection et conteneur sont différents. Le premier correspond à un objet qui propose les concepts de “premier élément”, “dernier élément” et “élément suivant”, le second correspond à un objet qui peut contenir d'autres objets.

En pratique, les conteneurs de la bibliothèque standard du C++ sont

aussi des collections, vous verrez souvent dans ce cours les termes “collection” et “conteneur” utilisés indifféremment. (Lorsque vous créerez vos propres conteneurs, il est recommandé de les implémenter également sous forme de collections, pour être compatibles avec les algorithmes de la bibliothèque standard).

Mais n'oubliez pas que ce sont des concepts distincts (par exemple, les vues *views* proposent les fonctionnalités des collections, mais ne contiennent les données qu'elles manipulent. Et les graphes peuvent être des conteneurs de données, mais ne proposent généralement pas les concepts de “premier élément”, “dernier élément” et “élément suivant”).

Conteneurs séquentiels et associatifs

La première grande classification des collections est la séparation entre conteneurs séquentiels et associatifs. Un conteneur associatif associe une clé à des informations (un objet). A partir de cette clé, il est alors facile de retrouver les informations associées. C'est une notion que vous pouvez retrouver dans la vie de tous les jours : par exemple, pour trouver une définition (l'information) d'un mot (la clé) dans un dictionnaire (la collection de données), ou lorsque vous allez à la banque consulter vos dépenses du mois (les informations) en utilisant votre numéro de compte (la clé).

Un conteneur associatif va gérer en interne l'organisation des données pour optimiser les accès à partir de la clé. Il existe plusieurs types de conteneurs associatifs, qui se distinguent selon leur façon d’organiser les données et donc les facilités d'accès.

Au contraire, un conteneur séquentiel ne présente pas d'accès privilégié aux données selon une clé et n'organise pas les données en interne. Il est possible d'accéder aux données uniquement de façon séquentielle, c'est-à-dire en utilisant les concepts communs à toutes les collections : “début d'une collection”, “fin d'une collection” et “élément suivant”. Certains conteneurs séquentiels proposent des fonctionnalités supplémentaires, comme par exemple “accéder directement à l'élément n” ou “élément précédent”.

Notez bien que les conteneurs associatifs sont aussi des collections et peuvent donc être manipulés comme des conteneurs séquentiels. Cependant, n'oubliez pas qu'ils gèrent automatiquement en interne les données, cela n'a pas de sens par exemple de trier les éléments avec `std::sort`.

Pour bien comprendre ce principe de “gestion automatiquement interne” des données par un conteneur associatif, le code suivant utilise une conteneur non-associatif (`std::vector`) et un conteneur associatif (`std::map`) et affiche les éléments séquentiellement (avec une boucle `for` que vous verrez dans la suite de ce cours).

main.cpp

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

int main() {
    using data_t = std::pair<int, std::string>;
    const std::vector<data_t> v {
        { 2, "hello" },
        { 3, "every" },
        { 1, "body" }
    };
    for (auto p: v)
        std::cout << p.first << ' ' << p.second << std::endl;

    std::cout << std::endl;

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

affiche :

```
2 hello
3 every
1 body

1 body
2 hello
3 every
```

Dans le conteneur non-associatif (`std::vector`), les éléments sont affichés dans le même ordre qu'ils ont été entrés dans la collection. Dans le conteneur associatif (`std::map`), les éléments ont été trié en fonction de la clé (la valeur entière).

Vous pouvez voir dans le code précédent qu'il est possible de manipuler les mêmes informations, quelque soit le type de conteneur. Il est tout à fait possible de trier un tableau `std::vector` en fonction de la valeur entière avec `std::sort` et de faire une recherche sur cette valeur avec `std::find`. La différence est que dans le cas des conteneurs associatifs, l'une des informations a un rôle particulier pour le stockage et l'accès.

Choisir la structure de données la plus adaptée à une problématique fait partie des bases de la programmation.

Les conteneurs séquentiels

Les tableaux

Le premier type de conteneur séquentiel (et probablement le plus utilisé) est le tableau. Vous connaissez déjà `std::vector` (tableau de taille dynamique) et `std::array` (tableau de taille fixée à la compilation), il existe également `std::valarray` (un tableau spécialisé pour les calculs arithmétiques, ce type est peu utilisé) et `std::dynarray` (un tableau de taille fixé à l'exécution, qui sera ajouté dans le C++17, mais qui est déjà disponible dans certains compilateurs dans `std::experimental::dynarray`).

Anciennes syntaxes
Il existe également deux types de tableaux hérités du C, de taille fixée à la compilation et de taille dynamique. La syntaxe est la suivante :

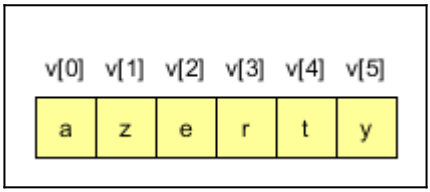
```
// taille fixe
int static_array[10];

// taille dynamique
int dynamic_array[] = new[10] int;
// utilisation...
delete[] dynamic_array;
```

L'utilisation correcte de ce type de tableau (et plus généralement la gestion manuelle de la mémoire avec `new` et `delete`) est relativement complexe et doit être évitée en C++ moderne. Les cas d'utilisation acceptables peuvent être la compatibilité avec un code C++ ancien ou avec le C, ou pour des implémentations de conteneurs, à partir du moment où ce type de code est correctement isolé du reste du programme (encapsulation, cela sera détaillé dans la partie sur la programmation orientée objet).

La particularité des tableaux est d'avoir ses éléments contiguës en mémoire, dans un bloc mémoire réservé pour ce tableau. Cela permet d'avoir un accès efficace direct à n'importe quel élément, selon son indice dans le tableau, avec l'opérateur d'indexation `[]`. (Il est également possible d'utiliser la fonction `at`, mais son utilisation est déconseillée). Les indices commencent à partir de zéro jusque `size()-1`.

```
const std::vector<char> v { 'a', 'z', 'e', 'r', 't', 'y' };
std:cout << v[0] << std::endl;
```



L'indice est une valeur entière positive, de type `vector<T>::size_type`, mais qui est en général similaire au type `size_t`.

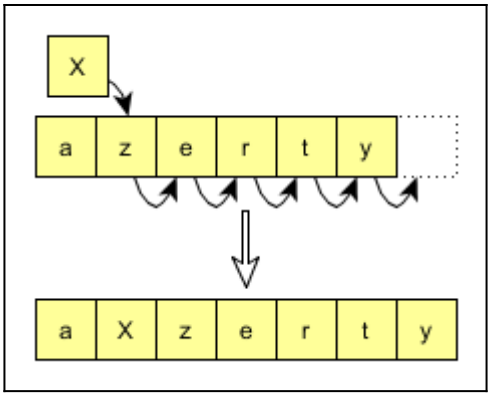
L'accès en dehors des limites d'un tableau produit un comportement indéterminé. Il est de la responsabilité du développeur de vérifier les accès à un tableau, au minimum avec `assert`. Un indice est obligatoirement positif, mais il n'est pas nécessaire de vérifier cela si vous utilisez `size_t` (cela n'aurait pas de sens d'utiliser un type entier signé, comme `int`). Il faut également vérifier que la valeur ne soit pas plus grande que la taille du tableau.

```
const size_t i { ... };
assert(i < v.size());
v[i]; // ok
```

(Le code `i < v.size()` est strictement équivalent à `i ⇐ v.size()-1` pour les entiers, mais n'utilise qu'une seule opération de comparaison, alors que la seconde syntaxe utilise une comparaison et une soustraction, ce qui est moins performant).

Gestion de la mémoire des tableaux

Avoir les éléments contiguës en mémoire permet aux tableaux d'être très efficace (c'est plus optimisé pour les caches mémoires des ordinateurs modernes, mais cela sort du cadre de ce cours), mais impose également des contraintes. Par exemple, si vous souhaitez ajouter un élément au milieu du tableau du code précédent, il ne sera pas possible de placer cet élément directement dans le tableau (puisqu'il n'y a pas d'emplacement mémoire libre entre deux éléments). Il est alors nécessaire de déplacer tous les éléments qui se trouvent à droite de l'élément inséré, pour libérer un emplacement.



Il peut alors y avoir un nouveau problème. Cet insertion ne peut avoir lieu que s'il est possible d'utiliser l'emplacement mémoire situé directement à droite du dernier élément. Si ce n'est pas possible (en général parce qu'il y a déjà quelque chose à cet emplacement mémoire), il faudra alors créer un nouveau tableau en mémoire (avec la nouvelle taille), copier tous les éléments depuis l'ancien tableau vers le nouveau, puis insérer le nouvel élément. (Tout cela est réalisé automatiquement par `std:vector` en interne, cela ne nécessite pas d'écrire un code spécifique de votre part).

Toutes ces copies et allocations de mémoire sont très coûteuses en termes de performances. De plus, cela invalide les indirections (cela sera détaillé dans un prochain chapitre). Il est donc important de limiter ce phénomène.

Pour cela, `std::vector` alloue en mémoire plus d'éléments que nécessaire. Lorsque vous insérer un nouvel élément, cette reserve sera utilisée en propriété et le tableau ne sera copié que si vous dépasser la capacité de cette reserve.

Cette réserve est en partie gérée automatiquement par `std::vector`, mais il est également possible de la gérer manuellement. Pour cela, vous pouvez utiliser les fonctions suivantes. Pour modifier la réserve :

- `reserve(n)` permet d'augmenter la taille de la réserve en mémoire. Cette fonction prend en paramètre le nombre d'éléments total que vous souhaitez. La réserve est augmentée

uniquement si vous demandez plus que la réserve actuelle.
- `shrink_to_fit()` permet de réduire au maximum la taille de la réserve, de façon à ce que la taille totale correspond au nombre d'éléments réellement utilisés. (Autrement dit, la réserve sera mise à zéro).

Pour connaitre l'utilisation de la mémoire :

- `size()` permet de connaître le nombre total d'élément actuellement utilisés dans le tableau.
- `capacity()` permet de connaître le nombre totale d'élément dans le tableau (utilisés et réserve).
- `max_size()` permet de connaître le nombre maximal d'élément qu'il sera possible d'avoir dans un tableau.
- `empty()` permet de savoir si un tableau ne contient pas d'élément utilisé (autrement dit, si `size()` vaut zéro).

Pour avoir un code le plus performant possible, il faudra donc faire attention de :

- créer les tableaux directement avec des valeurs (de préférence comme constant) ;
- réserver assez d'éléments pour éviter les ré-allocations de mémoire ;
- éviter les copies de tableaux.

```
// tableau constant
const std::array<int, 4> a { 1, 2, 3, 4 };

// tableau dynamique
std::vector<int> v { 1, 2, 3, 4 };
std::vector<int> v(100); // I

// réserve
std::vector<int> v;
v.reserve(100);
```

Notez bien la différence entre `std::vector<int> v(100)` et

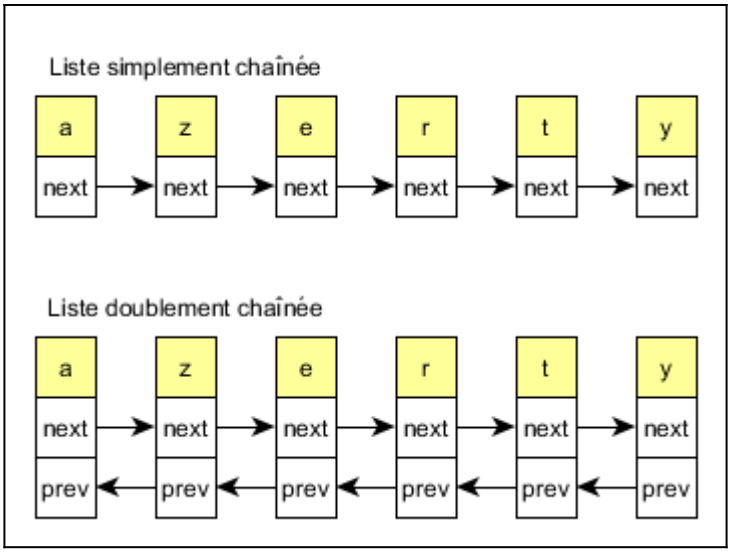
`v.reserve(100)`. La première syntaxe crée un tableau qui contient 100 éléments (la taille de la réserve est choisie automatiquement), la seconde crée un tableau vide (0 élément), mais qui contient 100 éléments dans la réserve.

Les listes

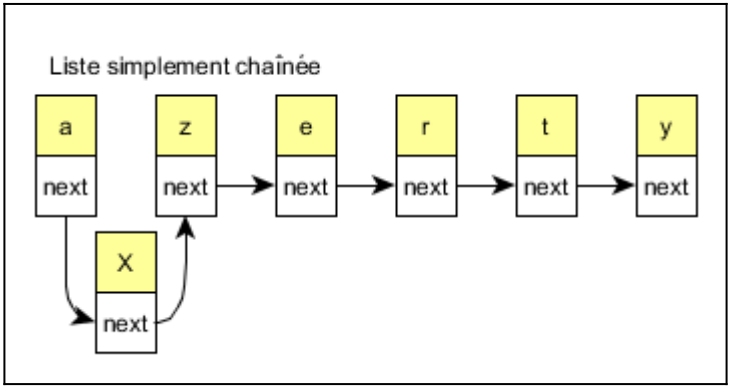
Les listes chaînées sont également des structures de données classiques. Chaque élément d'une liste contient un lien vers l'élément suivant (liste simplement chaînée) et l'élément précédent (liste doublement chaînée). A partir d'un élément, il suffit donc d'appeler l'élément suivant récursivement pour parcourir la collection dans le sens direct et d'appeler l'élément précédent pour parcourir dans le sens inverse.

```
// liste simplement chaînée
const std::forward_list<char> l { 'a', 'z', 'e', 'r', 't',
'y' };

// liste doublement chaînée
const std::list<char> l { 'a', 'z', 'e', 'r', 't', 'y' };
```



Dans une liste, l'insertion ou la suppression d'un élément nécessite simplement de modifier les indirections vers les éléments suivants et précédents. Il n'y a donc pas besoin de copier des éléments pour les déplacer ou de ré-allouer la mémoire. Les indirections sur les éléments d'une liste ne sont pas invalidés dans ces opérations.



En termes de mémoire, les listes consomment un peu plus que les tableaux, du fait de la présence des indirections pour chaque élément. Si une collection contient N éléments, alors une liste simplement chaînée aura un surcoût mémoire de N x sizeof(indirection), et une liste

doublement chaînée aura un surcoût de 2N x sizeof(indirection). A cela, il faut également ajouter une perte de performances, à cause des caches mémoires (la mémoire n'étant pas contiguë, l'utilisation des listes sont moins performantes que les tableaux).

Les deques

Le dernier type de collection proposé par la bibliothèque standard est `std::deque`. Cette collection permet l'insertion et la suppression d'éléments au début et à la fin. En termes de fonctionnalités, cette collection est similaire à `std::list`, mais les accès sont plus performants. (Voir l'article sur Wikipédia pour les détails sur l'implémentation).

```
const std::deque<char> q { 'a', 'z', 'e', 'r', 't', 'y' };
```

Les conteneurs associatifs

Les maps

Les *maps* (terme qui peut être traduit par “cartes associatives” ou “tableaux associatifs”) sont des conteneurs associatifs, avec une clé et une valeur. Par défaut, les clés sont triées en utilisant le foncteur `std::less` (cela implique donc que le type utilisé comme clé doit proposer le concept “comparable plus petit que”) et que les éléments sont triés par ordre croissant.

Il existe deux types de *maps* :

- `std::map` dont toutes les clés sont distinctes (l'insertion d'un élément avec une clé existante échouera) ;
- `std::multimap`, qui peut contenir plusieurs éléments utilisant la même clé.

main.cpp

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

int main() {
    std::map<int, std::string> m {
        { 1, "hello" },
        { 2, "world" }
    };
    m.emplace(2, "everyone");

    for (auto p: m)
        std::cout << p.first << ' ' << p.second << std::endl;
    std::cout << std::endl;

    std::multimap<int, std::string> mm {
        { 1, "hello" },
        { 2, "world" }
    };
    mm.emplace(2, "everyone");

    for (auto p: mm)
        std::cout << p.first << ' ' << p.second << std::endl;
}
```

affiche :

```
1 hello
2 world

1 hello
2 world
2 everyone
```

Notez que ces deux types de *maps* sont dans le même fichier d'en-tête `<map>`.

Les maps non ordonnées

Dans les *maps* non ordonnées, les clés ne sont pas utilisées directement pour organiser les éléments, mais via une fonction de hachage. Une telle fonction prend en argument une valeur et génère une clé à partir de cela. Cette clé est généralement plus simple à manipuler que la valeur d'origine et donc plus performante.

main.cpp

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

int main() {
    auto hash_function = std::hash<std::string>{};
    std::cout << hash_function("hello, world!") << std::endl;
    std::cout << hash_function("bonjour tout le monde !") <<
std::endl;
}
```

affiche :

```
11337894577627512872
8427391994038891973
```

Ce type de collection est en particulier intéressant lorsque la clé est une chaîne de caractères. Une chaîne prend du temps a être copiée et être comparée à une autre chaîne, alors qu'une simple valeur numérique est très rapide.

Comme pour les *maps*, il existe deux versions des *maps* non ordonnées : avec clé unique ou non.

main.cpp

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> m {
        { 1, "hello" },
```

```
        { 2, "world" }
    };
    m.emplace(2, "everyone");

    for (auto p: m)
        std::cout << p.first << ' ' << p.second << std::endl;
    std::cout << std::endl;

    std::unordered_multimap<int, std::string> mm {
        { 1, "hello" },
        { 2, "world" }
    };
    mm.emplace(2, "everyone");

    for (auto p: mm)
        std::cout << p.first << ' ' << p.second << std::endl;
}
```

affiche :

```
2 world
1 hello

2 everyone
2 world
1 hello
```

Les éléments sont triés en interne selon la valeur retournée par la fonction de hachage, ce qui implique que l'ordre n'est pas déterminée (cela va dépendre de la fonction de hachage). En particulier, rien n'oblige à ce que les éléments soient triés en fonction de leur clé ou de leur ordre d'insertion dans le conteneur.

Notez que ces deux types de *maps* sont dans le même fichier d'en-tête `<unordered_map>`.

les sets

Dans les *sets* (qui peut être traduit par “ensemble”, mais cela pourrait

porter à confusion avec le concept mathématique d'ensemble, ce que ne sont pas les *sets*), les valeurs sont utilisées comme clés. Il n'y a donc pas de séparation entre clés et valeurs, c'est équivalent à utiliser les *maps* en utilisant la même valeur comme clé et valeur.

Les *sets* existent dans les mêmes versions que les *maps*, c'est-à-dire en version “multi” ou non et en version “non ordonnée” ou non.

- `std::set` ;
- `std::multiset` ;
- `std::unordered_set` ;
- `std::unordered_multiset`.

main.cpp

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<std::string> s {
        { "hello" },
        { "world" }
    };
    s.emplace("everyone");

    for (auto v: s)
        std::cout << v << std::endl;
}
```

affiche :

```
everyone
hello
world
```

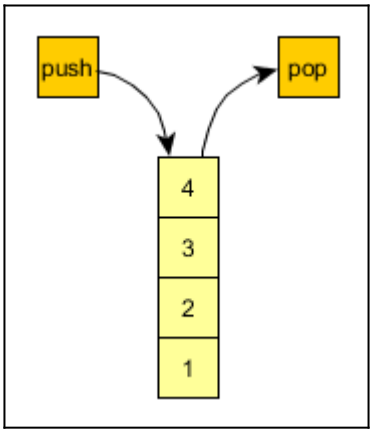
Les adaptateurs de collections

Les adaptateurs de collections ne sont pas des collections à proprement parlé, mais utilisent une collection en interne et lui ajoutent de nouvelles fonctionnalités. Par exemple, `std::stack` propose le concept de pile

(LIFO, *Last In, First Out*, “dernier arrivé, premier sorti”) et utilise par défaut en interne un `std::deque`.

std::stack

Le concept de pile est très simple à comprendre, puisque c'est quelque chose que vous avez appris lorsque vous étiez bébé : lorsque vous empiliez des petits cubes (ou n'importe quel autre type d'objets), vous créez une pile. Vous ne pouvez ajouter un élément que sur le dessus de la pile et vous ne pouvez retirer que l'élément qui se trouve sur le dessus (tous les bébés ont expérimentés que retirer un autre élément que celui qui se trouve au dessus détruit la pile… ce qu'ils apprécient généralement beaucoup).



- `push` permet d'ajouter un élément ;
- `emplace` permet de créer un élément directement sur le dessus de la pile ;
- `pop` permet de retirer un élément ;
- `top` permet d'accéder à l'élément sur le dessus de la pile ;
- `empty` permet de savoir si une pile est vide ;
- `size` permet de connaître le nombre d'éléments.

Les fonctions `push` et `emplace` sont similaires, elles permettent d'ajouter un élément sur la pile. La première prend un objet existant et le place sur la pile. Le second créé directement un objet sur la pile. La différence sera importante pour les objets complexes (ceux qui prennent du temps à être créer, ceux qui ne peuvent pas être copié). Cela sera plus clair pour vous lorsque vous verrez la création d'objets (programmation orientée objet).

main.cpp

```
#include <stack>
#include <iostream>

int main() {
    std::stack<char> s;
    s.push('a');
    s.emplace('b');
    s.push('c');
    s.pop();
    std::cout << s.top() << std::endl;
    std::cout << s.size() << std::endl;
}
```

affiche :

```
b
2
```

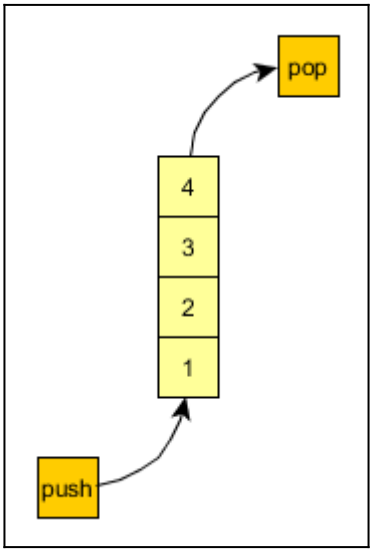
(Dans ce code, `push` et `emplace` donnent le même résultat, puisque un entier est une objet simple en mémoire).

Notez que les fonctions `top` et `pop` nécessitent que la pile contient au moins un élément, sinon cela produit un comportement indéfini (*undefined behavior*). Il est donc nécessaire de vérifier que la pile ne soit pas vide.

```
assert(!s.empty());
s.top();
s.pop();
```

std::queue

`std::queue` est un second type classique de structure de données : les files (FIFO, *First In, First Out*, “premier arrivé, premier sorti”). Vous pouvez vous représenter cette structure de données comme un tuyau, dans lequel vous poussez les éléments d'un côté et les récupérez de l'autre côté.



- `push` permet d'ajouter un élément ;
- `emplace` permet de créer un élément directement à l'entrée de la file ;
- `pop` permet de retirer un élément ;
- `front` permet d'accéder à l'élément à l'entrée de la file ;
- `back` permet d'accéder à l'élément à la sortie de la file ;
- `empty` permet de savoir si une file est vide ;
- `size` permet de connaître le nombre d'éléments.

main.cpp

```
#include <queue>
#include <iostream>
```

```
int main() {
    std::queue<char> q;
    q.push('a');
    q.emplace('b');
    q.push('c');
    q.pop();
    std::cout << q.front() << std::endl;
    std::cout << q.back() << std::endl;
    std::cout << q.size() << std::endl;
}
```

affiche :

```
b
c
2
```

Il faut également tester si la file est vide avant d'utiliser les fonctions `front`, `back` et `pop`.

```
assert(!q.empty());
q.front();
q.back();
q.pop();
```

std::priority_queue

Le dernier type d'adaptateur est similaire à `std::queue`, sauf que les éléments sont triés. De fait, lorsque vous retirer un élément de la queue, c'est celui qui a la plus grande valeur qui sort en premier.

main.cpp

```
#include <queue>
#include <iostream>

int main() {
    std::priority_queue<char> pq;
    pq.push('z');
```

```
    pq.emplace('b');
    pq.push('c');
    pq.pop();
    std::cout << pq.top() << std::endl;
    std::cout << pq.size() << std::endl;
}
```

affiche :

```
c
2
```

Pour comprendre pourquoi ce code affiche “c”, il faut regarder ce que contient la queue à chaque étape. Après les trois ajouts d'éléments, la queue contient les éléments dans l'ordre : z, c, b (du plus grand au plus petit). Après le `pop`, l'élément “z” est supprimé et l'élément avec la plus haute priorité dans la queue est “c”.

Comme pour `std::queue`, il faut que la queue contienne au moins un élément pour pouvoir utiliser `top` et ''pop'.

```
assert(!pq.empty());
pq.top();
pq.pop();
```

Chapitre précédent	Sommaire principal	Chapitre suivant