
Bibliothèque STL en C++

P. ELYAKIME

pierre.elyakime@imft.fr

E. Courcelle

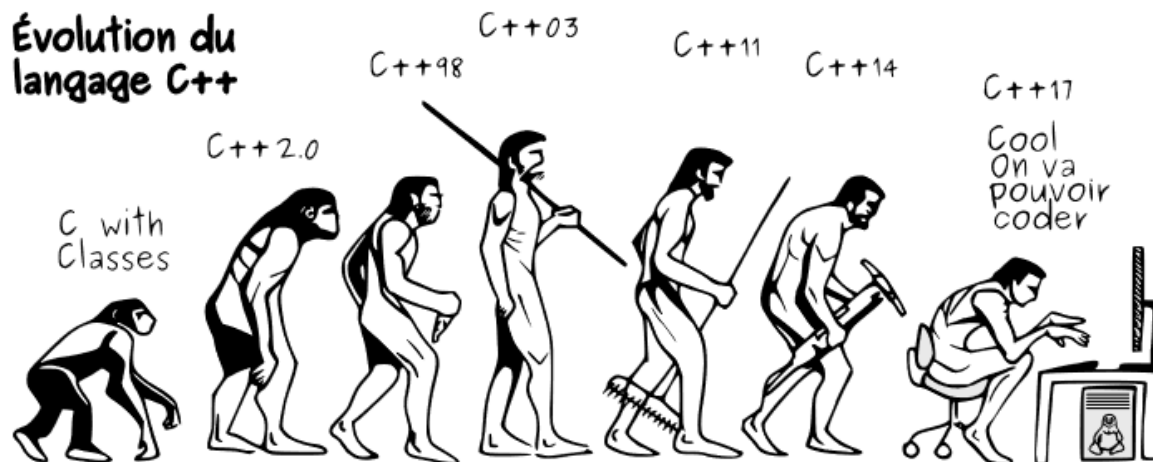
emmanuel.courcelle@inp-toulouse.fr

STL

Standard Template Library : bibliothèque C++ développée par Alexander Stepanov (SGI) à partir de 1992, inclus dans la norme ANSI/ISO C++ dès 1998 par l'Organisation International de la Normalisation (ISO) et mise en oeuvre à l'aide des templates



STL = « Bibliothèque standard basée sur des templates »



Suivre les nouveautés

ISO org : <https://isocpp.org/files/papers/p0636r0.html>
<https://isocpp.org/>

FRench User Group (FRUG) : https://github.com/cpp-frug/materials/blob/gh-pages/news/2016_n5_Bilan-Cpp17-et-attentes-Cpp20.md

GeeksforGeeks : <https://www.geeksforgeeks.org/c-plus-plus/>

...

La documentation

- SGI STL Guide :

http://www.martinbroadhurst.com/stl/stl_introduction.html

- CommentCaMarche :

<https://www.commentcamarche.net/faq/11255-introduction-a-la-stl-en-c-standard-template-library>

- CPLUCPLUS : <http://www.cplusplus.com/>

<http://www.cplusplus.com/reference/stl/>

- GeeksforGeeks : <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

- Et tant d'autres ...

Philosophie générale

Issue de concepts non orientés objet : séparation très forte entre la notion de conteneur et celle d'algorithme

→ les algorithmes classiques sont des fonctions externes qui interagissent avec les conteneurs via les itérateurs

Et puissant :

Tableaux extensibles, Listes chaînée

Tableaux associatifs

Chaînes de caractère

Queues, piles, ...

Travaillez en C++ ... avec les performances du C et sans les complications du C++ !

Vue générale de la STL

- Des conteneurs : vector, list, map, set, queue, stack, deque, ... des objets pour contenir d'autres objets, et même des séries d'objets
- Les itérateurs : une abstraction des pointeurs pour parcourir les conteneurs et accéder aux données
- Des méthodes appliqués aux conteneurs : sort, find, ... pour manipuler les données
- Une classe string permettant de gérer de manière sûre les chaînes de caractères

Les conteneurs généraux

- **Séquentiel** : le programmeur choisi l'ordre des éléments
 - **array** : tableau 1D non redimensionnable
 - **vector** : tableau 1D redimensionnable
 - **deque** : liste chaînée à accès rapide
 - **list** : liste chaînée bidirectionnelle
- **Adaptateurs de conteneur** : construits à partir de vector, deque ou list
 - **stack** : piles
 - **queue** et **priority_queue** : files d'attentes

Les conteneurs généraux

- **Associative** : collections d'éléments (ou paires) dont l'ordre est déterminé par le conteneur lui-même pour un accès rapide

(Non) ordonné :

(unordered_) map/multimap : paire d'éléments=(clé,valeur) → **table associative**

(unordered_) set/multiset : {clé} → **des ensembles**

- **Spécialisés :**

string : chaînes de caractères

bitset : tableaux de booléens

Conteneurs d'objet

Types des membres :

- **value_type T** : l'objet en lui-même
 - **(const_)reference value_type &** : sa référence (non modifiable)
 - **size_t** : nombres d'éléments
 - **(const_)iterator value_type*** : balaye le conteneur (non modifiable) (Pas tous)
 - **(const_)reverse_iterator value_type*** : balaye le conteneur à l'envers (non modifiable) (Pas tous)
 - **(const_)pointer** : des pointeurs (Pas tous)
 - ...
- et pour les maps, set :
- **key_type key** : clé d'accès aux éléments
 - **mapped_type T** : type d'élément stockés = valeur
 - **value_type pair<const key, T>** : les objets stockés dans map

Conteneurs d'objets

Fonctions membres :

→ dépend des conteneurs (voir plus loin)

Fonctions annexes des membres :

!= , ==

et pour les ordonnés

<, <=, >, >=

Les itérateurs

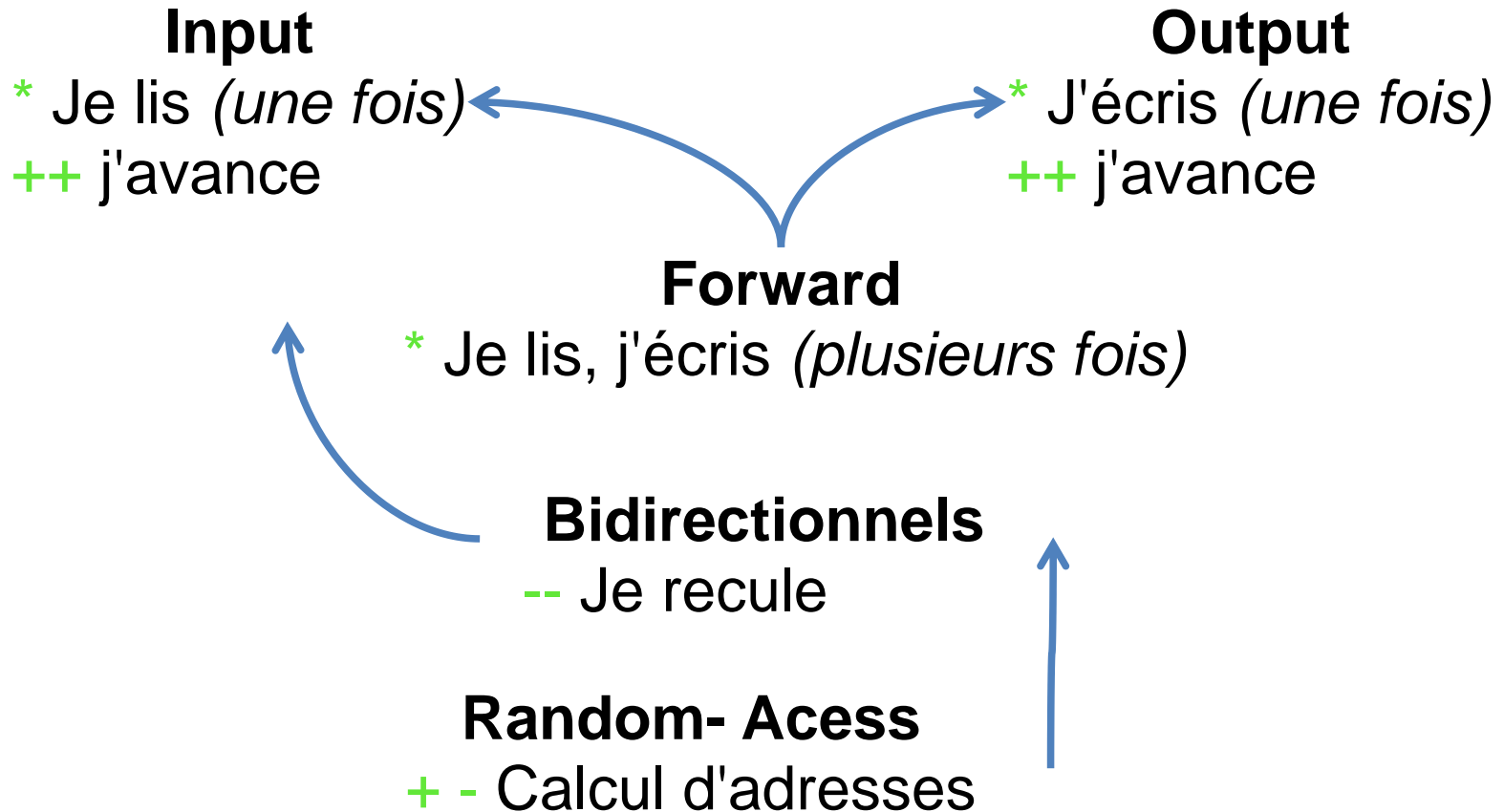
Les itérateurs

- Généralise la notion de pointeur : utilisés pour parcourir les éléments des conteneurs

```
std :: class_name <template_parameters> :: iterator name;
```

- **Pratique** : permet de se déplacer dans un conteneur sans connaître sa taille, pratique avec des conteneurs dont la taille varie souvent !
- **Code réutilisable** : en changeant le class_name
- **Dynamique** : permet d'insérer et supprimer dynamiquement des éléments quand et comme nous le voulons
- Utile pour se déplacer dans un conteneur qui n'a pas une mémoire contigue map, set

5 catégories d'itérateurs



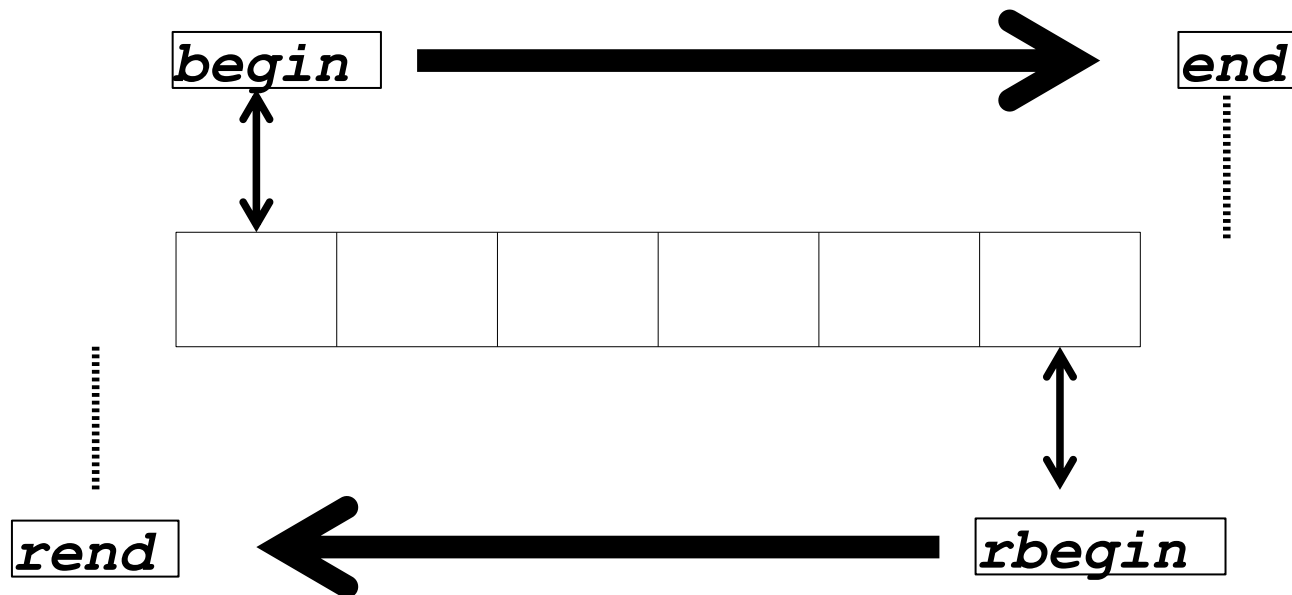
Qui peut le plus peut le moins

A chaque conteneur son itérateur

- **Bidirectionnel** : list, map, multimap, set, multiset
- **Random access** : vector, deque
- **Input/Output/Forward** : istream
- **No iterator supported** : stack, queue, priority-queue

Plusieurs types d'itérateurs

container<T>::iterator : *it* (read/write)
container<T>::reverse_iterator : *it* [au moins *Bidir*]
container<T>::const_iterator : *it* (read-only)
container<T>::const_reverse_iterator : *it*



Fonctions de déplacements :

- *begin()/end()*
- *rend()/rbegin()*
- *cbegin()/cend()*
- *crend()/crbegin()*

Intervalle : [Itérateur1, Itérateur2 [

Opérateurs / Fonctions

Opérateurs :

**(it+i)* ou *it[i]* : retourne l'élément *i* pointé par l'itérateur *it*

++ et *--* : passe à l'élément suivant et précédent

== et *!=* : compare 2 itérateurs qui pointent sur le même élément

+= *-=* : affecte en additionnant ou en soustrayant

Fonction membres :

advance(InputIt &it, Distance n) : avance l'itérateur de *n*

distance(InputIt first, InputIt last) : calcul le nombre d'éléments entre *first* et *last*

begin() / *end()* : renvoie un itérateur sur le début /fin de la sequence

prev(Distance n)/next(Distance n) : renvoie l'itérateur pointant sur l'élément qui avance/recul de *n*

Les opérations par type d'itérateurs

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->, []	= *i	*i=	++, --, +=, -=, +, -	==, !=, <, >, <=, >=

geeksforgeeks.org

Validité des itérateurs

Un itérateur est dit **valide** s'il pointe sur un élément

➔ `it*` renvoie un élément du conteneur

S'il ne pointe sur rien, il est dit invalide

Il peut devenir invalide si :

- Il n'a pas été initialisé
- Le conteneur a été redimensionné (par des insertions/suppressions par ex.)
- Le conteneur a été détruit
- L'itérateur pointe sur la fin de la séquence

Balayer un itérateur

```
conteneur<float> c; // conteneur == vector par exemple
conteneur<float>::iterator i;
conteneur<float>::reverse_iterator i;

// Lecture par le début
for (i=c.begin(); i!=c.end(); ++i) {
    cout << *i << " " ;
}

// Lecture par la fin
for (i=c.rbegin(); i!=c.rend(); ++i) {
    cout << *i << " " ;}
```

Conteneurs séquentiels

- Array
- Vector
- List
- Deque

STD::ARRAY<T>

- L'équivalent du tableau en C, à taille constante
- **Gestion automatique de la mémoire** (allocation à la création d'un array, désallocation à la fin de l'exécution du binaire)
- Accès rapide aux éléments du tableau => **itérateurs Random Access**

Complexité :

++ Accès en $O(1)$

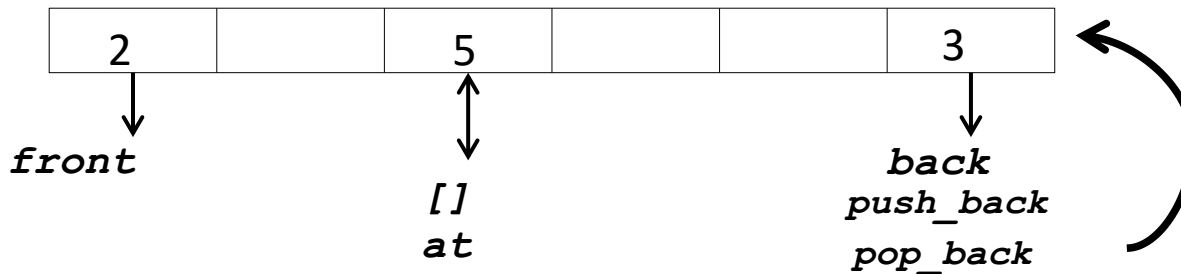
++ Insertion et suppression en $O(1)$ en fin de vector (push_back). Dans les deux cas des réallocations peuvent survenir

-- Insertion et suppression en $O(n)$ en début de vector (pop_back),

STD:: ARRAY<T>

```
template < class T, std::size_t N > struct array;
```

Itérateur
Random



```
std::array<int> a=(6, 2); // création d'un array de taille 6 rempli de 2
a.fill(5); // remplit le tableau avec la valeur specifie
a.at(2); ou a[2]; // accès aux éléments
a.data(); // Renvoie un pointeur sur le premier element de a, depuis c++11
```

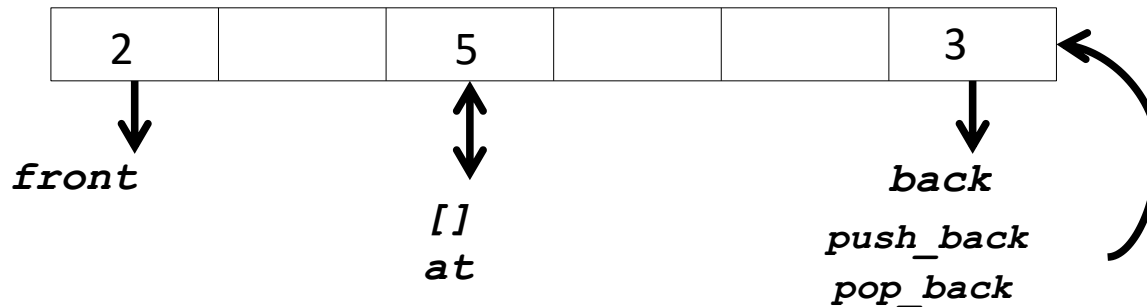
```
a.front(); // accès au 1ier élément
a.back(); // accès au dernier élément
a.push_back(3); // insertion de 3 par la fin
a.pop_back(); // suppression du dernier élément
```

STD::VECTOR<T>

- Un tableau dynamique mais à taille variable
 - **Gestion automatique et dynamique de la mémoire**
(allocation à la création d'un vector, désallocation à la fin de l'exécution du binaire)
 - Accès rapide aux éléments du tableau => **itérateurs Random Access**
 - *Complexité :*
 - ++ Accès $O(1)$
 - ++ Insertion et suppression en $O(1)$ en fin de vector (push/pop_back)
 - Insertion et suppression en $O(n)$ en début de vector
- ⇒ Une réallocation mémoire est coûteuse en terme de performances
- ⇒ Créer autant que possible la bonne taille du vecteur dès le début

STD:: VECTOR<T>

```
template < class T > class vector;
```



Itérateur
Random

```
vector<int> v(6, 2); // création d'un vector de taille 6 rempli de 2  
vector<int> v={1,2,3,4,5}; // Nouvelle façon d'initialiser un vector (c++11)  
v.at(2); ou v[2]; // accès aux éléments
```

```
v.front(); // accès au 1ier élément  
v.back(); // accès au dernier élément
```

```
v.push_back(3); // insertion de 3 par la fin  
v.pop_back(); // suppression du dernier élément
```


STD::VECTOR<T> - Exemple

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

using namespace::std;

int main()
{
    vector<int> v;
    v.push_back(1); v.push_back(2); v.push_back(3);

    cout << "Le premier élément est " << v.front() << endl;
    cout << "Le dernier élément est " << v.back() << endl;
    v.pop_back();
    cout << "Le dernier élément est maintenant " << v.back() <<
endl;
    return 0;
}
```

STD::VECTOR<T> - Exemple

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

```
using namespace std;
```

```
int main()
{
```

```
    vector<int> v;
    v.push_back(1,
```

Output :

Le premier élément est 1

Le dernier élément est 3

Le dernier élément est maintenant 2

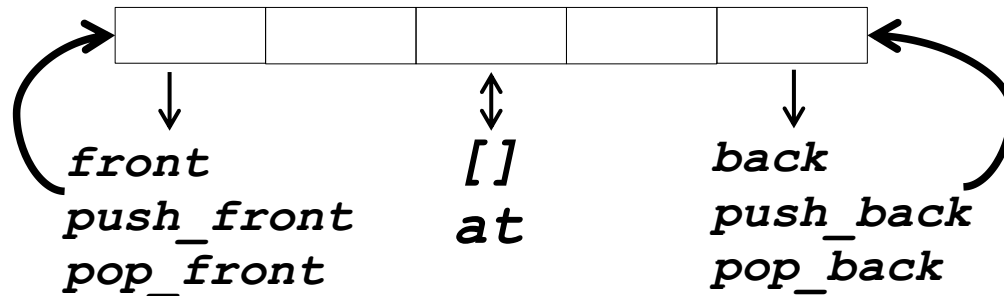
```
    cout << "Le premier élément est " << v.front() << endl;
    cout << "Le dernier élément est " << v.back() << endl;
    v.pop_back();
    cout << "Le dernier élément est maintenant " << v.back() <<
endl;
    return 0;
}
```

STD :: DEQUE<T>

- Double Ended QUEUE = queue à deux bouts
 - ➔ Tableau à double entrée avec insertion et suppression rapide aux extrémités (fin,début)
- Stockage non contiguë des éléments (\neq vector)
- Stockage automatiquement **contracté et dilaté** selon les besoins
- Accès rapide aux éléments du tableau => **itérateurs Random Access**
- *Complexité :*
 - ++ Accès rapide aux éléments en $O(1)$
 - ++ Insertion et suppression en début et fin en $O(1)$
 - Insertion et suppression d'éléments lente en $O(n)$

STD :: DEQUE<T>

```
template <class T> class deque;
```



Itérateur
Random
Access

```
std::deque<int> d(6, 2); // création d'un deque de taille 6 rempli de 2  
d.at(2); ou d[2]; // accès aux éléments
```

```
d.front(); // accès au 1ier element
```

```
d.push_front(2); // insère des éléments (ici 2) par le début
```

```
d.pop_front(); // supprime le 1ier element
```

```
d.back(); // accès au dernier element
```

```
d.push_back(3); // insère des éléments (ici 3) par la fin
```

```
d.pop_back(); // supprime le dernier element
```

STD::DEQUE<T> - Exemple

```
#include <iostream>
#include <deque>

using namespace::std;

int main()
{
    deque<int> dq;
    dq.push_front(10); dq.push_back(15);
    dq.push_front(14);dq.push_front(20);

    cout << "Le premier élément est " << dq.front() << endl;
    cout << "Le dernier élément est " << dq.back() << endl;
    dq.pop_back();
    dq.pop_front();
    cout << "Le premier élément est " << dq.front() << endl;
    cout << "Le dernier élément est " << dq.back() << endl;
    return 0;
}
```

STD::DEQUE<T> - Exemple

```
#include <iostream>
#include <deque>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    deque<int> dq;
```

```
    dq.push_front(10);
```

```
    dq.push_front(14); dq.push_front(20);
```

```
    cout << "Le premier élément est " << dq.front() << endl;
```

```
    cout << "Le dernier élément est " << dq.back() << endl;
```

```
    dq.pop_back();
```

```
    dq.pop_front();
```

```
    cout << "Le premier élément est " << dq.front() << endl;
```

```
    cout << "Le dernier élément est " << dq.back() << endl;
```

```
    return 0;
```

```
}
```

Output :

Le premier élément est 20

Le dernier élément est 15

Le premier élément est 14

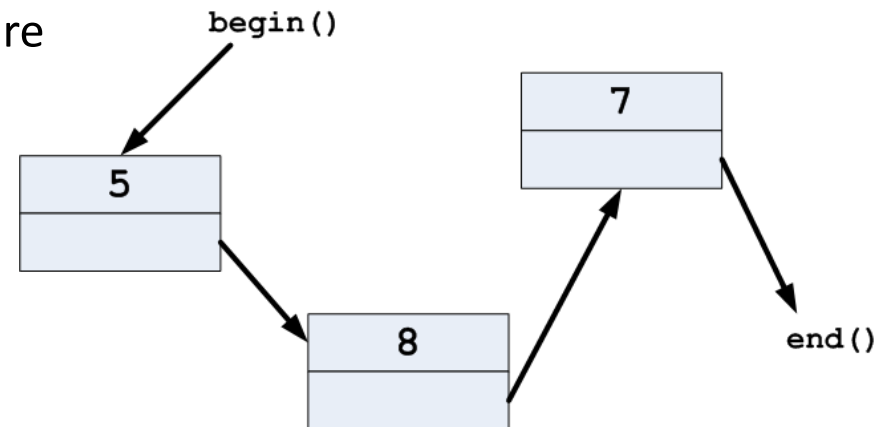
Le dernier élément est 10

STD::LIST<T>

- List doublement chaînée : itère dans les deux sens => **itérateur bidirectionnel**
- Chaque « case » contient un élément et un pointeur sur la « case suivante » située ailleurs dans la mémoire
- Pas nécessairement contigue en mémoire

!! insertion et suppression rapide de tout éléments : avantage vs vector et deque

!! pas d'itérateurs à accès direct => recherche d'éléments très lentes



Complexité : n = taille du vector

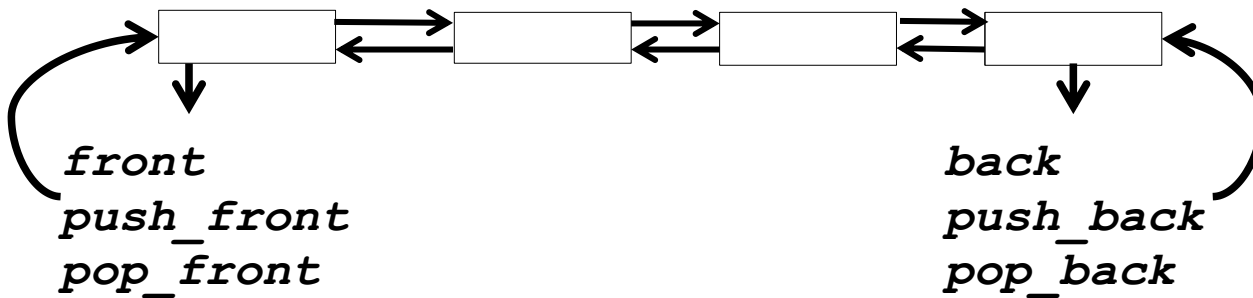
++ Insertion ou suppression en $O(1)$

++ Tri (avec sort) en $O(n \cdot \log(n))$

-- Recherche : $O(n)$ en général, $O(1)$ pour le premier et le dernier maillon

STD::LIST<T>

```
template <class T > class list;
```



Itérateur
bidirectionnel

```
std::list<int> l1st(6) // création d'une liste de taille 6
```

```
l1st.front(); // accès au 1ier élément
```

```
l1st.push_back(); // insère des éléments au début
```

```
l1st.pop_front(); // supprime le 1ier element
```

```
l1st.back(); // accès au dernier element
```

```
l1st.push_back(); // insère des éléments à la fin
```

```
l1st.pop_back(); // supprime le dernier element
```


STD::LIST<T> - Exemple

```
#include <iostream>
#include <list>
using namespace::std;

int main() {

    list<int> lst; // création d'une liste
    // On remplit la liste
    lst.push_back(5); lst.push_back(6); lst.push_back(7);
    lst.pop_back(); // enleve le dernier element et supprime le 7

    // utilisation d'un itérateur pour parcourir la liste lst
    for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
        cout << " " << *it;
    cout << "\n";
    // Afficher le premier et dernier element
    cout << "Premier élément : " << lst.front() << endl;
    cout << "Dernier élément : " << lst.back() << endl;

    return 0;
}
```

STD::LIST<T> - Exemple

```
#include <iostream>
#include <list>
using namespace std;

int main() {

    list<int> lst; //
    // On remplit la liste
    lst.push_back(5);
    lst.push_back(6);
    lst.pop_back(); // enleve le dernier element et supprime le 7

    // utilisation d'un itérateur pour parcourir la liste lst
    for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
        cout << " " << *it;
    cout << "\n";
    // Afficher le premier et dernier element
    cout << "Premier élément : " << lst.front() << endl;
    cout << "Dernier élément : " << lst.back() << endl;

    return 0;
}
```

Output :

5 6

Premier élément : 5

Dernier élément : 6

Initialisation depuis c++11

- Liste d'initialisation

```
// Liste d'initialisation c++11:  
deque<string> mots1 {"le", "frogurt", "est", "aussi", "maudit"};  
vector<int> v{1,2,3,4,5};  
list<int> l{1,2,3,4,5};
```

- Copier une séquence d'éléments à partir d'un autre vecteur

```
deque<string> mots2 (mots1.begin(), mots1.end());
```

- Copier les éléments d'un autre vecteur

```
vector<int> vec (v);
```

- Initialiser avec une même valeur dans une taille donnée

```
list<string> mots3 (5, "Mo");
```

Fonctions membres communes à vector, deque, list

Modificateurs :

```
clear(); // taille du tableau est nulle  
insert(); // insere des elements  
erase(); // efface des elements  
emplace(); // construit des elements en mémoire  
emplace_front(); // construit des elements en place au debut  
emplace_back(); // construit des elements en place a la fin  
resize(); // modifie le nombre d'elements stockes  
swap(); // permute les contenus
```

Fonctions membres communes à vector, deque, list

Capacité :

- `max_size()`; // retourne le plus grand nombre possible d'élément
- `size()`; // donne la taille du tableau
- `empty()`; // vérifie si le conteneur est vide

Pour deque et vector :

- `shrink_to_fit()` // réduit l'utilisation de la mémoire en libérant la mémoire inutilisée (C++11)

Pour vector :

- `reserve()` // réserve de l'espace mémoire
- `capacity()` // renvoie le nombre d'éléments qui peuvent être contenus dans l'espace mémoire actuellement alloué

Fonctions membres communes à vector, deque, list

Fonctions annexes :

`== != < <= > >= //` compare lexicographiquement les valeurs
dans la list

Fonctions membres spécifiques à LIST<T>

`merge(); // fusionne deux listes`

`splice(); // déplace les éléments d'une autre liste`

`remove(); remove_if(); // supprime des éléments`

`reverse(); // inverse l'ordre des éléments`

`unique(); // supprime les doublons successifs`

`sort(); // trie les éléments en $n \cdot \log(n)$`

Les conteneurs adaptateurs

- stack (pile)
- queue (file)
- priority_queue

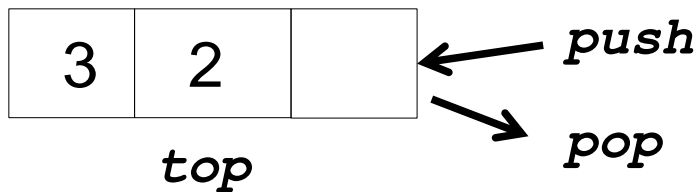
Classes patrons construites à partir des conteneurs ***vector***, ***deque*** ou ***list*** et qui modifient leur interface en les restreignant et en les adaptant à des fonctionnalités données

La pile (stack)

```
template <class T, class Container = deque<T> > class stack;
```

- `queue<T> pile;` => avec `deque`
- `queue<T, vector<T>> pile;` => avec `vector`
- `queue<T, list<T>> pile;` => avec `list`

«Dernier arrivé, premier sorti» ou LIFO (Last In, First Out)



itérateur **Aucun**

```
pile.top()=99; // Accede a l'element en haut de la pile, modifiable
```

```
pile.push(a); // Ajoute l'element par le haut de la pile
```

```
pile.pop();   // Retire l'element par le haut de la pile
```

Exemple

```
#include <stack>
#include <iostream>
#include<vector>
using namespace std;

int main()
{
    stack< int, vector<int> > pile;
    cout << "taille initiale : " << pile.size() << endl;
    for (int i=0; i<10; i++)
        pile.push(i*i);
    cout << "taille après for : " << pile.size() << endl;
    cout << "sommet de la pile : " << pile.top() << endl;
    pile.top() = 99; // on modifie le sommet de la pile
    cout << "on déplie : ";
    while (!pile.empty()) {
        cout << pile.top() <<" ";
        pile.pop();
    }

    cout << endl;
    cout << "Taille de la pile : " << pile.size() << endl;
    return 0;
}
```

Exemple

```
#include <stack>
#include <iostream>
#include<vector>
using namespace std;
```

```
int main()
{
```

```
    stack< int, vector<int> > pile;
    cout << "taille initiale : 0\n";
    for (int i=0; i<10; i++)
        pile.push(i);
    cout << "taille après for : 10\n";
    cout << "sommet de la pile : 81\n";
    pile.top() = 99; // modification du sommet
    cout << "on déplie : ";
    while (!pile.empty()) {
        cout << pile.top() << " ";
        pile.pop();
    }
```

```
    cout << endl;
    cout << "Taille de la pile : " << pile.size() << endl;
    return 0;
}
```

Output :

taille initiale : 0

taille après for : 10

sommet de la pile : 81

on déplie : 99 64 49 36 25 16 9 4 1 0

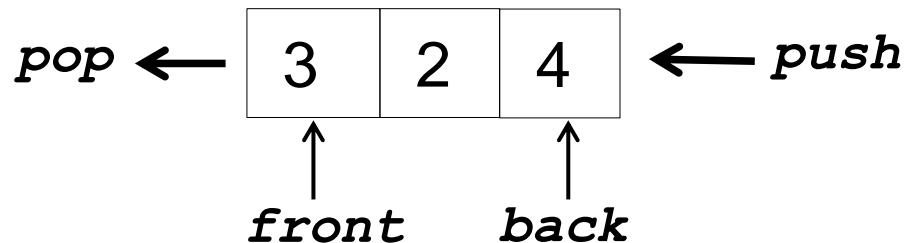
Taille de la pile : 0

La file (queue)

```
template <class T, class Container = deque<T> > class queue;
```

- `queue<T> file;` \Rightarrow avec `deque`
- `queue<T, vector<T>> file;` \Rightarrow avec `vector`
- `queue<T, list<T>> file;` \Rightarrow avec `list`

« Premier arrivé, premier sorti » ou FIFO (First In, First Out)



itérateur **Aucun**

```
file.front(); // Accède à l'élément en tête de file, le 1ier  
file.back();  // Accède à l'élément en fin de file, le dernier
```

```
file.pop();   // Retire l'élément situé en tête de file  
file.push();  // Ajoute un élément à la fin de la file
```

La file (queue)

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

using namespace std;

int main()
{
    queue<int> file;
    file.push(1);
    file.push(4);
    file.pop();

    cout << "Taille de la file : " << file.size() << endl;
    while (!file.empty())
    {
        cout << file.front() << endl;
        file.pop();
    }

    return 0;
}
```

La file (queue)

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

```
using namespace std;
```

```
int main()
{
```

```
    queue<int> file;
    file.push(1);
    file.push(4);
    file.pop();
```

```
    cout << "Taille de la file : " << file.size() << endl;
```

```
    while (!file.empty())
```

```
    {
```

```
        cout << file.front() << endl;
        file.pop();
```

```
    }
```

```
    return 0;
```

```
}
```

Output :

Taille de la file : 1

4

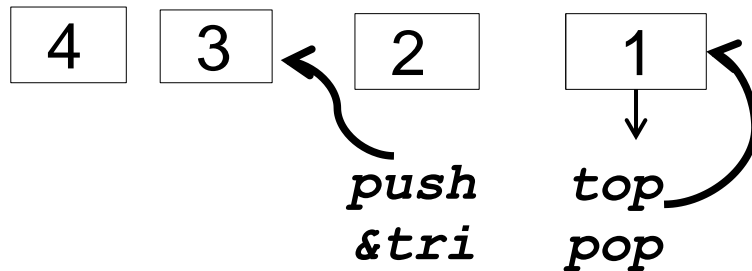
File d'attente prioritaire (priority_queue)

```
template <class T, class Container = vector<T>, class Compare comp = less<T> > class priority_queue;
```

- `priority_queue<T> file;` \Rightarrow `vector<T>` et `less<T>`
- `priority_queue<T, deque<T>, greater<T>> file;`

« NOT FIFO logic »

File dont les éléments sont introduits uniquement par le haut. A chaque introduction, ils sont classés tel que l'élément du haut respecte la relation d'ordre donnée



itérateur **Aucun**

```
pq.push(a); // Ajoute un element à la fin et tri selon la relation d'ordre
```

```
pq.top(); // Accède à l'element en fin de file, le dernier
```

```
pq.pop(); // Supprime l'element situe en tete de file
```

Example

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

using namespace std;

int main()
{
    int value;
    priority_queue<int, vector<int> > pq;
    pq.push(1); pq.push(2); pq.push(3);

    while(!pq.empty())
    {
        value = pq.top();
        pq.pop();
        cout<<value<< " ";
    }
    return 0;
}
```


Example

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

using namespace std;

int main()
{
    int value;
    priority_queue<int, vector<int> > pq;
    pq.push(1); pq.push(2); pq.push(3);

    while(!pq.empty())
    {
        value = pq.top();
        pq.pop();
        cout<<value<< " ";
    }
    return 0;
}
```

Output : 3 2 1

Exemple - greater

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

using namespace std;

int main()
{
    int value;
    priority_queue<int, vector<int>, greater<int> > pq;
    pq.push(1); pq.push(2); pq.push(3);

    while(!pq.empty())
    {
        value = pq.top();
        pq.pop();
        cout<<value<< " ";
    }
    return 0;
}
```

Exemple - greater

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

```
using namespace std;
```

Output : 1 2 3

```
int main()
{
    int value;
    priority_queue<int, vector<int>, greater<int> > pq;
    pq.push(1); pq.push(2); pq.push(3);

    while(!pq.empty())
    {
        value = pq.top();
        pq.pop();
        cout<<value<< " ";
    }
    return 0;
}
```

Fonctions membres communes

```
pile.swap(); // Permute les contenus
```

```
pile.empty(); // Retourne true si la pile est vide sinon false
```

```
pile.size(); // fournit le nombre d'éléments de la pile
```

Les conteneurs associatifs

- map/multimap
- set/multiset

Conteneurs associatifs

- Permet de trouver un élément, non plus en fonction de sa place dans le conteneur mais en fonction de sa valeur (ou d'une partie de la valeur nommée clé)
- A chaque insertion d'élément, le conteneur ordonne la table grâce à un opérateur de comparaison choisi lors de la construction (par défaut <)

++ recherche rapide d'éléments à partir d'une clé

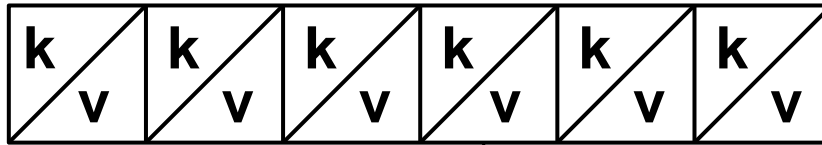
map/multimap

- Formé par des paires d'éléments : une clé et une valeur
- Construit à partir du patron de classe : `T pair<T1,T2> paire_name`

```
template <class key, class T, class Compare=less<Keys> > class  
map/multimap;
```

- map vs multimap → unicité des clés vs plusieurs éléments ont la même clé
Ex : Annuaire → Dupont et Dupont peuvent avoir des numéros de téléphone identiques ou différents avec multimap et pas avec map
- Trié automatiquement par ordre croissant des clés
- Accès rapide à la valeur associée à clé en $O(\log(n))$

map/multimap



[], at

Trié par ordre croissant des clés

itérateur **Bidirectionnel**
(opère sur les **paires**)
clés **const**

itr ptr ref

```
map<char, int> m;  
m['S']; // Accès à la valeur associée à la clé 'S'  
m['S'] = 5 // Cré la clé 'S' avec sa valeur associée 5  
make_pair('S',5); // Cré la clé 'S' avec sa valeur associée 5
```

```
map::itérateur it;  
*it; // représente l'élément  
it->first; it->second; // accès à la clé (first) et sa valeur (second)  
(*it).first; (*it).second;
```

→ Il est fortement déconseillé de modifier la valeur d'un élément d'une map par le biais d'un itérateur

Fonctions membres

Insertion / Suppression :

```
m.insert(); // insere un element (std::pair<>), à une position donnee  
m.erase(); // supprime un element en utilisant la clé
```

Autres fonctions :

```
m.find(kle); // fournit un it. sur un des elts ayant kle  
m.swap(); // echange les contenus de 2 tables de meme type  
m.extract(); // C++17 : extrait un nœud (clé+valeur) d'une map  
m.merge(m1); // C++17 : fusionne m1 dans m  
m.size(); // retourne le nombre de la map  
m.empty(); // retourne true si la carte est vide sinon false
```

Pour multimap :

```
m.count(kle); // nb d'elts ayant kle  
m.lower/upper/_bound(kle); // fournit un it. sur le 1er elt ayant kle  
m.upper_bound(kle); // fournit un it. sur le dernier elt ayant kle
```

extract/merge

`m.extract()` : seul moyen pour changer la clé d'une map sans réallouer

```
map<int, string> m{{1, "mango"}, {2, "papaya"}, {3, "guava"}};  
auto nh = m.extract(2);  
nh.key() = 4;  
m.insert(move(nh));  
// m == {{1, "mango"}, {3, "guava"}, {4, "papaya"}}
```

`m.merge()` : fusionne 2 tables associatives

```
m1.insert(make_pair("earth", 1))  
m1.insert(pair<string, int>("moon", 2));  
m2.insert(pair<string, int>("moon", 2));  
m2.insert({"sun", 3});  
m1.merge(m2);
```

Pour la compilation : `$g++ -std=c++17 mergeMap.cpp -o merge.exe`

Exemple complet avec merge

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

using namespace std;

template<typename Conteneur>
void printContainer(const Conteneur&
cont, const string& mess)
{
    cout << mess;
    typename Conteneur::const_iterator
it=cont.begin();
    for(; it!=cont.end();it++) {
        cout << "(" << it->first << ":
" << it->second << ") ";
    }
    cout << endl;
}
```

```
int main()
{
    map<string, int> m1;
    map<string, int> m2;

    m1.insert(make_pair("earth", 1));
    m1.insert(pair<string, int>("moon",2));
    m2.insert(pair<string, int>("moon",2));
    m2.insert({"sun", 3});

    printContainer(m1, "m1 : ");
    printContainer(m2, "m2 : ");

    m1.merge(m2);

    printContainer(m1, "m1 apres merge : ");
}
```

Exemple complet avec merge

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

```
using namespace std;
```

```
template<type
```

```
void printCon
cont, const s
{
```

```
    cout << mes
    typename Co
    it=cont.begin
```

```
    for(; it!=con
```

```
        cout << "(" << it->first << ":
        << it->second << ")" << " ";
```

```
    }
    cout << endl;
}
```

```
int main()
{
    map<string, int> m1;
```

Output :

m1 : (earth: 1) (moon: 2)

m2 : (moon: 2) (sun: 3)

m1 apres merge : (earth: 1) (moon: 2) (sun: 3)

```
printContainer(m1, "m1 apres merge : ");
```

```
}
```

set/multiset

- Cas particulier de map = ensemble de clés (ce ne sont plus des paires cle/val)
→ même construction, insertion, fonctions membres
- Ensemble d'éléments constitués de **valeurs constantes** -> **Non modifiables**

```
template <class key, class Compare=less<Keys> > class set/multiset;
```

- set vs multiset → unicité des clés vs plusieurs éléments ont la même clé
- Trié automatiquement par ordre croissant des clés selon un opérateur de comparaison choisi à la construction
- Accès rapide à la valeur associée à clé en $O(\log(n))$

set/multiset

k	k	k	k	k	k
---	---	---	---	---	---

itérateur **Bidirectionnel**
(opère sur les **paires**)
clés **const**

Trié par ordre croissant des clés

```
set<int> m;  
set::iterateur it;  
cout << *it; // represente elt de l'ensemble  
*it = ... ; // INTERDIT .. Valeurs constantes donc non modifiables
```

Insertion d'éléments possibles

Pas d'accès aux éléments avec []

Accès avec une méthode de recherche

Depuis la norme C++11 ...

Nouvelle syntaxe disponible pour « for »

```
list<int> l{0, 1, 2, 3, 4, 5};
```

```
for (const int& i : l) // access by const reference  
    std::cout << i << ' ';
```

```
std::cout << '\n';
```


Le type « auto »

- **auto** : utilisé lors de l'initialisation d'une variable à la place du type de la variable

```
auto d = 5.0 ;    // 5.0 is a double literal, so d will be type
double
auto i = 1+2 ;    // 1 + 2 evaluates to an integer, so i will be

int add (int x, int y)
    return  x  +  y ;

int main()
{
    auto sum = add (5 ,6 ) ;    // add() returns an int, so sum will be
    type int
    return  0 ;
}
```

La classe string

string

- Simplifie les tableaux de chaînes de caractères en C
- Construit à partir du conteneur vector → `vector<char>`

```
typedef basic_string<char> string;
```

- Accès aux éléments par :
 - `s[2]`, `s.at(2)`
 - `s.front()`, `s.back()` : premier et dernier caractère de `s`

- Simple à utiliser :

```
string s1="cogito";
```

```
s1 += " ergo sum";
```

```
cout << "Dans " << s1 << ", le 3ieme caractere est : " << s1[2];
```

Fonctions de manipulation

`+=`, `append()`, `push_back()` : concaténation, ajout à la fin de la chaîne
`insert()` : insère une sous-chaîne dans une chaîne
`erase()` : supprime une sous-chaîne dans une chaîne
`replace()` : remplace une partie d'une chaîne par une sous-chaîne
`pop_back()` : efface le dernier caractère

`size(s)`, `length(s)` : retourne la taille de `s`
`reserve()` : réserve de la mémoire
`clear()` : supprime toute la chaîne
`empty()` : retourne `true` si la sous chaîne est vide sinon `false`

`stod/stof/stoi/stol` : convertit un `double/float/integer/long` en `string`
`to_string()` : convertit un nombre en `string`

Fonctions de manipulation

`substr()` : extrait une sous-chaine d'une chaine

`compare()` : compare deux chaines

`empty()` : vérifie qu'une chaine est vide

`find()` : recherche d'une sous-chaine dans une chaine

`size_t npos` : valeur retournée (-1) si avec la fonction `find` la sous chaîne n'est pas trouvé dans la chaîne

Exemple

```
#include <string>
#include <iostream>
#include <stdio.h>

using namespace std;

int main()
{
    // Initialisation
    string hello = "bonjour ";
    string amis  = "les amis";
    string bye   = "bye bye ";

    // Concatenation
    string P1 = hello + amis;
    string P2 = bye;
    P2 += amis;
    cout << P1 << '\n';
    cout << P2 << '\n';

    // Acces et affichage d'un caractere
    cout << hello[3] << '\n';
```

```
// Longueur d'une chaine
int l = hello.length();

// Recherche de caractere
string::size_type p1 = hello.find('o');
string::size_type p2 = hello.find('o',p1+1);

if (hello.find('z') == string::npos)
    cout << "Pas de z\n";

// Acces aux caracteres d'une chaine
string h1 = hello.substr(3,4);

// Suppression d'une partie de la chaine
hello.erase(3,3);

// Insertion dans une chaine
hello.insert(3,"heu");

// Conversion string -> char*
string p = (string)"bonjour " + (string)"les amis";
const char* phrase = p.c_str();
printf("%s\n",p.c_str());
}
```

Output :
bonjour les amis
bye bye les amis
j
Pas de z
bonjour les amis

```
1.
{
    // Initialisation
    string hello = "bonjour ";
    string amis  = "les amis";
    string bye   = "bye bye ";

    // Concatenation
    string P1 = hello + amis;
    string P2 = bye;
    P2 += amis;
    cout << P1 << '\n';
    cout << P2 << '\n';

    // Acces et affichage d'un caractere
    cout << hello[3] << '\n';
}
```

emple

```
// Longueur d'une chaine
int l = hello.length();

// Recherche de caractere
string::size_type p1 = hello.find('o');
string::size_type p2 = hello.find('o',p1+1);

if (hello.find('z') == string::npos)
    cout << "Pas de z\n";

// Acces aux caracteres d'une chaine
string h1 = hello.substr(3,4); // jour

// Suppression d'une partie de la chaine
hello.erase(3,3); // bonr

// Insertion dans une chaine
hello.insert(3,"heu"); // bonheur

// Conversion string -> char*
string p = (string)"bonjour " + (string)"les amis";
const char* phrase = p.c_str();
printf("%s\n",p.c_str());
}
```

Les algorithmes

#include <algorithm>

Ensemble de fonctions/algorithmes appliqués aux conteneurs via les itérateurs

Travaillent sur des intervalles

Permettent de mettre en relation des conteneurs de types différents

Généralités

int count(Inp first, Inp last, const T& val);

→ compte les valeurs égales à une valeur donné

int count_if(Inp first, Inp last, Pred pred);

→ compte les valeurs égales à une valeur donné avec une condition

function for_each(Inp first, Inp last, Func func);

→ permet d'appliquer un traitement (non mutant) à tous les éléments

```
void afficher (int a) { cout << a << endl; }  
...  
vector<int>    v;  
// ... on remplit le vecteur  
  
for_each (v.begin(), v.end(), afficher);
```

Comparaison

```
bool equal(Inp1 first1, Inp1 last1, Inp2 first2);
```

→ détermine si deux conteneurs sont égaux en comparant leur contenu

```
bool lexicographical_compare(Inp1 f1, Inp1 l1, Inp2 f2, Inp2 l2);
```

→ compare les 2 intervalles [f1,l1] et [f2,l2]

```
T min(const T& a, const T& b)
```

```
T max(const T& a, const T& b)
```

```
T min_element(FwdIt first, FwdIt last)
```

```
T max_element(FwdIt first, FwdIt last)
```

→ calcul des min, max, ...

Recherche, remplacement

Fwd adjacent_find(Fwd first, Fwd last)

→ recherche deux valeurs consécutives égales

InputIt find(Ind first, Inp last, const T& value)

→ recherche une valeur

Fwd1 search(Fwd1 first, Fwd1 last, Fwd2 first2, Fd2 last2)

→ recherche une séquence d'éléments

void replace(Fwd first, Fwd last, const T& old, const T& new)

→ remplace les valeurs d'un conteneur

Copie, suppression

OutputIt copy(InputIt first, InputIt last, OutputIt result)

→ recopie le contenu d'un intervalle dans un conteneur

void fill(Fwd first, Fwd last, const T& val);

→ remplit le conteneur avec une valeur donnée

void generate(Fwd first, Fwd last, Generator gen);

→ produit une suite de valeurs dans un conteneur résultant de l'application d'une fonction

Fwd remove_if (Fwd first, Fwd last, Out result, Predicate pred);

→ suppression des valeurs qui correspondent à un critère

Réarrangements

void random_shuffle(Rnd first, Rnd last);

→ distribue uniformément les valeurs d'un conteneur

void reverse(Bidi first, bidi last);

→ inversion des valeurs d'un conteneur par rapport à un pivot

void rotate(Fwd first, Fwd middle, Fwd last);

→ rotation des valeurs d'un conteneur

Fwd swap_ranges(Fwd first1, Fwd last1, Fwd first2);

→ échange le contenu de deux conteneurs

!! Attention à l'allocation mémoire

Tri et fusion

`void sort(RdmAIt first, RdmAIt last, Compare comp);`
→ tri croissant des valeurs de [first,last], tri décroissant si `comp=greater`

`FwdIt lower_band(FwdIt first, FwdIt last, const T& val);`
`FwdIt upper_band(FwdIt first, FwdIt last, const T& val);`
→ recherche d'une borne inférieure/supérieure pour les valeurs d'un conteneur répondant à un critère donné

`FwdIt equal_range(FwdIt first, FwdIt last, const T& val);`
→ recherche les zones d'égalité

`OutputIt merge(InptIt f1, InptIt l1, InptIt2 f2, InptIt2 l2, OutputIt result);`
→ fusionne des séquences triées

Exemple - sort

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

using namespace std;

void show(int a[]) {
    for(int i = 0; i < 10; ++i)
        cout << a[i] << " ";
}

int main()
{
    int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "\n The array before sorting is : ";
    show(a);
    sort(a, a+10);
    cout << "\n\n The array after sorting is : ";
    show(a);
    return 0;
}
```


Example - sort

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

```
using namespace std;
```

```
void show(int a[]) {
    for(int i=0; i<10; i++)
        cout << a[i] << " ";
}
```

```
int main() {
```

```
    int a[10];
```

```
    cout << "The array before sorting is : ";
```

```
    show(a);
```

```
    sort(a, a+10);
```

```
    cout << "\n\n The array after sorting is : ";
```

```
    show(a);
```

```
    return 0;
```

```
}
```

Output :

The array before sorting is : 1 5 8 9 6 7 3 4 2 0

The array after sorting is : 0 1 2 3 4 5 6 7 8 9

#include <numeric>

Algorithmes permettant de réaliser des calculs sur les éléments d'un ou de deux conteneur(s) :

`T accumulate(InputIt first, InputIt last, T val, Pred pr);`
→ accumulation de données dans une variable

`T inner_product(Input1 first1, Input1 last1, Input2 first2, T init);`
→ somme le produit des éléments de deux conteneurs

`OutputIt partial_sum(InputIt first, InputIt last, OutputIt d_first);`
→ somme partielle des valeurs d'un conteneur

`OutputIt adjacent_difference(InputIt first, InputIt last, OutputIt d_first);`
→ différence entre deux éléments adjacents

Exemple accumulate

```
vector<float> v;  v.push_back  
(3.14);  
v.push_back (5);  
// ...  
float resultat = accumulate  
    (v.begin(), v.end(), 1.0, times<float>);  
  
cout << "Le résultat du produit est : " << resultat << endl;
```

#include <complex>

La bibliothèque complexe implémente la classe complexe pour contenir des nombres complexes sous forme cartésienne et plusieurs fonctions et surcharges pour fonctionner avec eux

<http://www.cplusplus.com/reference/complex/>

C++ avancé

- Librairie BOOST
- smart pointeur

Bibliothèque BOOST

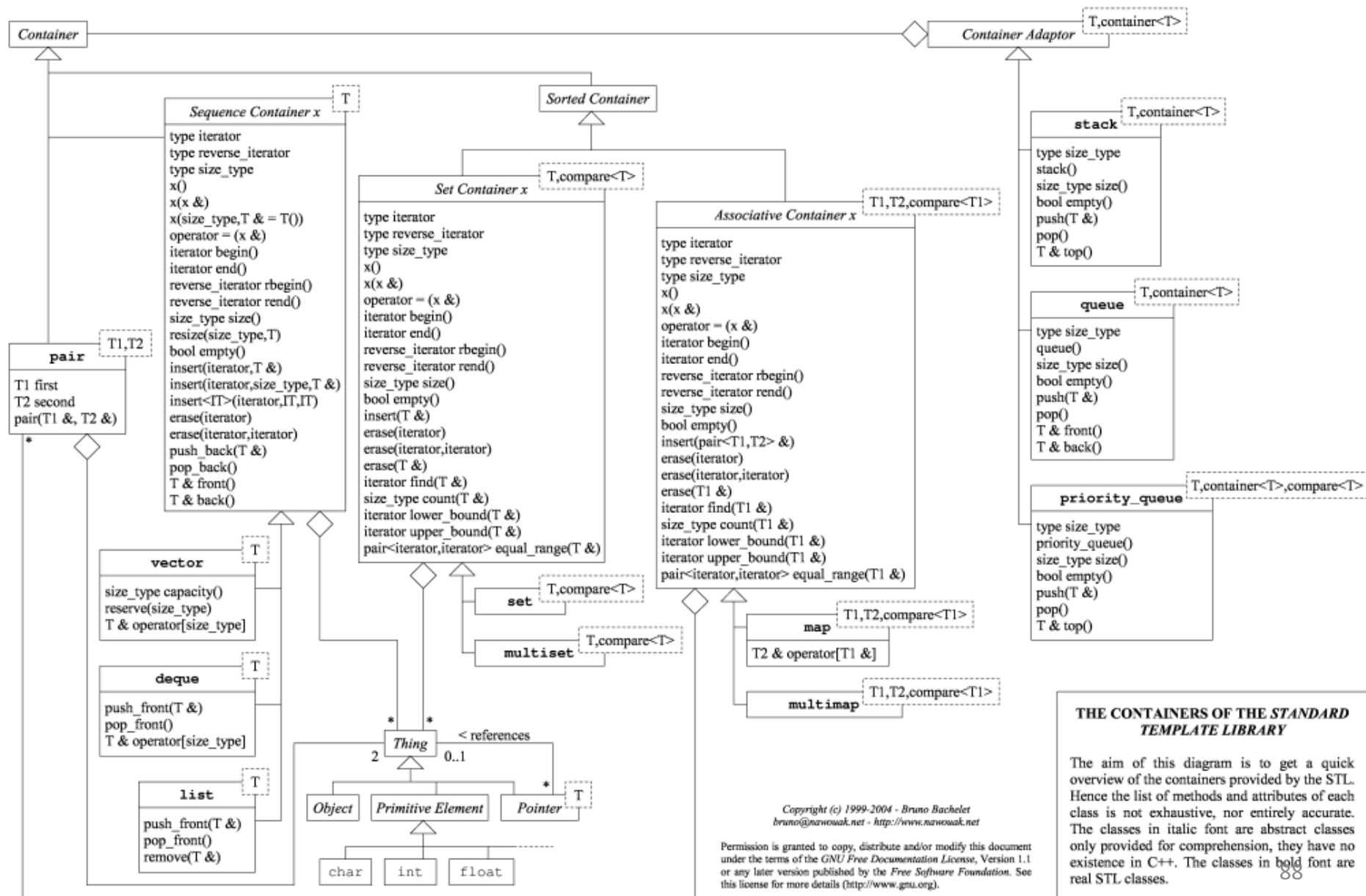
- Ensemble de classes formant un référentiel complémentaire à la bibliothèque STL
- Intégration des classes de BOOST dans les nouvelles normes C++ : C++11, C++14, C++17
- Sous licence de logiciel libre ➔ installation et linkage lors de la compilation pour l'utilisation
- Important pour le MPI lorsqu'on utilise des conteneurs de la STL : string, vector, ...

Smart pointeur

- Les pointeurs traditionnels présentent des insuffisances :
 - Si Allocation alors Désallocation sinon fuite mémoire → Seg. Fault
 - Interdit de désallouer une zone mémoire non allouée → Seg. Fault
 - Interdit de désallouer un pointeur déjà désalloué → Seg. Fault
 - ...
- Smart pointeur : corrige les insuffisances des pointeurs en ajoutant de l'intelligence
 - ➔ Classe qui encapsule la notion de pointeur en offrant une sémantique qui gère les opérations liées à la durée de vie des pointeurs (création, copie, destruction, ...), à la taille de la mémoire allouée (vérification des bornes)

unique_ptr()
shared_ptr()
weak_ptr()

Diagramme UML de la STL



Références

Livre :

Apprendre le C++, C. Delannoy

Site Web :

<https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

<https://www.commentcamarche.net/faq/11255-introduction-a-la-stl-en-c-standard-template-library>

<http://www.cplusplus.com/reference/stl/>

<https://cpp.developpez.com/cours/stl/>

Présentations :

<http://tvaira.free.fr/dev/cours/cours-conteneurs-stl.pdf>

https://calcul.math.cnrs.fr/attachments/spip/Documents/Journees/dec2005/C_avance.pdf

https://ensiwiki.ensimag.fr/images/1/15/Slides_cours3_c.pdf