Bibliothèque STL en C++

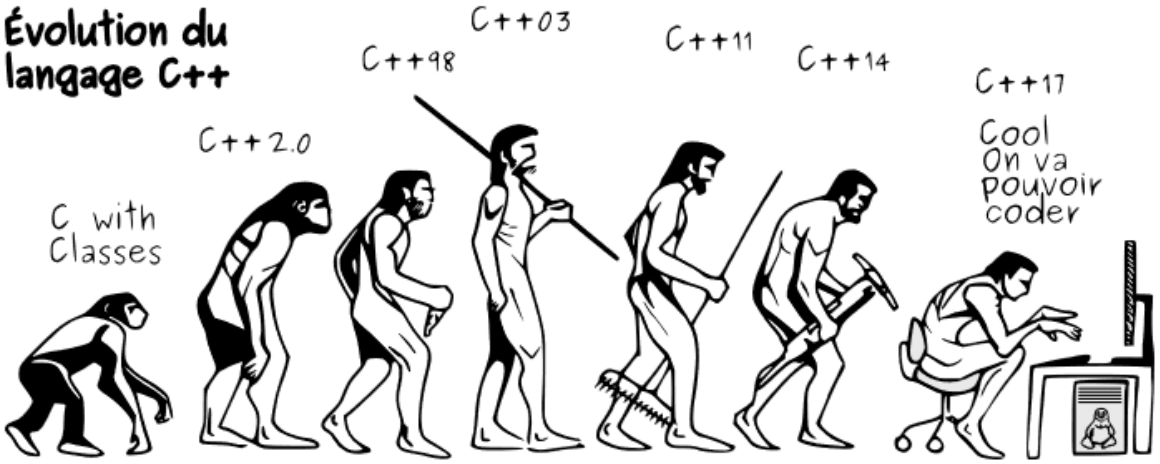
P. ELYAKIME
pierre.elyakime@imft.fr

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 »



Vue générale de la STL

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

→ L'héritage n'est que très faiblement utilisé

Et puissant :

Tableaux extensibles, Listes chainées

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

- **Principaux concepts mis en œuvre par la STL**
  - **Les conteneurs** : objets pour contenir d'autres objets (vector, list, map, set, queue, stack, …)
  - **Les itérateurs** : une abstraction des pointeurs pour parcourir les conteneurs et accéder aux données
  - **Les algorithmes** : des méthodes appliquées aux conteneurs pour manipuler les données (sort, find, …)
- Une classe **string** permettant de gérer de manière sûre les chaînes de caractères

Les 4 types de conteneurs

- **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 conteneurs séquentiels comme vector, deque ou list
  - **stack** : piles
  - **queue** et **priority_queue** : files d'attentes

Les 4 types de conteneurs

- **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

Les itérateurs

Les itérateurs

- Objet défini dans l'entête `# include <iterator>` qui généralise la notion de pointeur.
- Sert à pointer vers un élément dans une plage d'éléments (tableau,conteneur) grâce à des opérateurs et des fonctions membres qui **permettent de parcourir les conteneurs**
- Syntaxe d'appel général d'un itérateur

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

Exemple :

```
std::vector<int> it;
std::vector<int>::iterator it;
```

Les itérateurs

→ **Pratique** : permet de se déplacer dans un conteneur sans connaitre 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

Ancienne approche / approche STL

L'ancienne approche :

```
using namespace std;

vector<int> vec;
// Add some elements to vec
vec.push_back(1);
vec.push_back(4);
vec.push_back(8);

for(int y=0; y<vec.size(); y++)
{
    cout<<vec[y]<<" ";
    //Should output 1 4 8
}
```

L'approche STL :

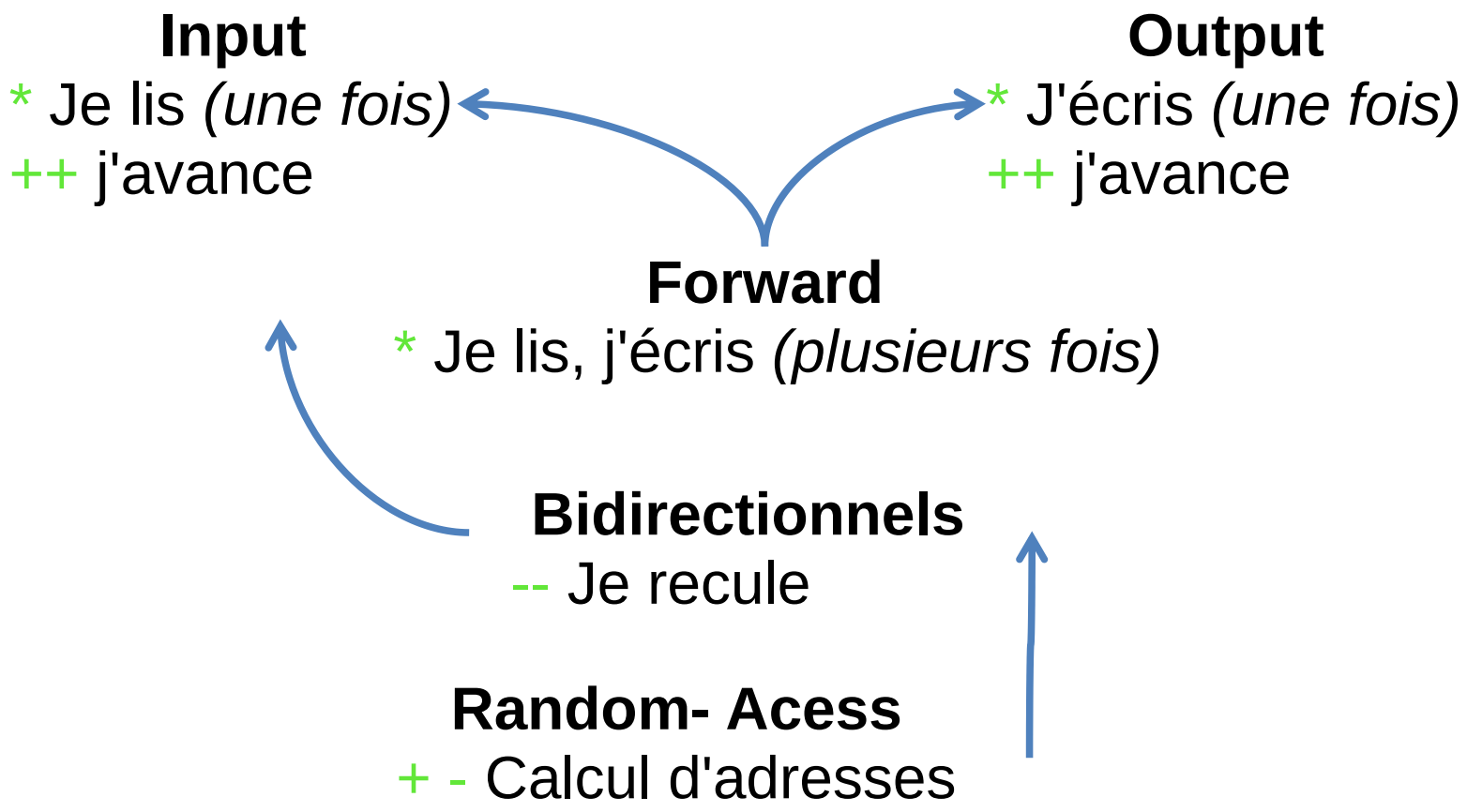
```
using namespace std;

vector<int> vec;
vector<int>::iterator it;

// Add some elements to vec
vec.push_back(1);
vec.push_back(4);
vec.push_back(8);

for(it=vec.begin();it!= vec.end(); it++)
{
    cout<<*i<<" ";
    //Should output 1 4 8
}
```

5 catégories d'itérateurs



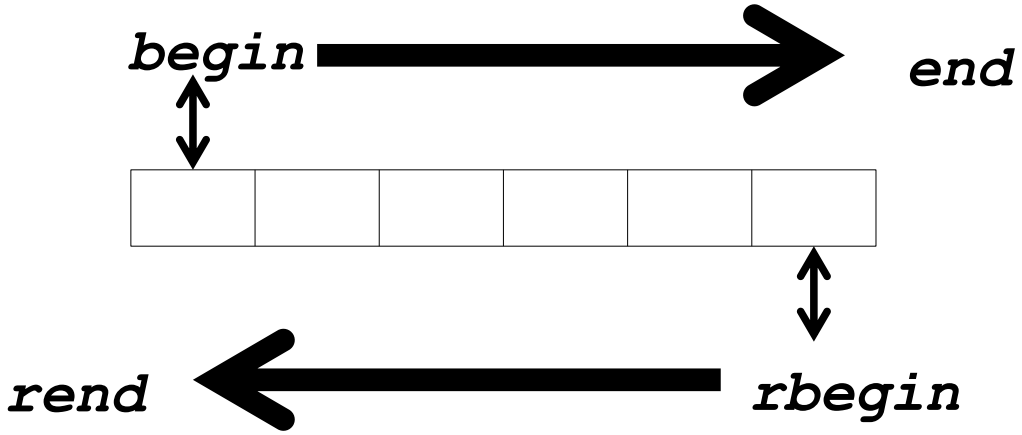
Qui peut le plus peut le moins

A chaque conteneur son itérateur

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

Plusieurs types d'itérateurs

```
container<T> :: iterator : it                    (read/write)
container<T> :: reverse_iterator : it            (Bidirectionnel et +)
container<T> :: const_iterator : it              (read-only)
container<T> :: const_reverse_iterator : it
```



Fonctions membres de déplacements :

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

Intervalle : [Itérateur1, Itérateur2 [

Exemple : balayer un conteneur

```
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 << " " ;}
```

Opérateurs et fonctions membres

Opérateurs d'incréments et de déréférencements:

`*(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

Fonctions 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 recul/avance de n

Exemple : distance

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

int main()
{
    std::vector<int> v{ 3, 1, 4 };
    std::cout << "distance(first, last) = "
              << std::distance(v.begin(), v.end()) << '\n'
              << "distance(last, first) = "
              << std::distance(v.end(), v.begin()) << '\n';
               //the behavior is undefined (until C++11)

}
```

Exemple : distance

```
#include <iostream>
#include <iterator>
#include <vect

int main()
{
    std::vecto
    std::cout <
              << std::distance(v.begin(), v.end()) << '\n'
              << "distance(last, first) = "
              << std::distance(v.end(), v.begin()) << '\n';
               //the behavior is undefined (until C++11)

}
```

Output :
distance(first, last) = 3
distance(last, first) = -3

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

Les conteneurs

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

Fonctions membres

Fonctions membres :

→ dépend des conteneurs (voir plus loin)

Fonctions annexes des membres :

!= , ==

et pour les ordonnés

<, <=, >, >=

Les 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 :

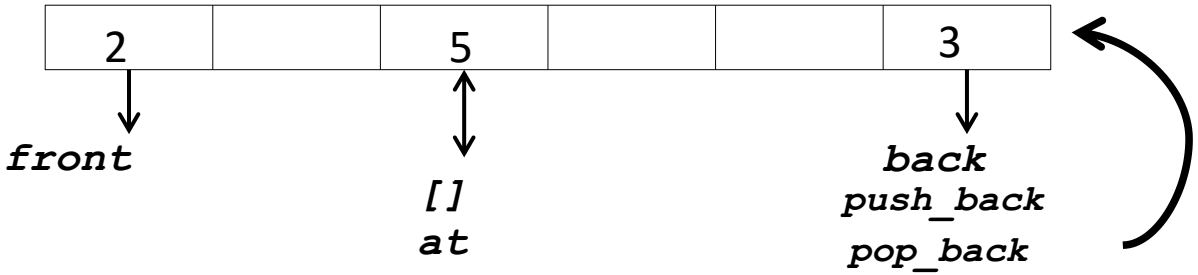
++ Accès en O(1)

++ 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;




```
std::array<int> a=(6, 2); // création d'un array de taille 6 remplit 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)
- Stockage en mémoire contigue
- Accès rapide aux éléments du tableau => **itérateurs Random Access**

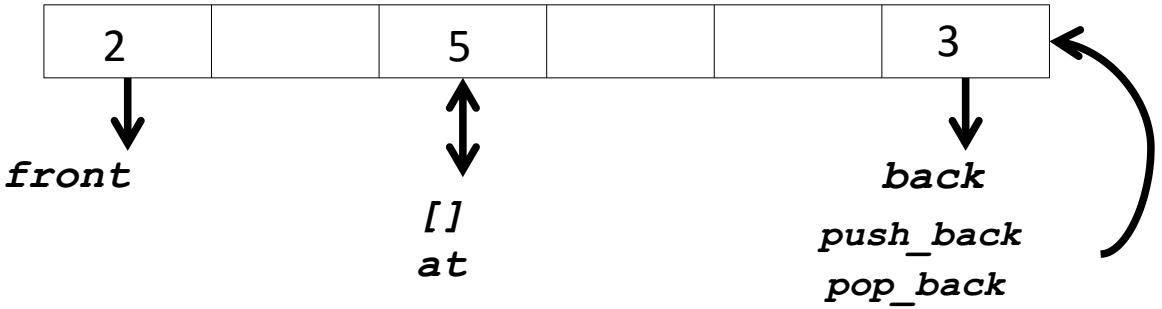
- *Complexité :*
  - ++ Accès O(1)
  - ++ Insertion et suppression en O(1) en fin de vector (push/pop_back)
  - -- Insertion et suppression en O(n) en début de vector

⇒ 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;




```
vector<int> v(6, 2); // création d'un vector de taille 6 remplit 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 namespac

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

  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;
}
```

Output :
Le premier élément est 1
Le dernier élément est 3
Le dernier élément est maintenant 2

STD :: DEQUE<T>

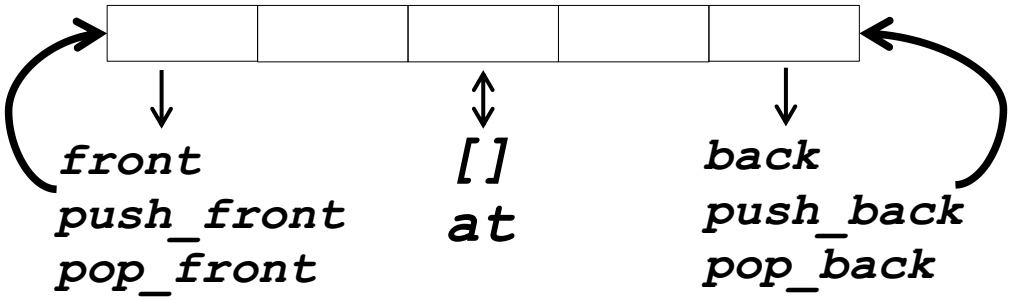
- Double Ended QUEue = queue à deux bouts

  ➔ Tableau à double entrée avec insertion et suppression rapide aux extrémités (fin,début)
- Stockage non contiguë des éléments (≠ 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;




```
std::deque<int> d(6, 2); // création d'un deque de taille 6 remplit 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::s

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;
}
```

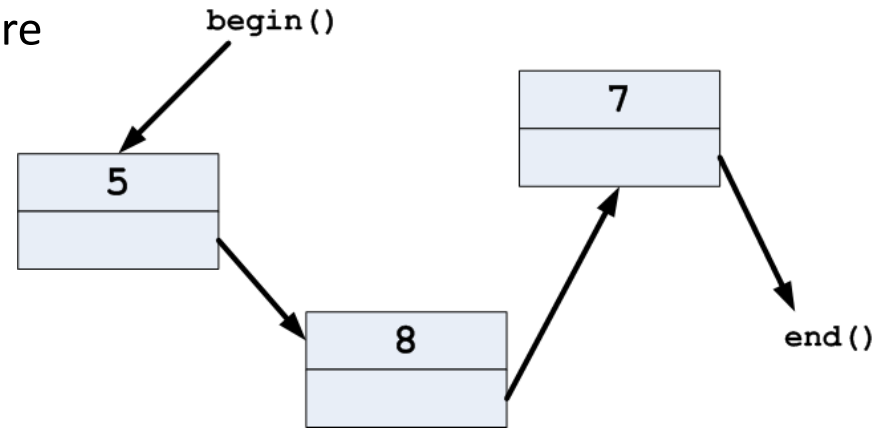
Output :
Le premier élément est 20
Le dernier élément est 15
Le premier élément est 14
Le dernier élément est 10

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

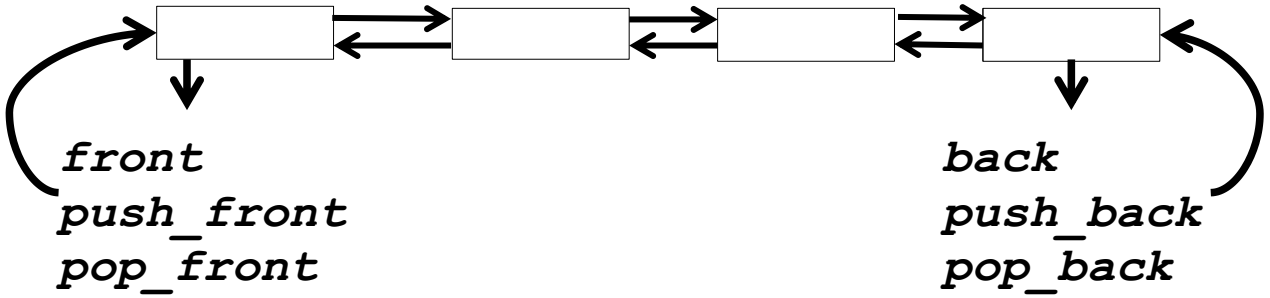
++ Insertion ou suppression en O(1)

++ Tri (avec sort) en O(n.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> lst(6) // création d'une liste de taille 6

lst.front(); // accès au 1ier élément
lst.push_front(); // insère des éléments au début
lst.pop_front(); // supprime le 1ier element


lst.back();  // accès au dernier element
lst.push_back(); // insère des éléments à la fin
lst.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 le7

  // 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 l
  lst.push_back(5); 
  lst.pop_back(); // enleve le dernier element et supprime le7

  // 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/Copie

- Liste d'initialisation depuis 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'element
- size();     // donne la taille du tableau
- empty();    //  verifie 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 spécifiques à LIST<T>

```
merge();      // fusionne deux listes
splice();     // déplace les elements d'une autre liste
remove(); remove_if(); // supprime des elements
reverse();    // inverse l'ordre des elements
unique();     // supprime les doublons successifs
sort();       // trie les elements en n.log(n)
```

Les conteneurs adaptateurs

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

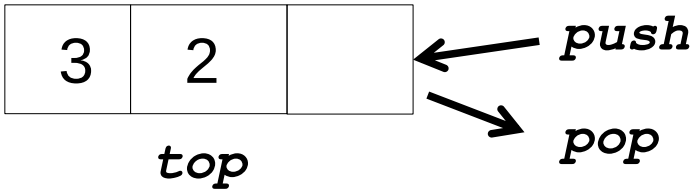
Classes patrons construites à partir des conteneurs ***vector***, ***deque*** ou ***list*** et qui modifient leur interface en les restreignant et en les adaptant à des fonctionnalitées données

La pile (stack)

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

```
stack<T> pile;              => avec deque
stack<T, vector<T>> pile;   => avec vector
stack<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, vecto
          cout << "taille i
          for (int i=0; i<1
                    pile.pus
          cout << "taille a
          cout << "sommet de
          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;
}
```

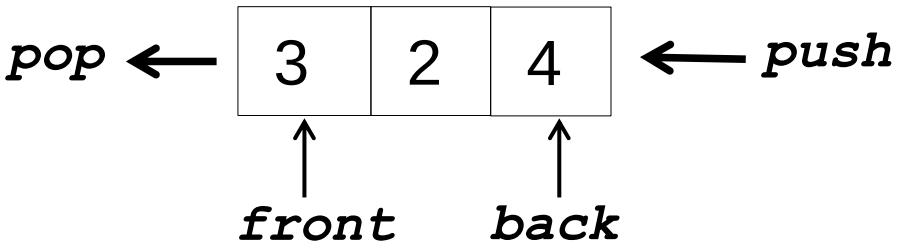
Output :
taille initiale : 0
taille après for : 10
sommet de la pile : 81
on dépile : 99 64 49 36 25 16 9 4 1 0
Taille de la pile : 0

La file (queue)

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

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

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



itérateur **Aucun**

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

file.pop();     // Retire l'element situe en tete de file
file.push();    // Ajoute un element 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;
}
```

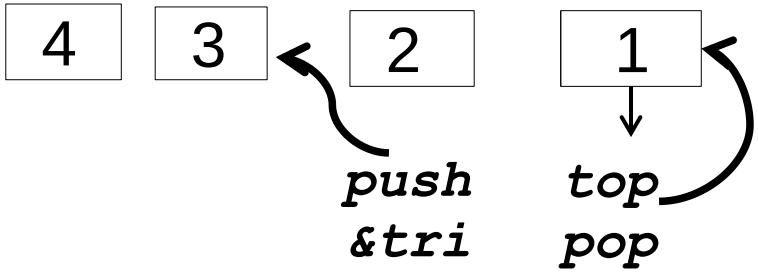
Output :
Taille de la file : 1
4

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;  => 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 : à chaque introduction, ils sont classés tel que l'élément du haut respecte la relation d'ordre donnée. Accès uniquement à l'élt supérieur.



itérateur **Aucun**

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

pq.top();   // Retourne l'element avec la priorité la plus haute
pq.pop();   // Supprime l'element avec la priorité la plus haute
```

Exemple

```
#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;
}
```

Exemple

```
#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;

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;
}
```

Output : 1 2 3

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 ré 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

et construit à partir du patron de classe : `T pair<T1,T2> paire_name`

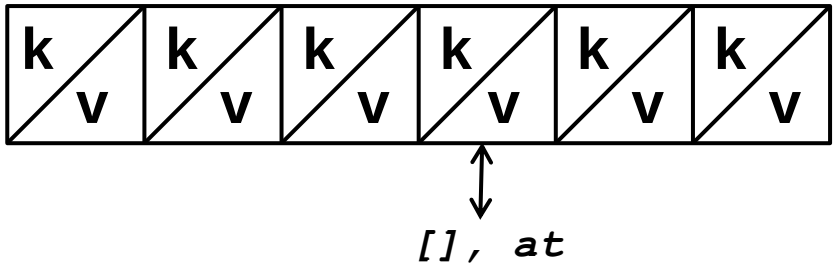
template <class key, class T, class Compare=less<Keys> > class map/multimap;

- **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 (peut être modifié)
- Itérateur **Bidirectionnel** (opère sur les **paires**) => clés **const**
- Accès rapide à la valeur associée à clé en O(log(n))

map/multimap



Trié par ordre croissant des klés

```
map<char, int> m;          // Déclaration
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::iterateur it;         // Déclaration de l'itérateur
*it;                       // represente l'element
it->first; it->second;     // acces à la cle (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 klé
```

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_bound(kle);     // fournit un it. sur le 1ier 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 namespa

template<type
void printCon
cont, const s
{
  cout << mes
  typename Co
it=cont.begin();
  for(; it!=cont.end();it++) {
     cout << "(" << it->first << ": "
<< it->second << ") ";
  }
  cout << endl;
}
```

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

  m1.merge(m2);

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

}
```

Output :
m1 : (earth: 1) (moon: 2)
m2 : (moon: 2) (sun: 3)
m1 apres merge : (earth: 1) (moon: 2) (sun: 3)

set/multiset

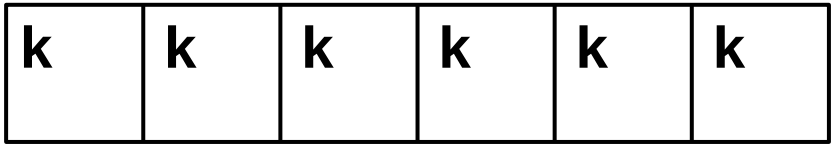
- Cas particulier de map = ensemble de clés (ce ne sont plus des paires cle/val)

  → 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
- Itérateur **Bidirectionnel,** (opère sur les **paires**) => clés **const**
- Accès rapide à la valeur associée à clé en O(log(n))

set/multiset



Trié par ordre croissant des klés

```
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

La classe string

string

- Simplifie les tableaux de chaines 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];
```

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");

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

}
```

Exemple

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

```
    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");

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

}
```

Fonctions de manipulation

```
+=, append(), push_back() // concaténation, ajout à la fin de la chaine
insert()        // insere une sous-chaine dans une chaine
erase()         // supprime une sous-chaine dans une chaine
replace()       // remplace une partie d'une chaine par une sous-chaine
pop_back()      // efface le dernier caractere

size(s), length(s) // retourne la taille de s
reserve()       // reserve de la mémoire
clear()         // supprime toute la chaîne
empty()         // retourne true si la sous chaine 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

```
string::size_type // définit le type  d'un string non signé (=0
et positif) assez grand pour représenter n'importe quelle taille en
mémoire, dépend de l'architecture de la machine (unsigned int ou
unsigned long int)
```

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

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;
}
```

Exemple - sort

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

using namespace std;

void sho
  for(in
    cout
}

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;
}
```

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

Exemple - copy

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

int main()
{
  int a[7]= {10, 20, 30, 40, 50, 60, 70};
  vector<int> vec(7);

  copy( a, a+7, vec.begin());

  cout << "my vector contains: ";
  for (vector<int>::iterator it=vec.begin(); it!=vec.end(); it++)
    cout << " " << *it;
  cout << " \n";
  return 0;
}
```

Exemple - copy

```
#include <iostream>
#include <algorithm>
#include <vector>
using names

int main()
{
  int a[7]=
  vector<int> vec(7);

  copy( a, a+7, vec.begin());

  cout << "my vector contains: ";
  for (vector<int>::iterator it=vec.begin(); it!=vec.end(); it++)
    cout << " " << *it;
  cout << " \n";
  return 0;
}
```

Output :
my vector contains: 10 20 30 40 50 60 70

Exemple - find

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector

int main () {
  int myints[] = { 10, 20, 30, 40 }; // using std::find with array and pointer:
  int * p;

  p = std::find (myints, myints+4, 30);
  if (p != myints+4)
    std::cout << "Element found in myints: " << *p << '\n';
  else
    std::cout << "Element not found in myints\n";

  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
}
```

Exemple - find

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector

int main () {
  int myints[] = { 10, 20, 30, 40 }; // using std::find with array and pointer:
  int * p;

  p = std::find
  if (p != myint
    std::cout <<
  else
    std::cout <<

  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
}
```

Output :
Element found in myints: 30
Element found in myvector: 30

<algorithm>: 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

<algorithm>: 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, …

<algorithm>: 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'unconteneur

<algorithm>: 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

<algorithm>: 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

<algorithm>: 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 donnée

```
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

<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, Output_It d_first);
```
→ somme partielle des valeurs d'un conteneur

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

Exemple - accumulate

```
#include <iostream>     // std::cout
#include <numeric>      // std::find
#include <vector>       // std::vector

 using namespace std;

int main () {
  vector<float> v;
  v.push_back (3.14); v.push_back (5);

  // ...
  float sum = accumulate(v.begin(), v.end(), 0.0);
  float produit = accumulate(v.begin(), v.end(), 1.0,
multiplies<float>());

  cout << "Le résultat de la somme est : " << sum << endl;
  cout << "Le résultat du produit est : " << produit << endl;

  return 0;
}
```

Exemple - accumulate

```
#include <iostream>      // std::cout
#include <numeric>       // std::find
#inclu

 usin

int m
  vec
  v.p

  // ...
  float sum = accumulate(v.begin(), v.end(), 0.0);
  float produit = accumulate(v.begin(), v.end(), 1.0,
multiplies<float>());

  cout << "Le résultat de la somme est : " << sum << endl;
  cout << "Le résultat du produit est : " << produit << endl;

  return 0;
}
```

Output :

Le résultat de la somme est : 8.14

Le résultat du produit est : 15.7

<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/

Depuis la norme C++11

Nouvelle syntaxe pour « for »

```
for ( loopVariable : collection ) statement;

for ( loopVariable : collection ) {
    [statement]...
}
```

-> Syntaxe de boucle for introduite avec la version **C++11** (-std=c++11)

-> Permet de réaliser une itération sur tous les éléments d'une collection, sans avoir à gérer de compteur de boucle.

-> Fonctionne sur toute collection itérable au sens de la STL : donc des deques, des vectors, des listes, ...

Exemple

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

int main()
{
        int arr[] = { 10, 20, 30, 40 };

        // Printing elements of an array using for loop
        for (int x : arr)
                cout << x << endl;
}
```

Exemple

```
// C++ program to demonstrate use of foreach
#include <i
using name

int main()
{
        int

        //
        // foreach loop
        for (int x : arr)
                cout << x << endl;
}
```

Output :
10
20
30
40

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 an integer

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 ;
}
```

Et bien d’autres

- constrexpr
- Lambda functions
- For_each
- Parallel algorithm
- …

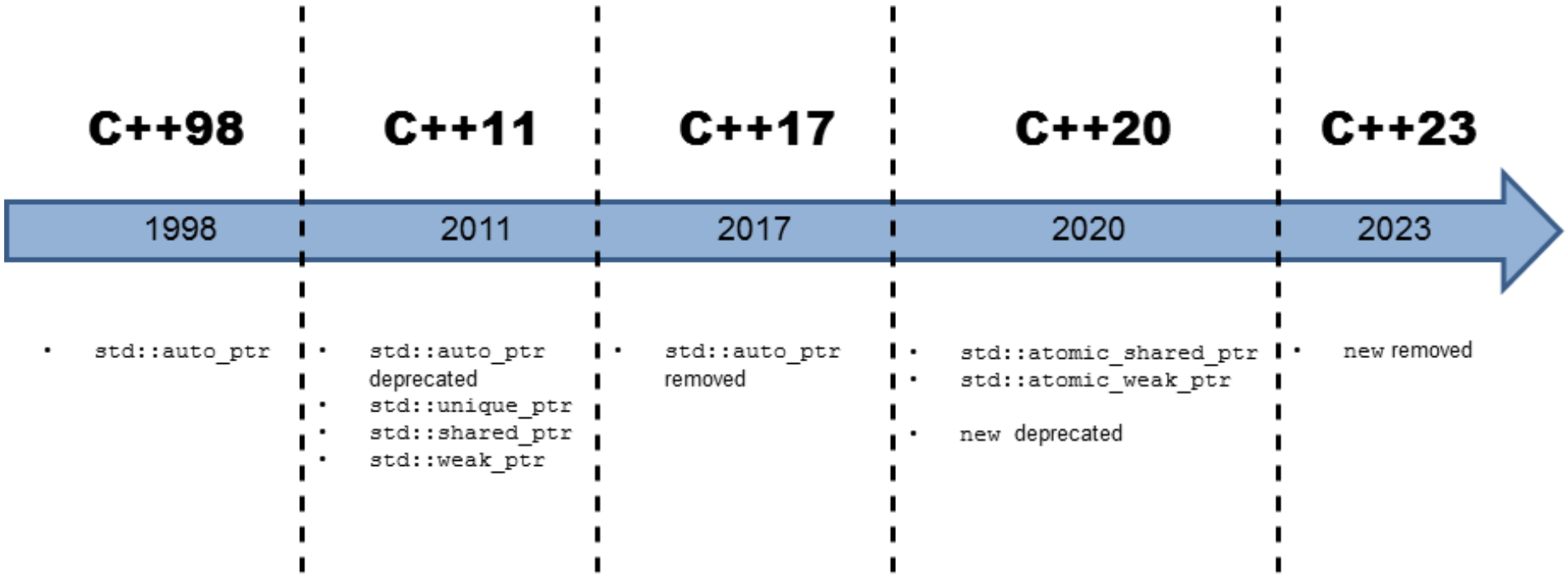
C++ avancé

- 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, …

Evolution of pointers in C++



Evolution of pointers in C++

- 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 : essaie de corriger 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)

RAII

- Basée sur la technique de programmation Resource Acquisition Is Initialisation (RAII) : consiste à lier une ressource à un objet (sur la pile) lors de son initialisation.
- La ressource est acquise dans le constructeur de l'objet et est libérée dans le destructeur
- A la sortie du scope de l'objet, le destructeur étant systématiquement appelé, la libération de la ressource est garantie

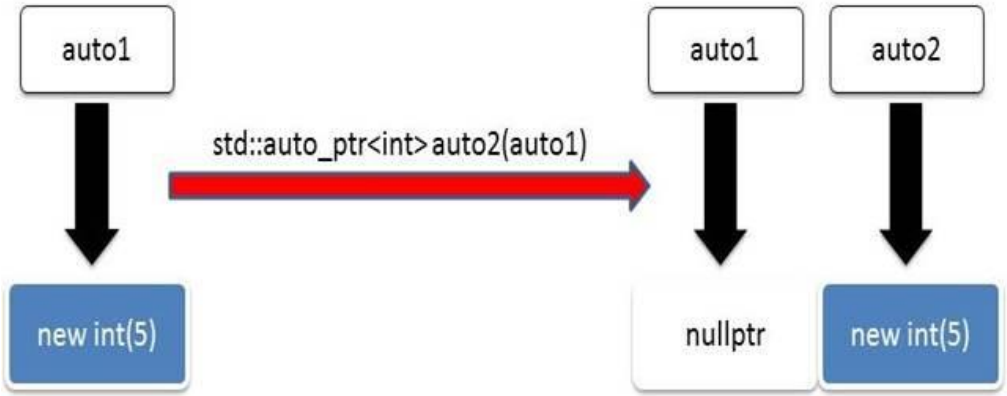
```
Class Manager
{
      public :
      Manager(MyObject *o) : obj_(o) {}
      ~Manager() {  delete o;   }
      private:
       MyObject *obj_;
} ;
```

```
void process()
{
  MyObject *myObject = new MyObject ();
  Manager myManager(myObject);
  // traitement
}
```

Smart pointers in C++

=> Libère automatiquement la mémoire allouée dynamiquement

En C++98 `std::auto_ptr()` déprécié depuis c++11 car présente un problème lors de la copie du pointeur (exprime la propriété exclusive), ne le copie pas mais le transfère laissant le 1ier vide :



Lors d'une copie la ressource est déplacée

Smart pointers in C++

Depuis C++11 :

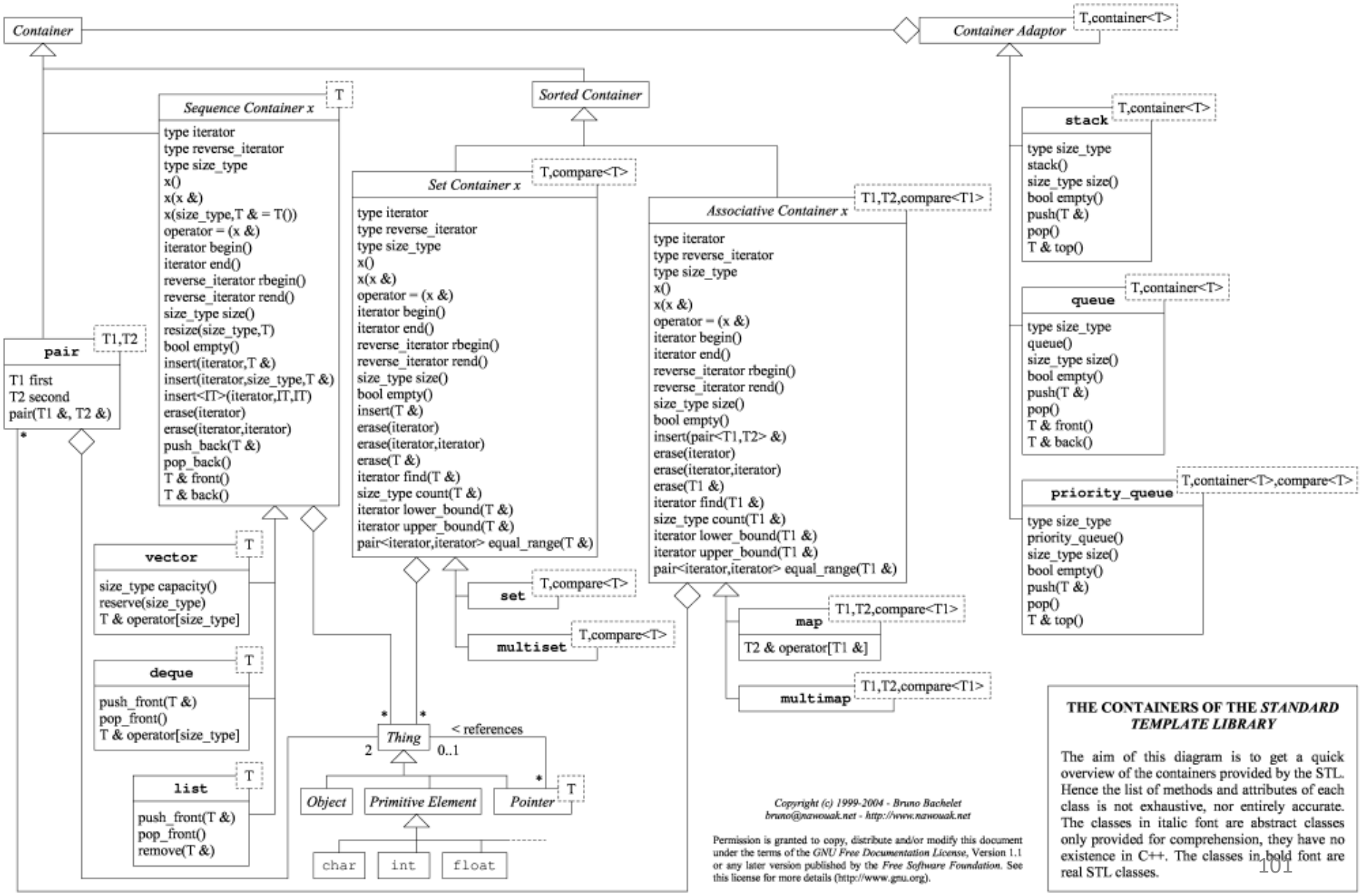
`std::unique_ptr()`: pas de copie possible, mais possible de le bouger avec move(). Met en avant la notion de propriété exclusive.

`std::shared_ptr()`: permet de partager la gestion de la mémoire allouée entre différentes instances. La mémoire est libérée lorsque la dernière instance est détruite. Met en avant la notion de propriété partagée.

`std::wake_ptr()`: a un fonctionnement particulier, s'utilise en complément du shared_ptr(). Permet de résoudre le problème des références circulaires rencontrées avec le shared_ptr().

Lien utile : https://www.invivoo.com/introduction-a-la-gestion-automatique-de-la-memoire-en-c11/

Diagramme UML de la STL



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 …

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