université de TOURS

Développement objet C++

Dominique H. Li

dominique.li@univ-tours.fr

Licence Informatique - Blois

Université de Tours

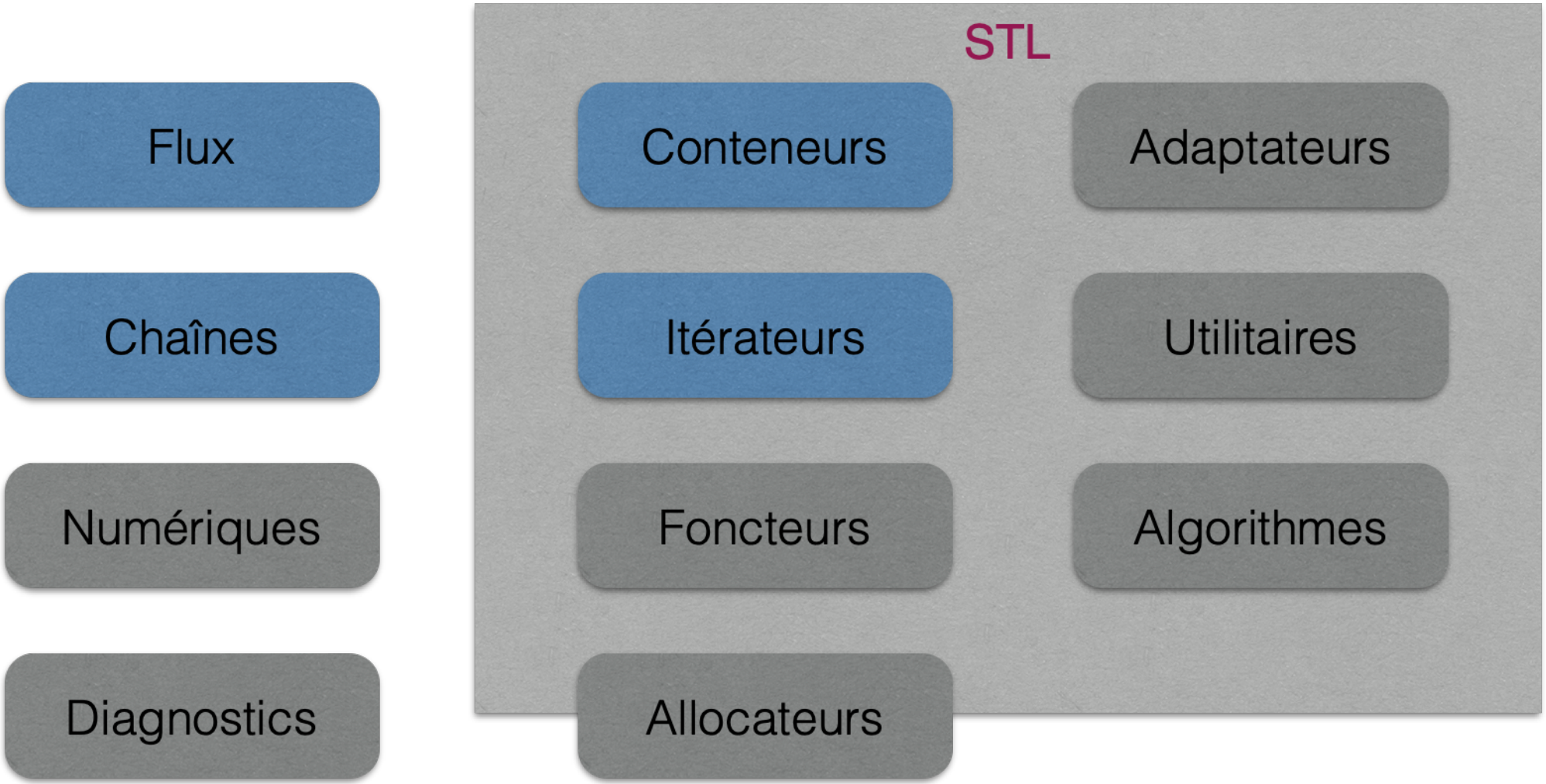
université de TOURS

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World !" << std::endl;
    return 0;
}
```

Les conteneurs STL

Développement objet C++

La bibliothèque standard de C++



La STL (Standard Template Library)

- La *bibliothèque standard de C++* fournit un cadre d'application *STL* (*Standard Template Library*) pour gérer les données sous forme de conteneurs
- La STL est une collection de types de données et d'algorithmes offrant des solutions pour une variété de problèmes
- La STL se base sur un concept appelé *programmation générique*
- La STL masque les difficultés inhérentes aux structures de données complexes et en permet une utilisation simplifiée

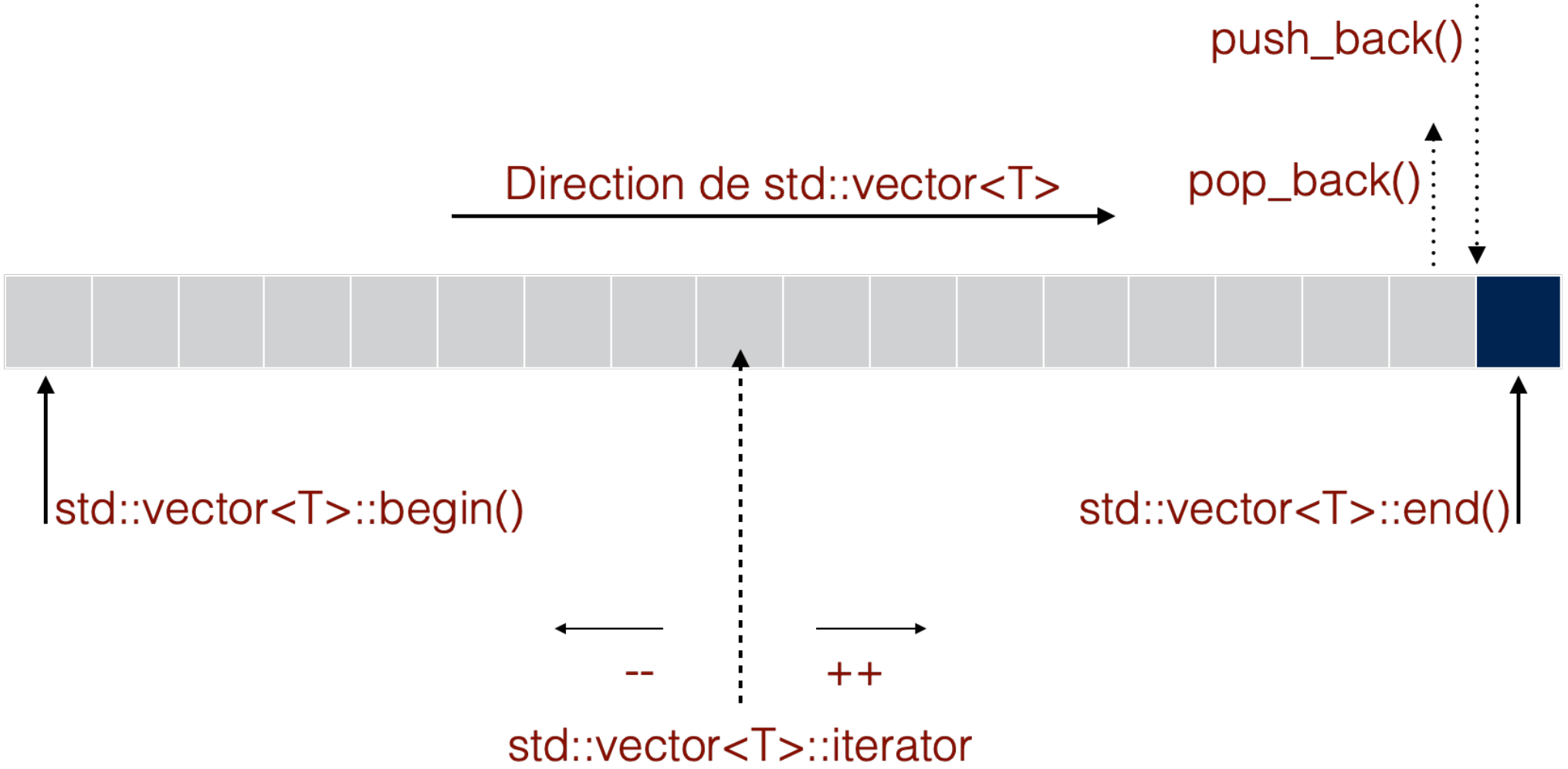
Les conteneurs STL

- Les *conteneurs* sont des objets contenant d'autres objets
- Les *conteneurs séquentiels*
  - Vecteur : **std::vector**
  - Liste : **std::list**
- Les *conteneurs associatifs*
  - Ensemble : **std::set** et **std::multiset**
  - Map : **std::map** et **std::multimap**

Le conteneur std::vector

- Vecteur standard (tableau dynamique)
- Caractéristiques
  - Accès direct rapide à chaque élément par une indice, comme dans les tableaux de C/C++
  - L'insertion et la suppression d'éléments se font en principe à la fin du vecteur en un temps quasi constant, sous réserve de problème de réallocation
- Fichier en-tête **<vector>**

L’itérateur std::vector<T>::iterator



Les méthodes usuelles de std::vector (1)

- Accès aux éléments

reference operator[](size_type n)

reference at(size_type n)

reference front()

reference back()

iterator begin() / reverse_iterator rbegin()

iterator end() / reverse_iterator rend()

Les méthodes usuelles de std::vector (2)

- Modificatifs

  void push_back(const value_type &val)

  void pop_back()

  iterator insert(iterator position, const value_type &val)

  iterator erase(iterator position)

Exemple : utilisation typique de std::vector

```
std::vector<int> v;

int x;
while (std::cin >> x) { v.push_back(x); }

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

std::vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it) { std::cout << *it << std::endl; }

it = v.begin();
while (it != v.end()) { std::cout << *it++ << std::endl; }
```

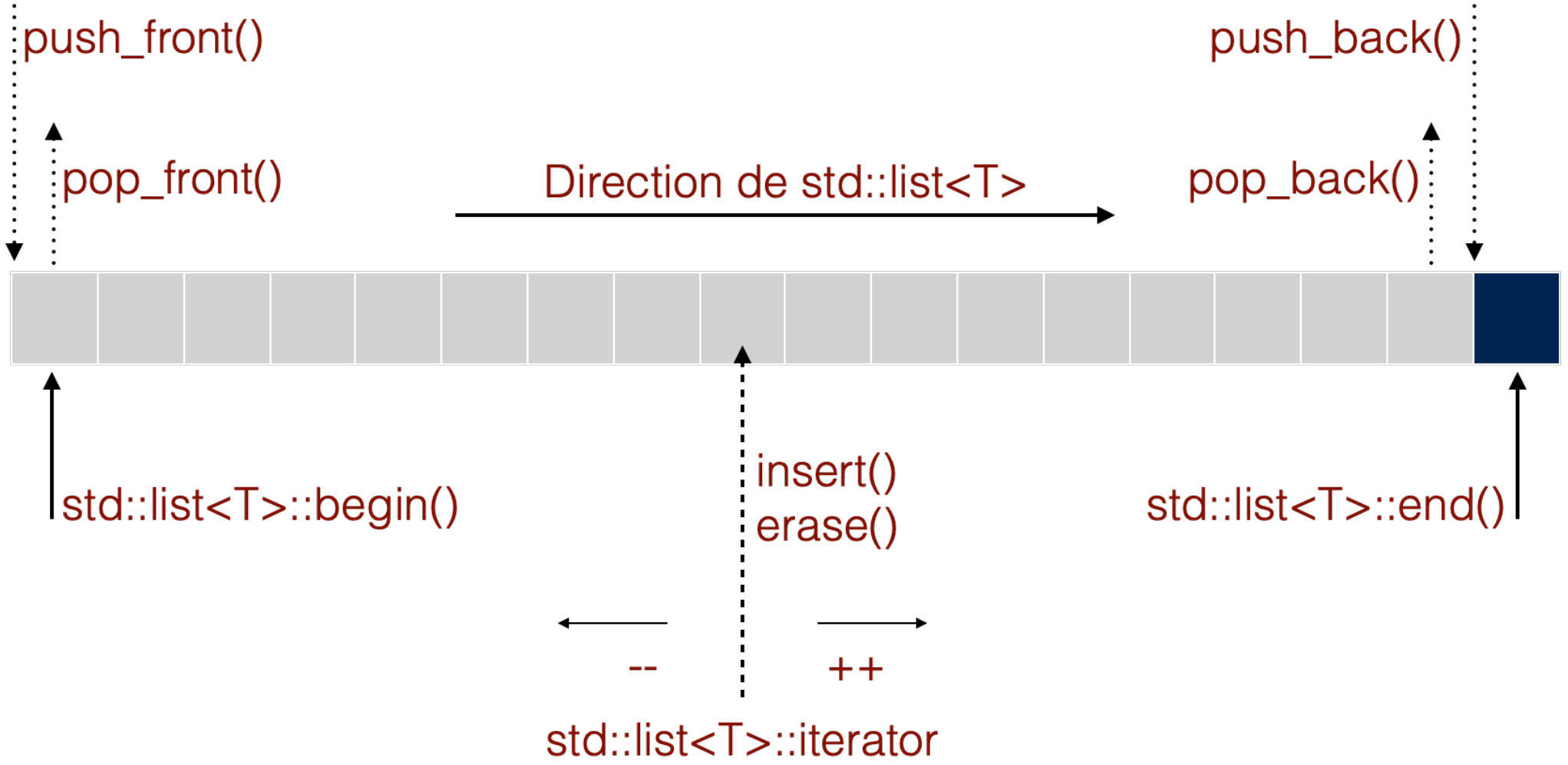
Exemple : utilisation typique de std::vector (C++11)

```
std::vector<int> v;

int x;

while (std::cin >> x) { v.emplace_back(x); }

for (auto &x: v) { std::cout << x << std::endl; }

for (auto it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << std::endl;
}
```

Le conteneur std::list

- Liste chaînée
- Caractéristiques
  - L'insertion et la suppression d'éléments à une quelconque position dans la liste s'effectuent en temps constant
  - Les listes peuvent être partagées, fusionnées et inversées
- Fichier en-tête **<list>**

L'itérateur std::list<T>::iterator



Les méthodes usuelles de std::list (1)

- Accès aux éléments

  reference front()

  reference back()

  iterator begin() / reverse_iterator rbegin()

  iterator end() / reverse_iterator rend()

Les méthodes usuelles de std::list (2)

- Modificatifs

  void push_front(const value_type &val)

  void pop_front()

  void push_back(const value_type &val)

  void pop_back()

  iterator insert(iterator position, const value_type &val)

  iterator erase(iterator position)

Les méthodes usuelles de std::list (3)

- Suppression d'éléments

  void remove(const value_type &val)

  void unique()

- Fusion de listes triées

  void merge(list& x)

Les méthodes usuelles de std::list (4)

- Tri d'éléments

  void sort()

- Inversion de l'ordre des éléments

  void reverse()

Exemple : utilisation typique de std::list

```
std::list<int> l;

int x;
while (std::cin >> x) {
        l.push_back(x);          // Ajouter x à la fin de la liste
        l.push_front(x);         // Ajouter x au début de la liste
}

std::list<int>::iterator it;
for (it = l.begin(); it != l.end(); ++it) { cout << *it << endl; }

l.pop_front();                   // Enlever la valeur au début de la liste
```

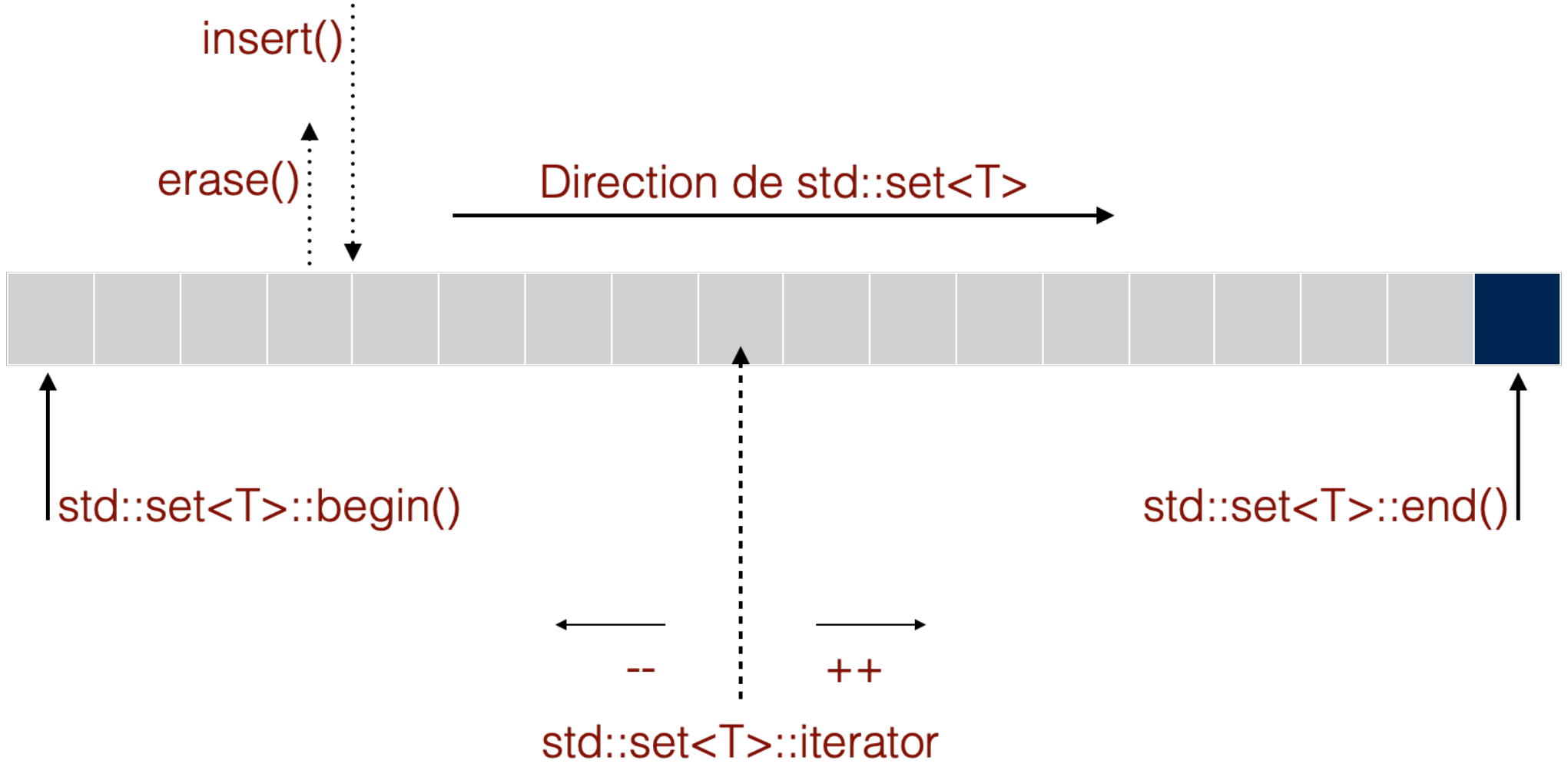
Exemple : utilisation typique de std::list (C++11)

```
std::list<int> l;

int x;
while (std::cin >> x) {
        l.emplace_back(x);      // Ajouter x à la fin de la liste
        l.push_front(x);        // Ajouter x au début de la liste
}

for (auto &x : l) { cout << x << endl; }

for (auto it = l.begin(); it != l.end(); ++it) { cout << *it << endl; }
```

Les conteneurs std::set et std::multiset

- Ensemble et ensemble multiple
- Caractéristiques
  - Contiennent seulement des clés
  - Dans le **std::set**, les clés doivent avoir des valeurs uniques
  - Dans le **std::multiset**, les clés peuvent avoir des valeurs non uniques
- Fichier en-tête **<set>**

L'itérateur std::set<T>::iterator



Exemple : utilisation typique de std::set

```
std::set<char> s;

s.insert('C');                  // Insérer un élément dans l'ensemble
s.insert('B');
s.insert('A');
s.insert('C’);

std::set<char>::iterator it = s.begin();
for (; it != s.end(); ++it) { std::cout << *it << std::endl; }

s.erase('C');                   // Supprimer un élément de l'ensemble
s.insert('F');
s.insert('F');
```

Exemple : utilisation typique de std::set (C++11)

```
std::set<char> s;

s.insert('C');                  // Insérer un élément dans l'ensemble
s.insert('B');
s.insert('A');
s.insert('C');

for (auto &x : s) { std::cout << x << std::endl; }

s.erase('C');                   // Supprimer un élément de l'ensemble
s.insert('F');
s.insert('F');
```

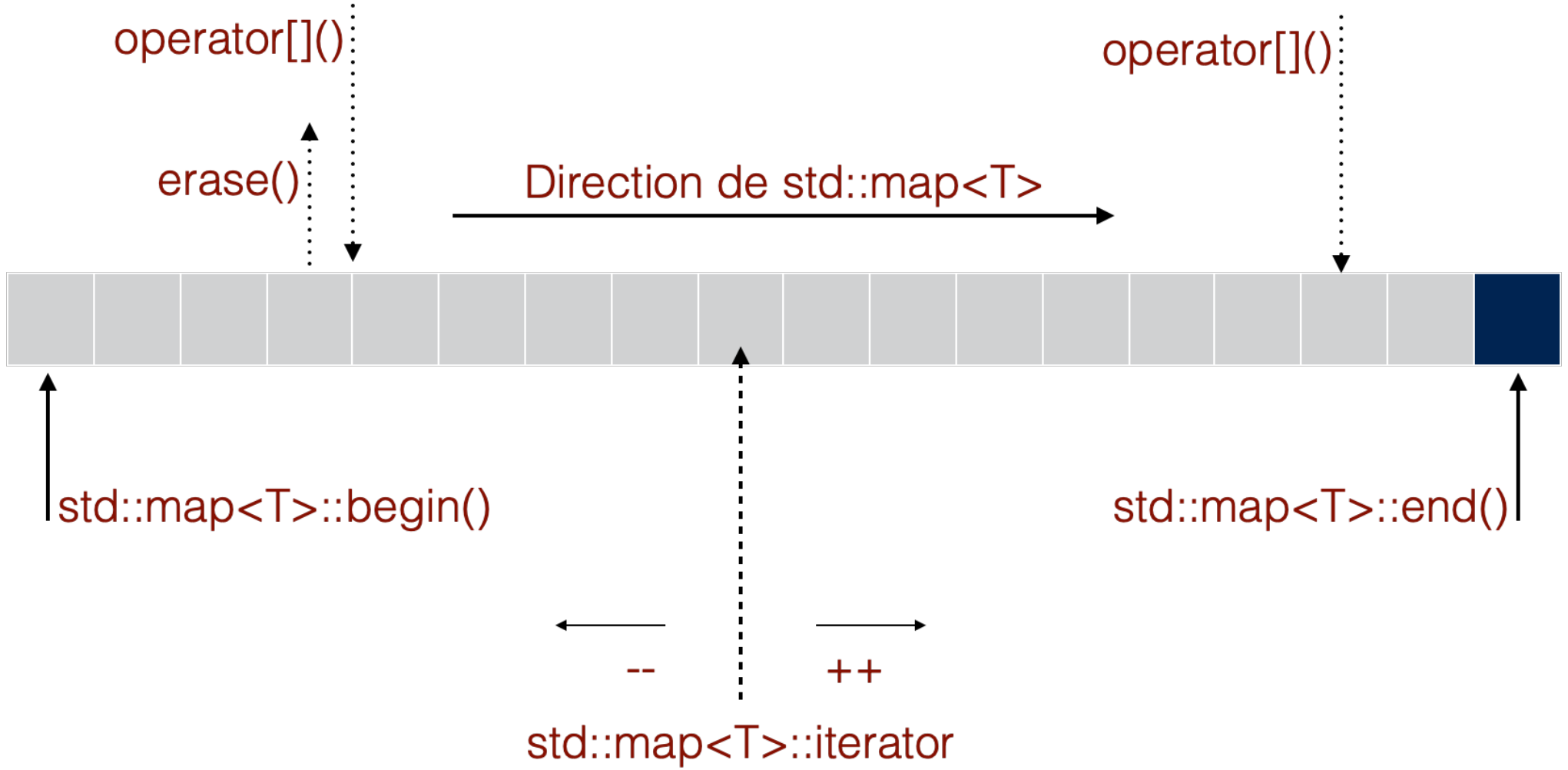
Exemaple : recherche d'éléments dans std::set

```
std::set<std::string> mots;       // Ajouter des mots

if (mots.count("hello"))
        std::cout << "Le mot 'hello' existe !" << std::endl;
else
        std::cout << "Le mot 'hello' n'existe pas !" << std::endl;

std::set<std::string>::iterator it = mots.find("hello");
if (it != mots.end())
        std::cout << "Le mot 'hello' existe !" << std::endl;
else
        std::cout << "Le mot 'hello' n'existe pas !" << std::endl;

if (mots.find("bonjour") != mots.end())

        std::cout << "Le mot 'bonjour' existe !" << std::endl;
```

Les conteneurs std::map et std::multimap

- Tableau associatif et tableau associatif multiple
- Caractéristiques
  - Contient des paires *<clef, valeur>*
  - Dans le **std::map**, chaque clé doit être unique et ne peut référencer qu'une seule valeur
  - Dans le **std::multimap**, les clés ne sont pas uniques, chaque clé peut donc référencer plusieurs valeurs
- Fichier en-tête **<map>**

L'itérateur std::map<T>::iterator



Exemple : utilisation typique de std::map

```
std::map<char, std::string> m;

m['A'] = "Alpha";
m['T'] = "Tango";
m['T'] = "Tank";

std::map<char, std::string>::iterator it = m.begin();
for (; it != m.end(); it++)
        std::cout << it->first << " : " << it->second << std::endl;

m.erase('T’);

if (m.find('T') != m.end())
        std::cout << m['T'] << std::endl;
```

Exemple : utilisation typique de std::map (C++11)

```
std::map<char, std::string> m;

m['A'] = "Alpha";
m['T'] = "Tango";
m['T'] = "Tank";

for (auto &it: m)
        std::cout << it.first << " : " << it.second << std::endl;

m.erase('T');

if (m.find('T') != m.end())
        std::cout << m['T'] << std::endl;
```

Exemaple : recherche d'éléments dans std::map

```
std::string mot;
std::map<std::string, int> mots;

std::cin >> mot;
++mots[mot];

std::cin >> mot;

if (mots.count(mot))
        std::cout << mot << " : " << ++mots[mot] << std::endl;
else
        mots[mot] = 1;

std::map<std::string, int>::iterator it = mots.find(mot);
if (it != mots.end())
        std::cout << it->first << " : " << it->second << std::endl;
else
        mots.insert(std::pair<std::string, int>(mot, 1));
```

En savoir plus sur les conteneurs associatifs

- Le temps d'accès est en relation logarithmique avec le nombre d'éléments (implémentés par des arbres balancés)
- Ne connaissent pas l'opérateur **==**
  - Toutes les opérations de comparaison sont effectuées sur la base de l'opérateur **<**
  - Ainsi **a == b** est exprimé par **!(a < b) && !(b < a)**
- Sont parcourus par leurs itérateurs en ordre croissant, parce qu'ils ne connaissent que l'opérateur **<**