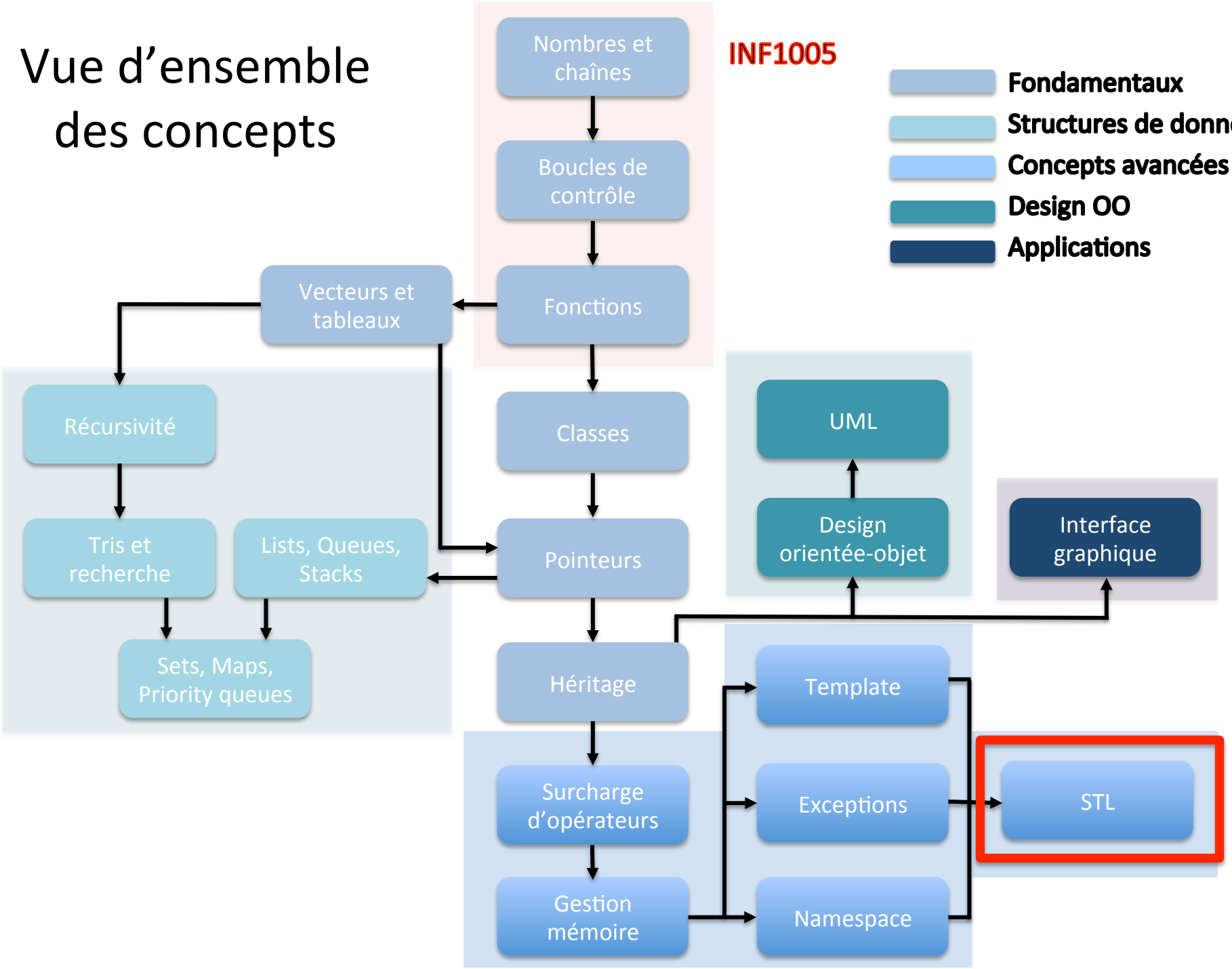
Programmation orientée objet

Itérateurs STL

Vue d'ensemble des concepts



Aperçu

- Introduction
- Intervalles d'itérateurs
- Déclaration des itérateurs
- Catégories
  - D'entrée
  - De sortie
  - Avançant
  - Bidirectionnel
  - À accès aléatoire
- Itérateurs constants
- Introducteurs
- Itérateurs renversés
- Résumé

Introduction

- Les itérateurs sont une généralisation des pointeurs.
- Ils sont utilisés pour accéder aux données du conteneur.
- Élément très important de la STL : ils permettent de **séparer conteneurs et algorithmes** pour plus de flexibilité.
- Un algorithme peut manipuler plusieurs types de conteneurs à travers leurs itérateurs.

Les intervalles d'itérateurs

- On peut représenter un intervalle d'éléments dans un conteneur par deux itérateurs : *début* et *fin*. Les éléments inclus sont **[*début*, *fin*[**.
- Il est important que *fin* ne soit pas inclus dans l'intervalle
  - Cela permet de retourner une valeur pour indiquer qu'une recherche est infructueuse.
  - Ou de représenter un intervalle vide (lorsque *début* et *fin* sont équivalents).
- Les méthodes `begin()` et `end()` renvoient des itérateurs qui couvrent tout l'intervalle d'éléments que possèdent un conteneur.

Déclaration des itérateurs

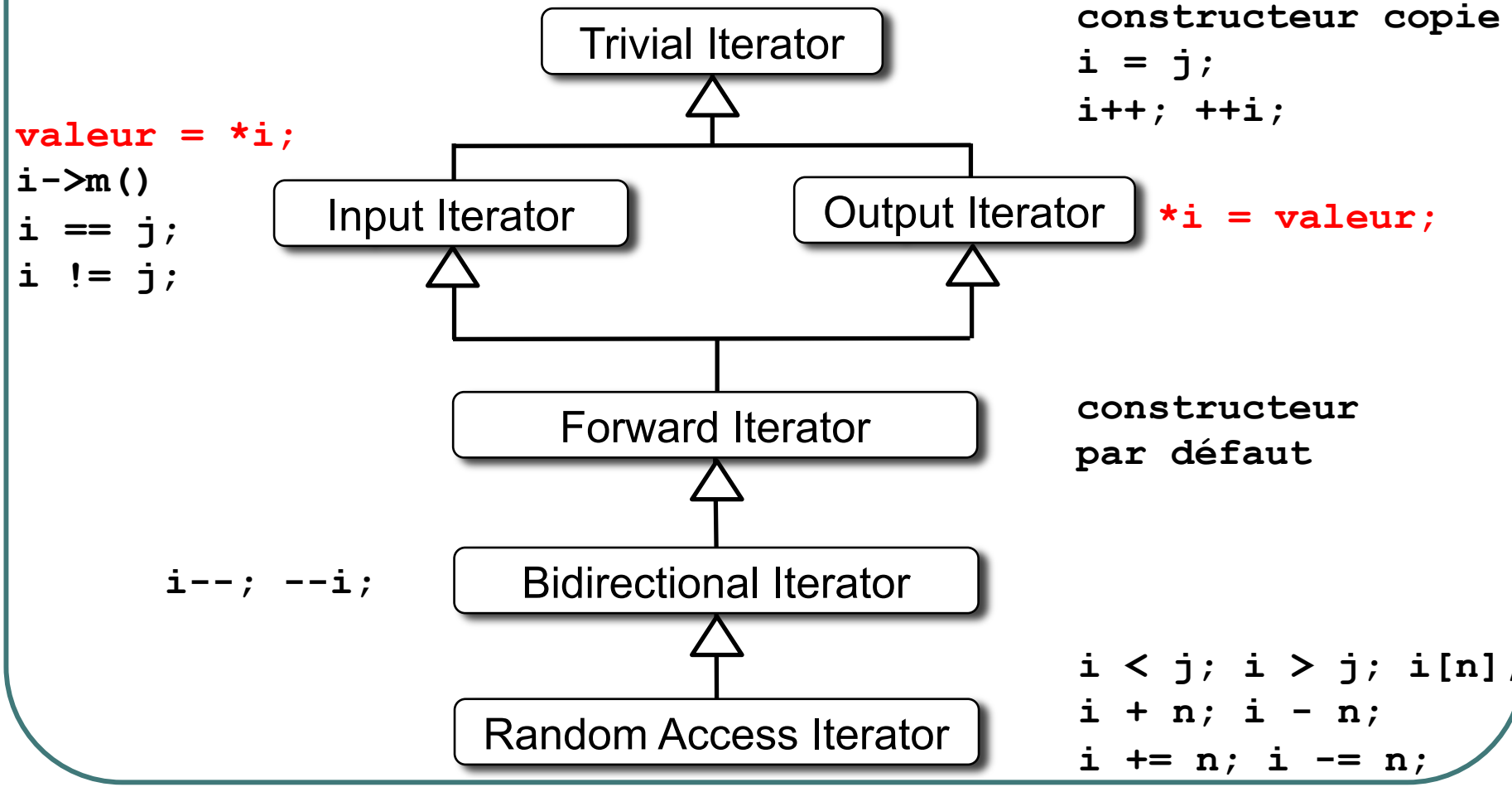
- Les itérateurs de la STL sont définis à l'intérieur des classes des conteneurs

```
template <typename T>
class vector
{
public :
      class iterator
      {
            ...
      };
      ...
};
```

- On déclare toujours un itérateur en faisant référence au type du conteneur qu'il pointe :

```
vector<double>::iterator it;
```

Catégories d'itérateur



Itérateur d'entrée (input)

- La principale utilité de cet itérateur est d'être déréférençable pour que l'on puisse copier la valeur pointée :

```
x = *it;      // opérateur unaire *
```

- On doit aussi pouvoir l'incrémenter pour qu'il pointe sur la prochaine valeur :

```
++it;         // opérateur unaire ++
```

- Il n'est **pas nécessairement mutable** (accepte que l'on modifie la valeur pointée)
- Exemple : `istream_iterator` (manipule des objets de type `istream` comme `cin`)

```
vector<int> V;

copy(istream_iterator<int>(cin), istream_iterator<int>(),
back_inserter(V));
```

Itérateur de sortie (output)

- À l'inverse de l'itérateur d'entrée, il permet d'emmagasiner des données par l'opérateur d'assignation :

```
*it = x;
*it++ = x;        // équivalent à *it = x; ++it;
```

- On ne **peut pas toujours le déréférencer ni tester** s'il est équivalent à un autre itérateur.
- Il n'y pas de type de valeur ni de distance associés.
- Exemple : `ostream_iterator` (manipule des objets de type `ostream` comme `cout`).

```
vector<int> V;
// ...
copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
```

Itérateur avançant (forward)

- Il permet **toutes les opérations que supportent les itérateurs d'entrée et de sortie** :
  - Déréférencement.
  - Modification de la valeur pointée (mutable).
  - Incrémentation de l'itérateur pour pointer vers la prochaine valeur.
  - Test d'égalité entre pointeurs et entre valeurs pointées.
- De plus, il **peut repasser plusieurs fois sur le même intervalle** (impossible avec les itérateurs d'entrée et de sortie).
- Exemple : tous les itérateurs associés aux conteneurs de la STL et les pointeurs arithmétiques.

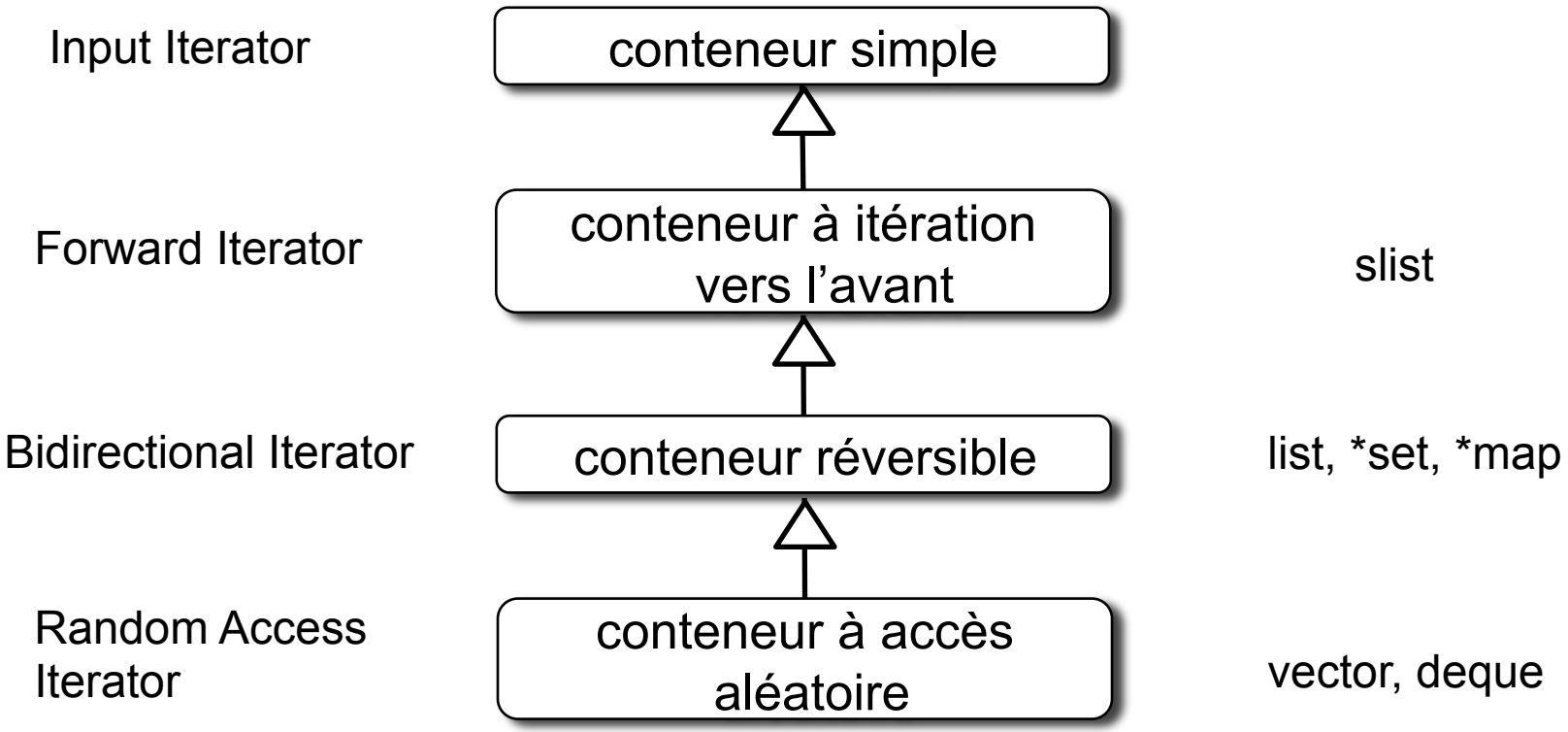
Itérateur bidirectionnel

- Encore plus polyvalent que l'itérateur avançant.
- Il est possible de le **décrémenter avec l'opérateur--** pour le faire reculer dans l'intervalle :
  `--it;`
- Il ne peut pas avancer ou reculer par bonds de plusieurs positions.
- Exemple : tous les itérateurs associés à la plupart des conteneurs de la STL et les pointeurs arithmétiques.

Itérateur à accès aléatoire

- Encore plus polyvalent que l'itérateur bidirectionnel.
- Il permet toutes les opérations des pointeurs arithmétiques C :
  - Incrémentation : `++it;`
  - Décrémentation : `--it;`
  - Déréférencement : `x = *it;`
  - Assignation : `*it = x;`
  - Égalité : `it1 == it2;`
  - Saut : `it += 5;` ou `it = it - 2;`
  - Opérateur d'élément : `it[i] = x;`
- Exemple : Seulement les itérateurs associés aux conteneurs *`vector`* et *`deque`* ainsi que les pointeurs arithmétiques.

Itérateurs associés aux conteneurs de la STL



Itérateurs associés au conteneurs de la STL

- iterator
- const_iterator
- reverse_iterator
- const_reverse_iterator
- InputIterator

Itérateur constant

- Le type `const_iterator`, qui n'est pas mutable, est fourni pour tous les conteneurs :

```
vector<Pion*> pions;
vector<Pion*>::const_iterator cIt;
/* ... */
for(cIt = pions.begin(), cIt != pions.end(); ++cIt)
      cIt->avancer(); // avancer d'une case
```

Refusé à la compilation
On ne peut pas appeler une méthode non const

- Seul un `const_iterator` peut manipuler un attribut dans une méthode déclarée const :

```
int Equipe::getNbMembresPresents() const
{
      int nbMembres = 0;
      vector<Membre*>::iterator it = membres_.begin();
      for(it; it != membres_.end(); ++it)
            if( it->estPresent() )
                  nbMembres++;

      return nbMembres;
}
```

Refusé à la compilation, car ce n'est pas un const_iterator

Introducteurs (ou itérateurs d'insertion)

- Ils ne forment pas une catégorie d'itérateurs, mais sont plutôt des adaptateurs d'itérateurs.
- Ils permettent d'insérer des éléments dans un conteneur sans écraser ceux déjà présents.
- On peut les classer dans la catégorie des itérateurs de sortie.
- La STL en définie trois types :
  - `insert_iterator`
  - `back_insert_iterator`
  - `front_insert_ieterator`

Introducteurs ou itérateurs d'insertion (suite)

- `inserter`(conteneur, itérateur) : crée un `insert_iterator` qui est utile pour insérer des éléments à un endroit précis dans un conteneur séquentiel.

```
insert_iterator<list<int> > i_it(L, it);
*i_it = x;     ⇔(équivalent à)  L.insert(it, x);
```

- `back_inserter`(conteneur) : crée un `back_insert_iterator` qui insert les éléments directement à la fin du conteneur

```
back_insert_iterator<list<int> > bi_it(L);
*bi_it = x;    ⇔ (équivalent à)  L.push_back(x);
```

- `front_inserter`(conteneur) : crée un `front_insert_iterator` qui insert les éléments directement au début du conteneur

```
front_insert_iterator<list<int> > ii(L);
*fi_it = x;    ⇔ (équivalent à)  L.push_front(x)
```

Itérateurs renversés

- Ce sont aussi des adaptateurs d'itérateurs.
- Deux types :
  - `reverse_iterator` : construit à partir d'un itérateur à accès aléatoire.
  - `reverse_bidirectional_iterator` : construit à partir d'un itérateur bidirectionnel.
- L'opérateur ++ a sur eux le même effet que l'opérateur -- sur les itérateurs normaux.
- Les conteneurs ont des méthodes pour renvoyer des itérateurs renversés:
  - `rbegin()` : pointe sur le dernier élément.
  - `rend()` : pointe passé le premier élément (avant le premier).

Résumé

- Les conteneurs `vector` et `deque` ont des itérateurs de catégorie itérateur à accès aléatoire et tous les autres conteneurs (sauf slist) de catégorie itérateur bidirectionnel.
- Les introducteurs insèrent les éléments assignés à des endroits précis dans le conteneur et sont de la catégorie itérateur de sortie.
- Les `ostream_iterator` et `istream_iterator` manipulent les objets de la librairie `<iostream>` pour les entrées et les sorties à la console.
- Les itérateurs renversés sont des adaptateurs d'itérateur qu'on utilise normalement, mais qui parcourent les conteneurs dans le sens inverse.
- Les `const_iterator` permettent d'accéder aux éléments d'un conteneur sans pouvoir les modifier (non mutable).