Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://claroline connect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

Standard Template Library (STL) : `http://www.cplusplus.com/reference/stl/`

La STL, c'est :

- les **classes de flots** (iostream, fstream) et la **classe string** ;
- des **conteneurs paramétrés selon le type d'élément** ;
- des **itérateurs** sur les conteneurs ;
- des **algorithmes** sur les conteneurs ;
- et des **prédicats**.

Vous connaissez bien les deux 1ères notions, on va donc regarder les nouvelles notions.

Introduction

La STL définit un ensemble de classes spécialisées dans le stockage d'éléments et appelées **conteneurs**, qui sont *paramétrées par le type des éléments à contenir* :

- les **vecteurs** (patron de tableaux dynamiques 1D) ;
- les **listes** (patron de listes doublement chaînées) ;
- les **ensembles** ;
- les **tableaux associatifs** ;
- et d'autres conteneurs que nous n'avons pas le temps d'étudier.

`http://www.cplusplus.com/reference/stl/`

Introduction

La STL définit un ensemble de classes spécialisées dans le stockage d'éléments et appelées **conteneurs**, qui sont *paramétrées par le type des éléments à contenir* :

- les **vecteurs** (patron de tableaux dynamiques 1D) ;
- les **listes** (patron de listes doublement chaînées) ;
- les **ensembles** ;
- les **tableaux associatifs** ;
- et d'autres conteneurs que nous n'avons pas le temps d'étudier.

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

Introduction

La STL définit un ensemble de classes spécialisées dans le stockage d'éléments et appelées **conteneurs**, qui sont *paramétrées par le type des éléments à contenir* :

- les **vecteurs** (patron de tableaux dynamiques 1D) ;
- les **listes** (patron de listes doublement chaînées) ;
- les **ensembles** ;
- les **tableaux associatifs** ;
- et d'autres conteneurs que nous n'avons pas le temps d'étudier.

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

Introduction

La STL définit un ensemble de classes spécialisées dans le stockage d'éléments et appelées **conteneurs**, qui sont *paramétrées par le type des éléments à contenir* :

- les **vecteurs** (patron de tableaux dynamiques 1D) ;
- les **listes** (patron de listes doublement chaînées) ;
- les **ensembles** ;
- les **tableaux associatifs** ;
- et d'autres conteneurs que nous n'avons pas le temps d'étudier.

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

Introduction

La STL définit un ensemble de classes spécialisées dans le stockage d'éléments et appelées **conteneurs**, qui sont *paramétrées par le type des éléments à contenir* :

- les **vecteurs** (patron de tableaux dynamiques 1D) ;
- les **listes** (patron de listes doublement chaînées) ;
- les **ensembles** ;
- les **tableaux associatifs** ;
- et d'autres conteneurs que nous n'avons pas le temps d'étudier.

`http://www.cplusplus.com/reference/stl/`

Exemple

```
#include <list>    // listes doublement chaînées
#include <vector>  // tableaux dynamiques 1D usuels
#include <deque>   // tableaux dynamiques 1D à double fin
#include <set>     // ensembles
#include <map>     // tableaux associatifs

#include <string>  // les chaînes du C++

using namespace std ;
...
list< int > li ;       // une liste doublement chaînée d'entiers
list< CPersonne > lp ; // une liste de personnes

vector< double > vd ;  // un tableau dynamique 1D de flottants double précision

set< char > sc ;       // un ensemble de caractères

// Type clé, Type associé
map<string, int> mois ;
mois["janvier"] = 1 ;
...
```

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Conteneurs séquentiels et conteneurs associatifs

Conteneurs séquentiels :

- leurs éléments sont rangés "linéairement" ;
- ajouts et suppressions relativement faciles (surtout en queue !) ;
- **recherche coûteuse** (linéaire !) ;
- exemples : vector, list, deque (tab 1D avec insertion au début ok, mais pas de contiguïté globale)...

Conteneurs associatifs :

- leurs éléments sont associés à une clé qui est utilisée pour les ranger (relation d'ordre) et pour y accéder ;
- **recherche rapide** (sous-linaire) ;
- exemples : *clé confondue avec élément* : set, multiset, *clé différente de l'élément* : map, multimap (1 élément unique par clé pour set et map)...

Introduction

La STL définit des **itérateurs** pour *parcourir de manière efficace ses conteneurs*.

Lors d'un parcours, un itérateur pointe sur un élément donné d'un conteneur : un **itérateur généralise la notion de pointeur** et les opérations autorisées sur les pointeurs, le sont (en général) pour les itérateurs.

Introduction

La STL définit des **itérateurs** pour *parcourir de manière efficace ses conteneurs*.

Lors d'un parcours, un itérateur pointe sur un élément donné d'un conteneur : un **itérateur généralise la notion de pointeur** et les opérations autorisées sur les pointeurs, le sont (en général) pour les itérateurs.

Différents types d'itérateur

Les itérateurs, de celui qui a le moins de fonctionnalités à celui qui en a le plus :

- **itérateur unidirectionnel** : ++, déréférencement *, ->, == et != ;
- **itérateur bidirectionnel** : idem unidirectionnel et -- ;
- **itérateur à accès direct** (*random access iterator*) : idem bidirectionnel et <, >, <=, >=, +=, -=, +, - : it+i, *(it+i) ou it[i] a un sens ;

Exemples : `vector` possède un itérateur à accès direct tandis que `list` ne possède qu'un itérateur bidirectionnel.

Différents types d'itérateur

Les itérateurs, de celui qui a le moins de fonctionnalités à celui qui en a le plus :

- **itérateur unidirectionnel** : ++, déréférencement *, ->, == et != ;
- **itérateur bidirectionnel** : idem unidirectionnel et -- ;
- **itérateur à accès direct** (*random access iterator*) : idem bidirectionnel et <, >, <=, >=, +=, -=, +, - : it+i, *(it+i) ou it[i] a un sens ;

Exemples : vector possède un itérateur à accès direct tandis que list ne possède qu'un itérateur bidirectionnel.

Différents types d'itérateur

Les itérateurs, de celui qui a le moins de fonctionnalités à celui qui en a le plus :

- **itérateur unidirectionnel** : ++, déréférencement *, ->, == et != ;
- **itérateur bidirectionnel** : idem unidirectionnel et -- ;
- **itérateur à accès direct** (*random access iterator*) : idem bidirectionnel et <, >, <=, >=, +=, -=, +, - : it+i, *(it+i) ou it[i] a un sens ;

Exemples : vector possède un itérateur à accès direct tandis que list ne possède qu'un itérateur bidirectionnel.

Différents types d'itérateur

Les itérateurs, de celui qui a le moins de fonctionnalités à celui qui en a le plus :

- **itérateur unidirectionnel** : ++, déréférencement *, ->, == et != ;
- **itérateur bidirectionnel** : idem unidirectionnel et -- ;
- **itérateur à accès direct** (*random access iterator*) : idem bidirectionnel et <, >, <=, >=, +=, -=, +, - : it+i, *(it+i) ou it[i] a un sens ;

Exemples : vector possède un itérateur à accès direct tandis que list ne possède qu'un itérateur bidirectionnel.

Exemple

objetConteneur.begin() : renvoie 1 itérateur pointant sur 1er elm
objetConteneur.end() : renvoie 1 itérateur pointant sur (last+1)

```
#include <iostream>
#include <vector>  // vecteurs et les itérateurs sur les vecteurs

using namespace std ;

...

vector< double > vd ;                              // construction d'un vector vide
for(int i=0; i<10; i++) vd.push_back(i) ;          // ajout en queue de 10 éléments

vector< double >::iterator it = vd.begin() ;       // itérateur qui pointe sur le 1er élément du
                                                   // conteneur
vector< double >::iterator it_fin = vd.end() ; // valeur itérateur à la fin du parcours

while( it != it_fin )    // parcours du conteneur via l'itérateur (il faut utiliser != et pas <)
{
    cout << *it << " " ; // affichage de l'élément courant

    it++ ;               // mon itérateur va pointer sur l'élément suivant
}
```

Exemple avec une boucle for

```
#include <iostream>
#include <vector>  // vecteurs et les itérateurs sur les vecteurs

using namespace std ;

...

vector< double > vd ;                          // construction d'un vector vide
for(int i=0; i<10; i++) vd.push_back(i) ;      // ajout en queue de 10 éléments

for( vector< double >::iterator it = vd.begin() ;
     it != vd.end() ; // utiliser != car valide pour tout type d'itérateur
     it++ )
{
    cout << *it << " " ; // affichage de l'élément courant
}

...
```

Exemple avec une liste

```
#include <iostream>
#include <list>  // lites doublement chaînées et ses itérateurs

#include "CPersonne.h"

using namespace std ;

...

list< CPersonne > lp ;

lp.push_back( CPersonne() ) ;
lp.push_back( CPersonne("John") ) ;

for( list< CPersonne >::iterator it = lp.begin() ;
     it != lp.end() ; // utiliser != car valide pour tout type d'itérateur
     it++ )
{
    cout << *it << " " ; // affichage de l'élément courant
}

...
```

Itérateur en sens inverse

Tous les conteneurs qui disposent d'un itérateur au moins bidirectionnel, disposent de :

- Conteneur : :reverse_iterator, de rbegin() et rend() pour parcourir le conteneur dans le sens inverse ;

Exemple

```
#include <iostream>
#include <vector>  // vecteurs et les itérateurs sur les vecteurs

using namespace std ;

...

vector< double > vd ;                          // construction d'un vector vide
for(int i=0; i<10; i++) vd.push_back(i) ;      // ajout en queue de 10 éléments

for( vector< double >::reverse_iterator it = vd.rbegin() ;
     it != vd.rend() ; // utiliser != car valide pour tout type d'itérateur
     it++ )
{
    cout << *it << " " ; // affichage de l'élément courant
}

...
```

Quelques précisions

Un itérateur peut devenir invalide dans les cas suivants :

- vector : insertion ou suppression d'un élément (réallocation dynamique) ;
- list / set / map : la suppression d'un élément invalide seulement les itérateurs qui pointaient sur l'élément supprimé ;

Que faut-il faire si on doit faire des suppressions/insertions dans un vecteur au sein d'une boucle gérée via un itérateur ? Utiliser un 2nd vecteur copie du 1er !

Quelques précisions

Un itérateur peut devenir invalide dans les cas suivants :

- vector : insertion ou suppression d'un élément (réallocation dynamique) ;
- list / set / map : la suppression d'un élément invalide seulement les itérateurs qui pointaient sur l'élément supprimé ;

Que faut-il faire si on doit faire des suppressions/insertions dans un vecteur au sein d'une boucle gérée via un itérateur ? Utiliser un 2nd vecteur copie du 1er !

Quelques précisions

Un itérateur peut devenir invalide dans les cas suivants :

- `vector` : insertion ou suppression d'un élément (réallocation dynamique) ;
- `list` / `set` / `map` : la suppression d'un élément invalide seulement les itérateurs qui pointaient sur l'élément supprimé ;

Que faut-il faire si on doit faire des suppressions/insertions dans un vecteur au sein d'une boucle gérée via un itérateur ? Utiliser un 2nd vecteur copie du 1er !

Quelques règles

Si une classe est dédiée à être contenue dans un conteneur de la STL, alors :

- son constructeur de recopie et son opérateur d'affectation doivent être publics ;
- si la classe contient un emplacement dynamique, le constructeur de recopie et l'opérateur d'affectation doivent être redéfinis ;
- il est conseillé d'avoir un constructeur sans argument.

Quelques règles

Si une classe est dédiée à être contenue dans un conteneur de la STL, alors :

- son constructeur de recopie et son opérateur d'affectation doivent être publics ;
- si la classe contient un emplacement dynamique, le constructeur de recopie et l'opérateur d'affectation doivent être redéfinis ;
- il est conseillé d'avoir un constructeur sans argument.

Quelques règles

Si une classe est dédiée à être contenue dans un conteneur de la STL, alors :

- son constructeur de recopie et son opérateur d'affectation doivent être publics ;
- si la classe contient un emplacement dynamique, le constructeur de recopie et l'opérateur d'affectation doivent être redéfinis ;
- il est conseillé d'avoir un constructeur sans argument.

Exemples

```
#include <vector>  // vecteurs et les itérateurs sur les vecteurs
#include "CMaClasse.h"

using namespace std ;

...

vector< CMaClasse > v(3) ; // Vecteur de taille 3, appel au constructeur sans argument
                           // de CMaClasse

vector< CMaClasse > v2(v) ;// Construction par recopie, appel au constructeur de recopie
                           // de CMaClasse

v = v2 ;                   // en fonction de la nouvelle capacité, soit affectation, soit
                           // destruction suivie de construction par recopie

...
```

Plan

- push_back(elm) : insère un élément elm de type T à la fin ;
- pop_back( ) : supprime le dernier élément à la fin ;
- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.

- push_back(elm) : insère un élément elm de type T à la fin ;
- pop_back( ) : supprime le dernier élément à la fin ;
- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.

- push_back(elm) : insère un élément elm de type T à la fin ;
- pop_back( ) : supprime le dernier élément à la fin ;
- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.

- push_back(elm) : insère un élément elm de type T à la fin ;
- pop_back( ) : supprime le dernier élément à la fin ;
- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.

- push_back(elm) : insère un élément elm de type T à la fin ;
- pop_back( ) : supprime le dernier élément à la fin ;
- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- front( ) : renvoie le 1er élément ;
- back( ) : renvoie le dernier élément ;
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- find(cleElm) : cherche l'élément dont la clé est cleElm et retourne un iterateur vers cet élément si trouvé ou vers conteneur.end() sinon.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- find(cleElm) : cherche l'élément dont la clé est cleElm et retourne un iterateur vers cet élément si trouvé ou vers conteneur.end() sinon.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- find(cleElm) : cherche l'élément dont la clé est cleElm et retourne un iterateur vers cet élément si trouvé ou vers conteneur.end() sinon.

- clear( ) : supprime tous les éléments ;
- empty( ) : tester si le conteneur est vide (ne contient pas d'élément) ;
- size( ) : le nombre d'éléments ;
- find(cleElm) : cherche l'élément dont la clé est cleElm et retourne un iterateur vers cet élément si trouvé ou vers conteneur.end() sinon.

- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

- insert(iterator, elm) : insère un élément elm de type T avant la position de l'itérateur ;
- erase(iterator) : supprime un élément à la position de l'itérateur ;
- erase(iteratorD, iteratorF) : supprime tous les éléments entre les positions iteratorD et iteratorF.
- swap(std : :memeTypeCont &) : échanger le contenu de 2 conteneurs de même type.

Plan

- Implanté par des **arbres binaires de recherche** ;
- Un std : :map contient des **itérateurs bidirectionnels** *avec 1 accès direct* via **[]** (les autres conteneurs associatifs sont seulement bidirectionnels).
- Chaque élément d'un conteneur std : :map est une **paire (clé, élément)** (std : :pair<const Key, T>, http://www.cplusplus.com/reference/utility/pair/).

```
template < class Key,                                  // map::key_type
           class T,                                    // map::mapped_type
           class Compare = less<Key>,                  // map::key_compare
           class Alloc = allocator<pair<const Key,T> > // map::allocator_type
           > class map;
```

- Implanté par des **arbres binaires de recherche** ;
- Un std : :map contient des **itérateurs bidirectionnels** *avec 1 accès direct* via **[]** (les autres conteneurs associatifs sont seulement bidirectionnels).
- Chaque élément d'un conteneur std : :map est une **paire (clé, élément)** (std : :pair<const Key, T>, http://www.cplusplus.com/reference/utility/pair/).

```
template < class Key,                                  // map::key_type
           class T,                                    // map::mapped_type
           class Compare = less<Key>,                  // map::key_compare
           class Alloc = allocator<pair<const Key,T> > // map::allocator_type
           > class map;
```

- Implanté par des **arbres binaires de recherche** ;
- Un std : :map contient des **itérateurs bidirectionnels** *avec 1 accès direct* via **[]** (les autres conteneurs associatifs sont seulement bidirectionnels).
- Chaque élément d'un conteneur std : :map est une **paire (clé, élément)** (std : :pair<const Key, T>, `http://www.cplusplus.com/reference/utility/pair/`).

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

Exemple

```
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;

  mymap['a']=50;  // insertion de la paire clé-valeur ('a',50)
  mymap['b']=100;
  mymap.insert( make_pair('c', 150) ) ; // aboutit seulement si la clé 'c' n'existe pas déjà
  mymap['d']=200;
  mymap['e'] ;    // insertion de la paire clé-valeur ('e',0)

  it=mymap.find('b');
  mymap.erase (it);  // on peut aussi faire un erase( clé )
  mymap.erase (mymap.find('d'));

  // afficher les resultats :
  std::cout << "elements in mymap:" << '\n';
  std::cout << "a val: " << mymap.find('a')->second << '\n';// second est le contenu du 2nd
                                                           // élément d'une paire (Type T)
  std::cout << "c key: " << mymap.find('c')->first << '\n'; // first est le contenu du 1er
                                                           // élément d'une paire (Type Key)
  return 0; }
```

Plan

Introduction

La STL a parfois besoin de **foncteurs** dans ses algorithmes, en particulier pour déterminer l'ordre de 2 éléments.

Un foncteur est un objet "fonction" qui a l'opérateur () surchargé, on peut donc l'utiliser comme une fonction !

```
class monFoncteur
{
  public :
  // prédicat binaire :
  bool operator()(T& a, T& b) // équivalent ici à une fonction de 2 arguments
  {
      return a < b ;
  }
} ;
...
monFoncteur f ; // création du foncteur
f(x,y) ;        // utilisation du foncteur
```

Introduction

La STL a parfois besoin de **foncteurs** dans ses algorithmes, en particulier pour déterminer l'ordre de 2 éléments.

Un foncteur est un objet "fonction" qui a l'opérateur **()** surchargé, on peut donc l'utiliser comme une fonction !

```
class monFoncteur
{
  public :
  // prédicat binaire :
  bool operator()(T& a, T& b) // équivalent ici à une fonction de 2 arguments
  {
      return a < b ;
  }
} ;
...
monFoncteur f ; // création du foncteur
f(x,y) ;        // utilisation du foncteur
```

Plan

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Introduction

La STL fournit des **algorithmes** à utiliser sur ses conteneurs via leurs itérateurs : `http://www.cplusplus.com/reference/algorithm/`.

- Le tri : sort si it à accès direct et opérateur < type élément ; algorithme non-autorisé pour les conteneurs associatifs ;
- La recherche : find si opérateur == type élément ;
- L'élément de valeur min : min_element(iterD,iterF, fonct) ;
- La copie : copy ;
- La suppression : remove ...

Souvent **algorithme sur intervalle** : [itDebutInclu, itFinExclu(

Exemple d'algorithme : std : :count

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

int main () {
  // counting elements in array:
  int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
  int mycount = std::count (myints, myints+8, 10);
  std::cout << "10 appears " << mycount << " times.\n";

  // counting elements in container:
  std::vector<int> myvector (myints, myints+8);
  mycount = std::count (myvector.begin(), myvector.end(), 20);
  std::cout << "20 appears " << mycount  << " times.\n";

  return 0;
}
```

Exemple d'algorithme : std : :swap

```
// swap algorithm example (C++98)
#include <iostream>     // std::cout
#include <algorithm>    // std::swap
#include <vector>       // std::vector

int main () {

  int x=10, y=20;                                  // x:10 y:20
  std::swap(x,y);                                  // x:20 y:10

  std::vector<int> foo (4,x), bar (6,y);           // foo:4x20 bar:6x10
  std::swap(foo,bar);                              // foo:6x10 bar:4x20

  std::cout << "foo contains:";
  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Exemple d'algorithme : std : :sort

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

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

  return 0;
}
```

Exemple d'algorithme : std : :sort

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

class ordre
{
  public :
    bool operator ()(int a, int b) const
    {
        return a < b;
    }
};

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using user comparison :
  std::sort (myvector.begin(), myvector.begin()+4, ordre());  //(12 32 45 71)26 80 53 33

  return 0;
}
```

Exemple d'algorithme : std : :copy

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

...

std::vector< int > v ;
...
std::list< int > l(v.size()) ;

std::copy(v.begin(), v.end(), l.begin()) ; // recopié [v.begin(), v.end()(
                                           // à partir de l.begin()

...
```

Plan

La fonction membre remove_if des listes

prédicat : fonction booléenne retournant **true** si la condition est vérifiée et **false** sinon.

remove_if : supprime toutes les éléments de la liste vérifiant le **prédicat** unaire en argument.

```
#include <list>          // std::list

...
bool estPaire(int n) { return n%2 ; }     // prédicat unaire
...
int t[] = {2, 5, 78, 4, 3, 7, 6} ;
std::list< int > l(t, t+7) ;              // l contient 2, 5, 78, 4, 3, 7, 6
l.remove_if(estPaire) ;                   // l contient 5, 3, 7
// Avec la STL, là où je peux utiliser une adresse de fonction, je peux
// en général utiliser un foncteur
...
```

Remarque : remove_if existe aussi en algorithme

La fonction membre remove_if des listes

prédicat : fonction booléenne retournant **true** si la condition est vérifiée et **false** sinon.

remove_if : supprime toutes les éléments de la liste vérifiant le **prédicat** unaire en argument.

```
#include <list>          // std::list

...
bool estPaire(int n) { return n%2 ; }     // prédicat unaire
...
int t[] = {2, 5, 78, 4, 3, 7, 6} ;
std::list< int > l(t, t+7) ;              // l contient 2, 5, 78, 4, 3, 7, 6
l.remove_if(estPaire) ;                   // l contient 5, 3, 7
// Avec la STL, là où je peux utiliser une adresse de fonction, je peux
// en général utiliser un foncteur
...
```

Remarque : remove_if existe aussi en algorithme.

L'algorithme find_if

std : :find_if(itD, itF, prédicatUnaire)

```
#include <iostream>
#include <list>          // std::list

...
bool estPaire(int n) { return n%2 ; }     // prédicat unaire
...
int t[] = { 2, 5, 78, 4, 3, 7, 6} ;
std::list< int > l(t, t+7) ;              // l contient 2, 5, 78, 4, 3, 7, 6
std::list< int >::iterator res = std::find_if(l.begin(), l.end(), estPaire ) ;
std::cout << " La 1ere valeur paire est : " << *res ;

...
```

Prédicats binaires prédéfinies

Dans <functional> il existe des patron de prédicats binaires permettant de comparer 2 éléments du même type :

- less < T > correspondant à la comparaison par l'opérateur < ;
- greater < T > correspondant à la comparaison par l'opérateur > ;
- list<int> li(...); ... li.sort(greater<int>);

http://www.cplusplus.com/reference/functional/

Prédicats binaires prédéfinies

Dans <functional> il existe des patron de prédicats binaires permettant de comparer 2 éléments du même type :

- less < T > correspondant à la comparaison par l'opérateur < ;
- greater < T > correspondant à la comparaison par l'opérateur > ;
- list<int> li(...) ; ... li.sort(greater<int>) ;

http://www.cplusplus.com/reference/functional/

Prédicats binaires prédéfinies

Dans <functional> il existe des patron de prédicats binaires permettant de comparer 2 éléments du même type :

- less < T > correspondant à la comparaison par l'opérateur < ;
- greater < T > correspondant à la comparaison par l'opérateur > ;
- list<int> li(...); ... li.sort(greater<int>);

`http://www.cplusplus.com/reference/functional/`