R2.01 : Développement Orienté Objets C++

cours 3

François Delobel, Anaïs Durand, Abdel Hasbani, Laurent Provot, Carine Simon

IUT CLERMONT AUVERGNE
Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

La bibliothèque standard

Bibliothèque standardisée selon la norme ISO, et appartenant au langage.
Contient :

- Outils *chaînes de caractères*, expressions rationelles ;
- Manipulation de *flux* (fichiers, entrée et sortie standard) ;
- Conteneurs (collections d'objets) ;
- Algorithmes sur les conteneurs.
- Les *complexités* sont spécifiées!
- Aussi: dates/durées, système de fichier, concurrence, coroutines, math...

Incluse dans l'espace de noms `std`.

Les conteneurs

Classes destinées à contenir des collections d'objets.

- Peuvent contenir n'impore quel type d'objet (génériques).
- Peuvent être construits avec des *listes d'initialiseurs*.
- Ont des fonctions membres pour ajouter et supprimer des éléments et exécuter d'autres opérations.
- Manipulables par les *algorithmes* de la STL.

Des conteneurs de séquence

- L'utilisateur décide de la position des éléments.
  - `array` : tableau de taille fixe
    avantage : accès direct aux éléments
    inconvénients : taille fixe
    seulement modification d'éléments
  - `vector` : tableau dynamique
    avantages : accès direct aux éléments
    et taille qui s'adapte automatiquement
    inconvénient : insertion et suppression d'élément implique d'en décaler
  - `list` : liste doublement chaînée
    insertion et supression rapides n'importe où,
    inconvénient : pas d'accès direct à un élément
- Ces trois conteneurs sont bi-directionnels (parcours dans les deux sens).

Des conteneurs associatifs clé / valeur

- Trouvent vite la *valeur* à partie de la *clé*.
- **Exemple** : annuaire de pseudos qui associe à chaque pseudo (clé) l'identité correspondante (valeur)
- Pas de doublons de clé (cf versions `multimap` au lieu de `map`).
- `map` : Clés ordonnables (<): accès en *O(log(n))*
  Souvent arbre binaire trié.
- `unordered_map` : Clés hashables : accès en *O(1)*
  Table de hachage.

Des conteneurs associatifs avec juste une clé

- La clé est la valeur!
  - set : les éléments sont triés
  - unordered_set : les éléments ne sont pas triés
- Pas de doublons de clé (cf versions `multiset` au lieu de `set`).

Parcours de conteneurs

Pour parcourir les éléments d'un conteneur, on peut utiliser :

- une boucle for range, exemple :

```
std::array<int> vec{2,4,6,8};
for (int num : vec){
    std::cout << num << " ";
}
```

affichage : 2 4 6 8

- mais aussi un itérateur

Flash Back

```
int t[] = {2,4,6,8};
int *begin = t;
int *end = begin + 4;

for (int *p = begin ; p != end ; ++p ) {
    cout << *p << ' ';
}
```

- `p` pointe sur un élément;
- `*p` renvoie l'élément pointé;
- `begin` : un pointeur sur le premier élément;
- `end` : pointeur sur le premier élément **après** le dernier;
- `*end` serait une grosse bourde;
- `p != end` (égalité/inégalité);
- `++p` décale le pointeur au suivant.

Itérateurs, vous les connaissez!

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

using namespace std;

int main() {
  vector<int> t = {2,4,6,8};
  vector<int>::iterator begin = t.begin();
  vector<int>::iterator end = t.end();

  for (vector<int>::iterator p = begin ; p != end ; ++p ) {
    cout << *p << ' ';
  }
}
```

- L'iterateur *peut être* vu comme un type interne du conteneur.
- Le pointeur natif est un itérateur.

Itérateurs, la base!

- Désignent une position dans un conteneur (**sauf** `end()`)
- On les récupère d'un conteneur
  - `c.begin()` ou `begin(c)` : itérateur sur le *premier* élément
  - `c.end()` ou `end(c)` : itérateur **après** le *dernier* élément

Les itérateurs ont des méthodes comme :

- `++` : permettant de faire avancer l'itérateur
- `*` : pour accéder à l'objet pointé par l'itérateur
  - `*i = 4`
  - `cout << *i`

Itérateurs, la base!

- Désignent une position dans un conteneur (**sauf** `end()`)
- On les récupère d'un conteneur
  - `c.begin()` ou `begin(c)` : itérateur sur le *premier* élément
  - `c.end()` ou `end(c)` : itérateur **après** le *dernier* élément

Les itérateurs ont des méthodes comme :

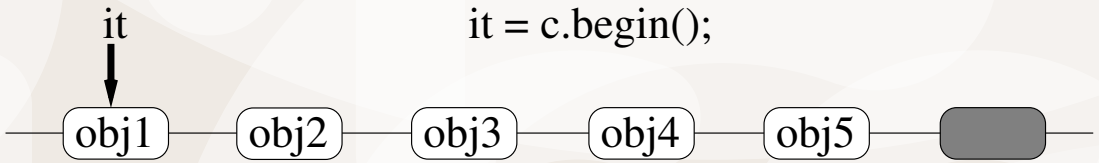
- `++` : permettant de faire avancer l'itérateur
- `*` : pour accéder à l'objet pointé par l'itérateur
  - `*i = 4`
  - `cout << *i`

Itérateurs, la base!

- Désignent une position dans un conteneur (**sauf** `end()`)
- On les récupère d'un conteneur
  - `c.begin()` ou `begin(c)` : itérateur sur le *premier* élément
  - `c.end()` ou `end(c)` : itérateur **après** le *dernier* élément

Les itérateurs ont des méthodes comme :

- `++` : permettant de faire avancer l'itérateur
- `*` : pour accéder à l'objet pointé par l'itérateur
  - `*i = 4`
  - `cout << *i`

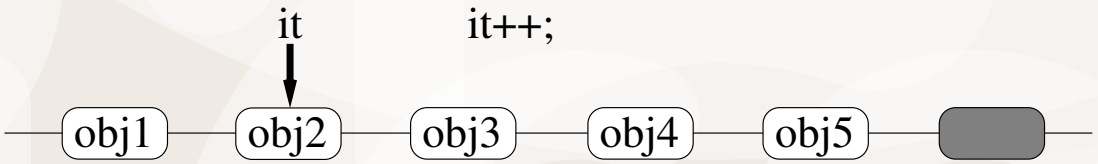


Itérateurs, la base!

- Désignent une position dans un conteneur (**sauf** `end()`)
- On les récupère d'un conteneur
  - `c.begin()` ou `begin(c)` : itérateur sur le *premier* élément
  - `c.end()` ou `end(c)` : itérateur **après** le *dernier* élément

Les itérateurs ont des méthodes comme :

- `++` : permettant de faire avancer l'itérateur
- `*` : pour accéder à l'objet pointé par l'itérateur
  - `*i = 4`
  - `cout << *i`

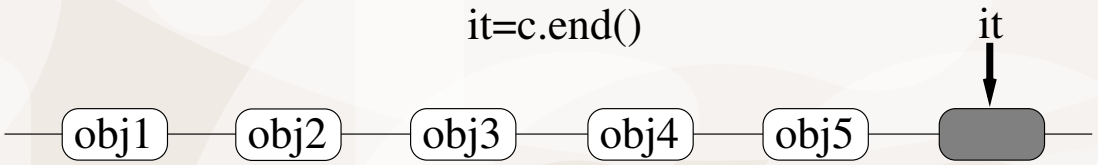


Itérateurs, la base!

- Désignent une position dans un conteneur (**sauf** `end()`)
- On les récupère d'un conteneur
  - `c.begin()` ou `begin(c)` : itérateur sur le *premier* élément
  - `c.end()` ou `end(c)` : itérateur **après** le *dernier* élément

Les itérateurs ont des méthodes comme :

- `++` : permettant de faire avancer l'itérateur
- `*` : pour accéder à l'objet pointé par l'itérateur
  - `*i = 4`
  - `cout << *i`



Itérateurs et constance

- Un conteneur peut être *constant*.
- Un itérateur peut modifier un conteneur (via $*$).

Alors, comment désigner des parties d'un conteneur constant?

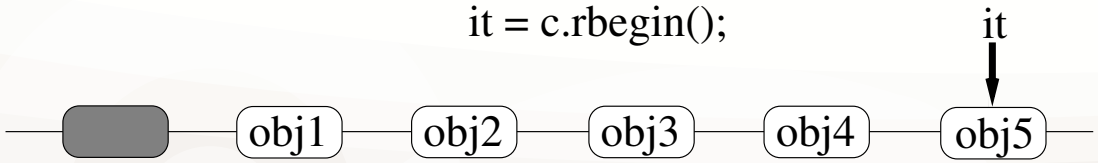
- `const_iterator`
  - `c.cbegin()`
  - `c.cend()`

Itérateurs : reverse

- `reverse_iterator` : on lui dit d'avancer, il recule!
  - `c.rbegin()`
  - `c.rend()`
- Oui, il y a un `constant_reverse_iterator`, à coups de `crbegin()`!

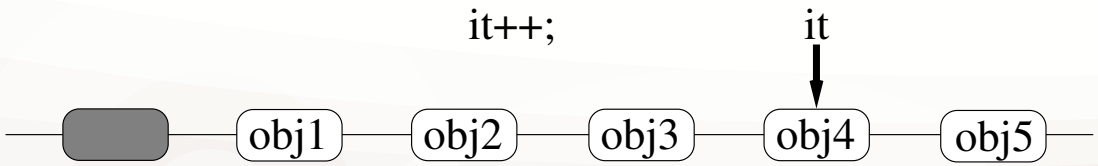
Itérateurs : reverse

- `reverse_iterator` : on lui dit d'avancer, il recule!
  - `c.rbegin()`
  - `c.rend()`
- Oui, il y a un `constant_reverse_iterator`, à coups de `crbegin()`!



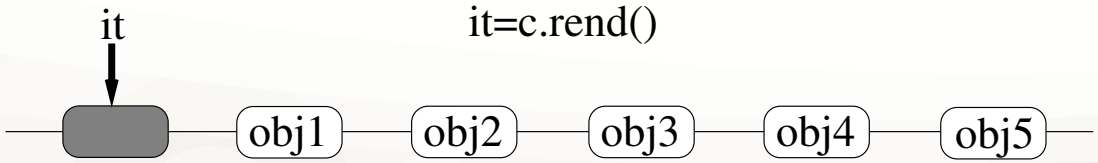
Itérateurs : reverse

- `reverse_iterator` : on lui dit d'avancer, il recule!
  - `c.rbegin()`
  - `c.rend()`
- Oui, il y a un `constant_reverse_iterator`, à coups de `crbegin()`!



Itérateurs : reverse

- `reverse_iterator` : on lui dit d'avancer, il recule!
  - `c.rbegin()`
  - `c.rend()`
- Oui, il y a un `constant_reverse_iterator`, à coups de `crbegin()`!



Itérateurs

Exemple :

```
std::vector<int> v{10,20,30,40};

std::vector<int>::const_iterator it = v.cbegin();
std::cout << "le premier : " << *it << std::endl;
it++;
std::cout << "le deuxième : " << *it << std::endl;

for (auto it2 = v.cbegin(); it2 != v.cend(); it2++){
    std::cout << *it2 << " ";
}
std::cout << std::endl;
for (auto it3 = v.crbegin(); it3 != v.crend(); it3++){
    std::cout << *it3 << " ";
}
```

affichage : le premier : 10
le deuxième : 20
10 20 30 40
40 30 20 10

Itérateurs, conteneurs et structures de données

Famille des vecteurs (`vector`, `array`...)

- Peut sauter directement (en *O(1)*) *n* éléments plus loin.
- *Random Access Conteneur* dont les itérateurs sont *random access*.
- Leurs itérateurs supportent `+=`, `[i]`...

Famille des listes (`list`, `slist`)

- Impossible d'aller directement (en *O(1)*) *n* éléments plus loin.
- N'est donc pas un *Random Access Conteneur*/

Nous vous cachons plein d'autres propriétés!

Itérateurs

Exemple :

```
using namespace std;

vector<int> v{10,20,30,40};
vector<int>::iterator it = begin(v);
vector<int>::iterator it2 = next(it, 2);

cout << "le premier : " << *it ;
cout << " , le troisième : " << *it2 << endl;

int d = distance(it,it2);
cout << "distance entre les deux : " << d << endl;

for (auto it3 = begin(v); it3 != end(v); advance(it3,1)){
    cout << *it3 << " ";
}
```

affichage : le premier : 10 , le troisième : 30
distance entre les deux : 2
10 20 30 40

Algorithmes

- Définis dans `<algorithm>`.
- Sont de *pures fonctions*, pas des *méthodes*.
- Travaillent sur des conteneurs ou des *parties* de conteneurs.
- Prennent des *iterateurs* en paramètre.
  - Parfois, exigent des itérateurs particuliers (genre `random_access_iterator`).
- Renvoient des *iterateurs*, en particulier `end()`.

Exemples de fonction :
find, for_each, copy, replace, remove, sort

Exemple : it find(itDeb, itFin, val)

- retourne un itérateur sur le premier élément entre itDeb et (itFin-1) égal à val
- ou itFin si il n'a pas trouvé!

```
std::vector<int> v {10,20,30,40};
std::vector<int>::iterator it;

it = find (v.begin(), v.end(), 30);
if (it != v.end())
    std::cout << "élément␣trouvé␣:␣" << *it << '\n';
else
    std::cout << "élément␣non␣trouvé␣\n";
```

affichage : élément trouvé : 30

Exemple : for_each(itDeb, itFin, fn)

applique la fonction fn à tous les éléments entre itDeb et itFin

```
void affiche (int i) {
  std::cout << '␣' << i;
}
void test(){
  std::vector<int> v{5, 10, 15, 20};
  v.push_back(30);
  std::cout << "contenu␣de␣v␣:␣";
  for_each(v.begin(), v.end(), affiche);
}
```

affichage : contenu de v : 5 10 15 20 30

Exemple : copy(itDeb, itFin, newItDeb)

copy(itDeb, itFin, newItDeb) : copy les éléments qui sont entre itDeb et itFin à partir de newItDeb

```
std::vector<int> v {10,20,30,40,50,60,70};
std::list<int> l(4);

std::copy ( v.begin()+1 , v.begin()+5, l.begin() );

std::cout << "la liste contient : ";
std::list<int>::iterator it;
for (it = l.begin(); it!= l.end(); ++it)
  std::cout << ' ' << *it;
```

affichage : 20 30 40 50

Exemple : replace(itDeb, itFin, oldVal, newVal)
remplace tous les éléments égaux à oldVal par newVal entre itDeb et itFin

```
std::vector<int> v { 10, 20, 30, 40, 20, 60, 70, 20 };
std::vector<int>::iterator it1 = v.begin();     // it1 sur 10
std::vector<int>::iterator it2 = next(it1,5);   // it2 sur 60
std::replace (it1, it2 , 20, 99);

std::cout << "v contient : ";
for (it1 = v.begin(); it1 != v.end(); ++it1)
    std::cout << ' ' << *it1;
```

affichage : 10 99 30 40 99 60 70 20

Exemple : newItFin remove(itDeb, itFin, val)

supprime tous les objets égaux à val entre itDeb et itFin, et retourne le nouvel itérateur sur la fin

```
std::vector<int> v = {10,20,30,40,20,60,70,20};

std::vector<int>::iterator itDeb = v.begin();
std::vector<int>::iterator itFin = v.end();

itFin = std::remove (itDeb, itFin, 20);
std::cout << "range contains:";
for (auto it = itDeb; it != itFin; ++it)
    std::cout << ' ' << *it;
```

affichage : 10 30 40 60 70

Exemple : sort(itDeb, itFin)

- Tri dans l'ordre croissant les éléments compris entre itDeb et itFin.
- itDeb et itFin doivent être des *random iterator* !

```
std::vector<int> v {32,12,45,26,53,33};

// trie seulement une partie
std::sort(v.begin(), v.begin()+3);  //( 12 32 45 ) 26 53 33

// trie tout
std::sort (v.begin(), v.end()); // 12 26 32 33 45 53
```

Et le reste?

cppreference.com