

Programmation Avancée Les différents mécanismes des langages (dont C++) pour la généricité

Norme ISO

Raphaëlle Chaîne
raphaelle.chaine@liris.cnrs.fr
2024-2025

1

1

IV Bibliothèque standard

- Ensemble de bibliothèques ISO regroupées dans l'espace de nom **std**
 - éléments de la bibliothèque standard C
`<cstdint>` `<climits>` `<cfloat>` `<cstdlib>` `<csignal>` `<ctime>`
`<csdarg>` `<cstdio>` `<cstring>` `<cctype>` `<wchar>` `<cwctype>`
`<cmath>` `<cassert>` `<locale>` `<setjmp>` `<errno>` `<iso646>`
 - des éléments spécifiques C++
(string, exception, gestion mémoire, identification de types, composants numériques, etc.)
 - IO Stream Library
 - **Standard Template Library (STL)**
structures de données et algorithmes

235

235

Standard Template Library

- Un ensemble de patrons génériques de classes et de fonctions
 - **conteneurs**
modèles génériques permettant de stocker des collections d'objets (séquences, types abstraits pile, file et file de priorité, tables associatives) avec des opérations de complexité garantie
 - **itérateurs**
abstraction de l'idée de pointeur permettant d'accéder aux éléments d'un conteneur selon une interface unique
(nouvelle syntaxe également offerte depuis C++11 pour parcourir les éléments d'un conteneur)
 - **algorithmes**
non écrits pour des conteneurs particuliers puisque n'accédant aux données qu'à travers des itérateurs
- Utilisent les template et non l'héritage couplé avec les fonctions virtuelles par soucis d'efficacité

236

236

Conteneurs

- Ensemble de modèles génériques représentant les structures de données les plus répandues
 - pour stocker des **séquences** : vector, deque, list
 - pour stocker des collections de **clefs** : set, multiset, avec des **valeurs associées** : map, multimap
- Paramétrés par le type de leurs éléments
- Chacun de ces modèles offre des fonctionnalités spécifiques
 - En particulier, tous les conteneurs offrent les fonctions membres `size_t size()` et `bool empty()`

237

237

- Exemple :

```
#include <list>
std::list<int> li1, // liste d'entiers (vide)
li2(9,55); // liste de 9 entiers valant chacun 55
int ti[] = {5, 9, 1};
std::list<int> li3(ti,t+3);
std::list<int> l = { 8, 15, 1, 9 }; //C++11

#include <vector>
std::vector<std::string> vs1, // vecteur de string (vide)
vs2(5,a); // 5 string copies de a
char m[]="to";
std::vector<char> mv(m,m+2);

#include<deque>
std::deque<double> dd1, //file à double entrée (vide)
dd2(6), // deque de 6 doubles
dd3(dd1); // par copie
```

238

238

- **vector** (tableau dynamique)
 - **accès direct** aux éléments
opérateur `[]`(int), fonction membre `at(int)`, accès au dernier élément par fonction membre `back()`
 - **insertion/suppression à la fin en temps constant**
fonctions membres `push_back(elt)`, `pop_back()`
 - **insertion/suppression en temps linéaire** sinon
fonctions membres `insert`, `erase`
 - nombre dynamique d'éléments avec gestion automatique d'extension mémoire si nécessaire
fonctions membres `size()`, `resize(nouvelletaille, elt)`, `capacity()`, `reserve(nouvellecap)`, `max_size()`

Implémentation correspondant à un tableau dynamique

239

239

Exemple :

```
std::vector<int> v;
v.push_back(5);
v.push_back(1);
v.insert( v.begin()+1, 9);
v.pop_back();
v.insert( v.begin(), 2,7);
std::cout <<v[1] << " " << v.at(2) << std::endl;
v[0]=v.at(1)=v.back()=1;
v.erase( v.begin()+2 );
v.insert( v.end(), 3);
std::cout << v.size() << std::endl;
v.resize(6,0);
v.resize(2);
```

240

240

Exemple :

```
std::vector<int> v;
v.push_back(5);           // 5
v.push_back(1);           // 5 1
v.insert( v.begin()+1, 9); // 5 9 1
v.pop_back();              // 5 9
v.insert( v.begin(), 2,7); // 7 7 5 9
std::cout <<v[1] << " " << v.at(2) << std::endl;
v[0]=v.at(1)=v.back()=1;   // 1 1 5 1
v.erase( v.begin()+2 );    // 1 1 1
v.insert( v.end(), 3);      // 1 1 1 3
std::cout << v.size() << std::endl;
v.resize(6,0);             // 1 1 1 3 0 0
v.resize(2);               // 1 1
```

241

241

- list (liste doublement chaînée)
 - insertion/suppression en temps constant à une position donnée
 insert, erase, push_front(elt), pop_front(),
 push_back(elt), pop_back()
 - pas d'accès direct à tous les éléments
 fonctions membres front() et back()
 - possibilité de parcours séquentiel dans les 2 sens
 - gestion automatique de l'espace utilisé
 fonctions membres size(), max_size()
 - des fonctions membres spécifiques
 remove(elt), unique, splice, reverse, sort, merge
- Implémentation correspondant à une liste chaînée

242

242

Exemple :

```
std::list<double> l(4,1.5);
l.push_front(5.9);
l.pop_back();
l.insert( l.begin(), 7.7);
l.reverse();
l.sort();
l.remove(1.5);
l.push_back(7.7);
l.unique(); // On peut aussi passer une fonction pour
            // comparer deux éléments et mettre en œuvre l'unicité
            // selon le critère de notre choix
l.erase(l.begin());
```

243

243

Exemple :

```
std::list<double> l(4,1.5); // 1.5 1.5 1.5 1.5
l.push_front(5.9);         // 5.9 1.5 1.5 1.5 1.5
l.pop_back();               // 5.9 1.5 1.5 1.5
l.insert( l.begin(), 7.7);  // 7.7 5.9 1.5 1.5 1.5
l.reverse();                // 1.5 1.5 1.5 5.9 7.7
l.sort();                   // 1.5 1.5 1.5 5.9 7.7
l.remove(1.5);              // 5.9 7.7
l.push_back(7.7);           // 5.9 7.7 7.7
l.unique();                  // 5.9 7.7
l.erase(l.begin());         // 7.7
```

244

244

- class deque (file à double entrée)
 - accès direct aux éléments en temps constant *
 opérateur [](int), fonctions membres at(int), back() et front()
 - insertion/suppression au début ou à la fin en temps constant*
 fonctions membres push_back(elt), pop_back(),
 push_front(elt), pop_front()
 - insertion/suppression en temps linéaire*, sinon
 fonctions membres insert, erase
 - nombre dynamique d'éléments, avec gestion automatique
 d'extension mémoire si nécessaire
 fonctions membres size(), resize(newtaille,elt), max_size()
 Pas de fonctions membres capacity(), reserve(taille)
- Implémentation correspondant à un tableau dynamique
 de tableaux de taille fixe (chunk), par exemple
- (*amorti)

245

245

- class array (tableau de taille fixe fournie en template)

```
std::array<std::string, 2> ha =
    {std::string("Marco"), "Polo"};
for(const auto& s: ha)
    std::cout << s << std::endl;
```

Principales opérations sur les std::array

```
at(unsigned int), operator [] (unsigned int), front(),
back()
```

246

246

Adaptateurs de séquences

- Fournis en association avec les séquences
 - Patrons de classes construits sur des conteneurs
- Définition d'une **nouvelle interface** pour un conteneur, afin de lui donner le comportement d'un type abstrait

stack, queue ou priority_queue

- Exemple :

```
#include <stack>
std::stack<int, std::vector<int> > pi;
#include <queue>
std::queue<int, std::deque<int> > fi;
std::priority_queue<int, std::deque<int> > fip;
std::priority_queue<int, std::deque<int>, std::greater<int> >
fip2;
```

247

247

Adaptateur stack

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

- Le conteneur utilisé à l'instanciation doit supporter les opérations back, push_back et pop_back
- Principales opérations sur les std::stack


```
push(elt), pop(), top(),
empty(), size()
```

248

248

Adaptateur queue

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

- Le conteneur utilisé à l'instanciation doit supporter les opérations front, push_back, pop_front et back
- Principales opérations sur les std::queue


```
push(elt), pop(), front(),
empty(), size()
```

249

249

Adaptateur priority_queue

```
template <class T, class Container=vector<T>, class Compare
=std::less<typename Container::value_type> >
class priority_queue;
```

Type abstrait queue de priorité offrant l'extraction de l'élément de plus grande priorité (au sens de la classe Compare *, qui teste l'**infériorité**)

- Le conteneur utilisé à l'instanciation doit supporter l'indexation ainsi que les opérations front, push_back et pop_back
- Principales opérations sur les priority_queue


```
push(elt), pop(), top(), empty(), size()
```

(*Compare étant une classe d'objets "fonctions de comparaison")

250

250

Exemple :

```
std::priority_queue<int, std::vector<int> > p1;
std::priority_queue<int, std::deque<int>, std::greater<int> > p2;
p1.push(9); p2.push(9);
p1.push(7); p2.push(7);
p1.push(30); p2.push(30);
while(!p1.empty())
{ std::cout << p1.top() << std::endl;
  p1.pop();
} // 30 9 7
while(!p2.empty())
{ std::cout << p2.top() << std::endl;
  p2.pop();
} // 7 9 30
```

251

251

Fabrication d'une classe d'objets "fonctions de comparaison"

- Classe disposant d'une surcharge de l'opérateur () correspondant à une fonction de comparaison : fonction de 2 arguments renvoyant 1 booléen

```
class MaCompare
{
public:
    bool operator () (int i, int j)
    { ... return ... }
};

int main()
{
    std::cout << MaCompare()(3,5) << std::endl; // Comparaison de 3 et 5
    std::priority_queue<int, std::vector<int>, MaCompare> p;
    ...
}
```

Construction d'un objet temporaire permettant de comparer 2 entiers

252

252

- Le dernier argument template d'une priority_queue peut aussi être une fonction

```
bool comparaison(int i, int j)
{ return (i >> 1) > (j >> 1); }
```

```
int main()
{
    std::cout << comparaison(3,5) << std::endl;
    std::priority_queue<int, std::vector<int>, decltype(comparaison)> p(comparaison);
    ...
}
```

- On découvrira bientôt les lambda fonctions!

```
int main()
{
    auto compare = [](int i, int j) { return (i >> 1) < (j >> 1); };
    std::priority_queue<int, std::vector<int>, decltype(compare)> p(compare);
    ...
}
```

253

253

Conteneurs associatifs

Les conteneurs associatifs ont pour vocation de stocker et de retrouver (des informations associées à) des clefs, en exploitant un ordre strict faible sur les clefs

- **set et multiset (*)**:
 - collection de clefs (sans association d'information)
 - les set ne peuvent contenir 2 clefs équivalentes (les multiset oui)
- **map et multimap (*)**:
 - collection de clefs avec des valeurs associées
 - les maps ne peuvent contenir 2 clefs équivalentes (les multimaps oui)

*« sorted » mais il existe aussi des « Hashed » conteneurs associatifs (hash_set, hash_multiset, hash_map, hash_multimap)

254

254

set et multiset

```
template <class Key, class Compare = less<Key> >
class set; // resp class multiset
```

- Principales opérations
insert(key), erase(key), find(key), size(), empty()
- Les set et multiset ne correspondent pas à des séquences (les opérations [], at, push_front, push_back, pop_front, pop_back ne sont donc pas définies)
- Un multiset conserve les éléments équivalents
- exemple
std::set<int> s; std::multiset<int> ms;
for(int i=0; i< 10; i++)
{ s.insert(i%2); ms.insert(i%2); }

Que contiennent les ensembles s et ms ?

s contient 2 éléments : 0 et 1; ms contient 10 éléments : 0 0 0 0 0 et 1 1 1 1 1

255

255

map

```
template <class Key, class T,
          class Compare = less<Key> >
class map;
```

- Une table associative est une collection de couples (clé, valeur) offrant l'accès à un couple à partir de sa clé
- Principale opération :
accès "direct" à une clé par operator[] (key)
Si la clé est présente... sinon INSERTION!!!!
- Les couples (clé, valeur) sont insérés sous forme de std::pair<key, T> (utiliser la fonction make_pair()) ou [])
- Utiliser find pour chercher une clé
- 2 éléments d'une map ne peuvent avoir des clefs équivalentes

256

256

- Exemple :

```
template<class K, class V>
std::ostream & operator << (std::ostream & os, std::pair<K,V> & e)
{
    os << " (" << e.first << ", " << e.second << ") ";
    return os;
}

int main()
{
    std::map<std::string, int> tel;
    tel["Fanny"] = 065555555;
    tel["Bozo"] = 014444444;
    tel["Jeez"] = 047777777;
    tel["Bozo"] = 02345678;
    std::cout << tel.size() // 3
              << tel["Jeez"] << tel["Fanny"]
              << tel["Bozo"] << std::endl; // 02345678
    return 0;
}
```

257

257

multimap

- Les multimap peuvent conserver plusieurs éléments dont les clefs sont équivalentes
- Pas d'accès direct avec l'opérateur []
- Insertion/Suppression des couples (clef,valeur) avec les fonctions membre `insert(key)` et `erase(key)`

```
std::multimap<std::string,int> telm;  
telm.insert(std::make_pair(std::string("Tonio"),06777777));  
telm.insert(std::make_pair(std::string("Lili"),03222222));  
telm.insert(std::make_pair(std::string("Tonio"),08111111));  
std::cout << telm.size(); // 3
```

- Accès aux couples (clef,valeur) à l'aide d'itérateurs et des fonctions membres `find(key)` `lower_bound(key)` et `upper_bound(key)`

258

258

Remarque générale

- Les conteneurs possèdent leurs éléments
 - insertion d'un élément = introduction d'une copie*
 - la destruction d'un conteneur s'accompagne de celle de ses éléments
 - la copie d'un conteneur entraîne celle de tous ses éléments

(*Mais on peut toujours faire des conteneurs de pointeurs!)

259

259

Les itérateurs

• Concept d'*iterator*

- désigne toute classe munie des opérateurs membres * (et si nécessaire de ->), et du test d'égalité
 - la valeur ou la référence retournée par l'opérateur * est dite "élément pointé"
- abstraction de la notion de pointeur
 - un pointeur est un cas particulier d'itérateur

260

260

- Concept d'*input iterator*
 - iterator muni de l'opérateur ++
 - accès en **lecture** à l'élément pointé (par retour de `operator *()`) (const ref ou copie)
- Concept d'*output iterator*
 - iterator muni de l'opérateur ++
 - accès en **écriture** à l'élément pointé (retour d'une référence par `operator *()`)
- Concept de *forward iterator*
 - désigne toute classe qui est à la fois un *input* et un *output iterator*

261

261

- concept de *bidirectional iterator*
 - forward iterator muni de l'opérateur --
- concept de *random access iterator*
 - forward iterator muni
 - d'une **arithmétique** similaire à celle des pointeurs (addition ou soustraction d'un entier, soustraction entre 2 itérateurs),
 - de l'opérateur [],
 - et supportant des **opérations de comparaison**

262

262

Opérateurs de déréférencement

- L'opérateur surchargé (* ou ->) s'applique à un objet de type *Iterator* et non à un pointeur

• Surcharge de l'opérateur unaire *

```
template <typename T> class Iterator  
{ T& operator *() {return blabla;} // Retour d'une référence  
                                //sur l'objet « pointé »  
                                // par l'itérateur  
  
    T* operator ->() {return blibli;} // Cas où l'objet « pointé »  
                                // a des membres  
};
```

• Remarque :

- La définition d'une conversion d'un itérateur vers un pointeur peut parfois éviter cette surcharge ainsi que celle de []

263

263

Autre exemple où on utilise une possibilité de conversion vers un pointeur pour s'économiser l'écriture de la surcharge des opérations de déréférencement (cet exemple n'a rien à voir avec les itérateurs, puisqu'il s'agit ici d'un Tableau)

```
class Tableau
{
    ...
    operator int*();
};
Tableau tab;
```

Conversion utilisée dans `*tab` ou `tab[i]`
(sans avoir surchargé `operator *` ni `[]`)

264

264

- Surcharge de l'opérateur unaire `->`

Iterateur it;
it->membre s'interprète comme

`(it.operator->())` \rightarrow membre;

– Doit renvoyer :

- un pointeur sur une classe possédant le nom de membre attendu
- ou un objet d'une classe disposant de `->` tel que ...
- ie « un truc » sur lequel \rightarrow a un sens

265

265

Surcharge de l'opérateur ++ (préfixé)

- Possibilité de donner un sens à `++z`, avec `z` de type non primitif (ex : Entier ou bien Iterateur)

– soit en surchargeant ++ en opérateur membre sans argument

```
Iterateur & Iterateur::operator++();
```

(`++z` interprété comme `z.operator++()`)

266

266

– soit en surchargeant ++ en fonction (amie) ayant pour unique argument un Iterateur

```
friend Iterateur & operator++(Iterateur &);
```

(`++z` interprété comme `operator++(z)`)

267

267

Surcharge de l'opérateur ++ (postfixé)

- Possibilité de donner un sens à `z++`, avec `z` de type non primitif (ex : Entier ou bien Iterateur)

– soit en surchargeant ++ en opérateur membre ayant un argument entier

```
Iterateur operator ++(int);
```

(`z++` interprété comme `z.operator++(0)`)

268

268

– soit en surchargeant ++ en fonction amie ayant pour arguments un Iterateur et un entier

```
friend Iterateur operator ++(Iterateur &, int);
```

(`z++` interprété comme `operator++(z,0)`)

– l'argument entier est juste un artifice

269

269

- Chaque conteneur de la STL fournit :
 - un type local d'itérateur permettant d'accéder à ses éléments, et d'en faire un parcours exhaustif
`ex: std::list<int> l1;`
`std::list<int>::iterator it;`
 - une fonction membre `begin()` renvoyant une valeur d'itérateur correspondant au type local du conteneur
 - si le conteneur est un conteneur séquentiel, `begin()` « pointe » sur le premier élément de la séquence
 - une fonction membre `end()` renvoyant un itérateur du type local au conteneur
 - si le conteneur est séquentiel, `end()` « pointe » après le dernier élément de la séquence

270

270

- Exemple :
Parcours exhaustif et séquentiel d'une liste

```
std::list<int> l;
l.push_back(3);
l.push_front(5);
l.push_back(2);
for (std::list<int>::iterator it=l.begin() ;
     it!=l.end() ; it++)
    {std::cout << *it << std::endl; }
```

Remarque :

Depuis C++ 11
`vector<int>v={1,2,3};`
`for (int &x: v) {x=4; }`

271

271

- Exemple :
Parcours d'une multimap

```
std::multimap<std::string,int> telm;
telm.insert(std::make_pair(std::string("Tonio"),06777777));
telm.insert(std::make_pair(std::string("Lili"),03222222));
telm.insert(std::make_pair(std::string("Tonio"),08111111));
std::multimap<std::string,int>::iterator it;
for (it=telm.begin() ; it!=telm.end() ; it++)
    {std::cout << (*it).first < std::endl; }
```

Parcours des éléments de clef donnée

```
for( it=telm.lower_bound("Tonio");
     it!=telm.upper_bound("Tonio");
     it++)
    {std::cout << (*it).second << std::endl; }
```

272

272

- Attention :
 - Certaines opérations sur un conteneur peuvent **invalid**er des itérateurs pointant sur des éléments de ce conteneur

(ex : extension mémoire suite à l'insertion d'un nouvel élément dans un vector : certains des éléments pointés ont pu "déménager", les itérateurs sur un deque peuvent également être invalidés)

273

273

Itérateurs d'insertion

- Itérateurs associés aux conteneurs de la STL :
 - accès aux éléments existants dans le conteneur
 - modification de ses éléments
 - mais pas d'ajout d'éléments ...
- Itérateur d'insertion :
output iterator permettant d'étendre le contenu d'un conteneur, par insertion de nouveaux éléments à l'endroit pointé (ici à la fin)


```
std::list<int> l;
std::back_insert_iterator<std::list<int>> >
it=std::back_inserter(l);
for(int i=0;i<10;i++)
    *++it= i; //equivaut l.pushback(i);
```
- Il existe également des itérateurs d'insertion au début, ou à une position donnée.

274

274

insert_iterator, quelle implantation?

Les insert-iterator peuvent aussi permettre l'insertion en une position donnée de la séquence

Code

```
insert_iterator<Container> insertIter(cont,target);
...
for (int i=1;i<10; i++)
    *insertIter+=i;
```

- cont : Conteneur dans lequel se font les insertions
- target : position avant laquelle les éléments seront insérés dans le conteneur (itérateur classique)
- Comment implanter les itérateurs d'insertions?

275

275

Retour sur les (back)_insert_iterator

Code

- ```
back_insert_iterator<Container> it(cont);
...
for (int i=1; i<10; i++)
 *it++ = i;
```
- Traduction :  
it.operator++(0).operator\*(0).operator=(i);
  - Travail déclenché par l'opérateur d'affectation : appel à cont.pushback(i) sur le conteneur cont associé à l'inserteur
  - Le déréférencement et l'incréméntation sont sans effet (return \*this;)

276  
276

276

## Itérateurs de flux

- Possibilité d'associer un itérateur à un flux de sortie (ostream), de manière à pouvoir écrire dessus

```
std::ostream_iterator<int> os(std::cout, " puis ");
//pour écrire des entiers séparés par " puis"
*os=8;
++os;
*os=9;
++os;
```

Affichage sur la sortie standard :  
8 puis 9 puis

277

277

## Itérateurs de flux

- Possibilité d'associer un itérateur à un flux de sortie (ostream), de manière à pouvoir écrire dessus

```
std::ostream_iterator<int> os(std::cout, " puis ");
//pour écrire des entiers séparés par " puis"
*os=8;
++os;
*os=9;
++os;
```

Affichage sur la sortie standard :  
8 puis 9 puis

- La classe ostream\_iterator possède un accès au flux sur lequel elle réalise de l'affichage
- Travail d'affichage réalisé par l'opérateur =

278

278

- Possibilité d'associer un itérateur à un flux d'entrée (istream) de manière à pouvoir y lire des données

```
std::istream_iterator<int> is(std::cin);
//pour lire des entiers sur l'entrée standard
int a,b;
a=*is;
++is;
b=*is;
++is;
```

279

279

- Possibilité d'associer un itérateur à un flux d'entrée (istream) de manière à pouvoir y lire des données

```
std::istream_iterator<int> is(std::cin);
//pour lire des entiers sur l'entrée standard
int a,b;
a=*is;
++is;
```

- Travail de lecture réalisé par une surcharge de l'opérateur \*

280

280

## Algorithmes

- Algorithmes applicables à différents conteneurs
- Comment?  
Accès aux éléments du conteneur, uniquement à travers des itérateurs
- De nombreux algorithmes (site web SGI) :
  - de copie (**copy**),
  - de recherche d'élément(s) (**find**, **min\_element**, **max\_element**, **search**),
  - de tri (**sort**)
  - d'opérations sur les tas (**make\_heap**, **push\_heap**, **pop\_heap**, **sort\_heap**),
  - d'application d'une fonction aux éléments d'un conteneur (**for\_each**),
  - d'opérations élémentaires sur des ensembles (**set\_union**, **includes**,...)
  - de mélange (**random\_shuffle**)
  - ...

281

281



### Exemple d'utilisation de copy

```
std::list<int> l(5,1);
std::vector<int> v(5);

// Copie des éléments de l dans v (qui en a la place!)
std::copy(l.begin(), l.end(), v.begin());

// Affichage des éléments de v
std::ostream_iterator<int> os(std::cout, " ");
std::copy(v.begin(), v.end(), os);

// Extension de v par copie de l
std::copy(l.begin(), l.end(), std::back_inserter(v));
```

282

282

### Exemple d'utilisation de find

```
std::list<std::string> l;
l.push_front("Lou");
l.push_front("Serge");
std::list<string>::iterator s;
s=std::find(l.begin(), l.end(), std::string("Anna"));
if (s != l.end())
{ ...// la string "Anna" est dans la liste à la position s;}
```

283

283

### Exemple d'utilisation de generate et de for\_each

```
void affiche(int i){std::cout << i << std::endl;}

class Aff //classe d'objets fonctions
{public :
 void operator () (int i)
 {std::cout << "int : " << i << std::endl;}
};

std::vector<int> v(10);
std::generate(v.begin(), v.end(), rand);
std::for_each(v.begin(), v.end(), affiche);
std::for_each(v.begin(), v.end(), Aff);
```

Objets fonctions  
ou fonctions =  
foncteurs

284

### Exemple d'utilisation de sort

```
std::vector<int> v(10);
std::generate(v.begin(), v.end(), rand);
std::sort(v.begin(), v.end(), std::less<int>()); // Tri croissant

#include<functional>

int poids[10] = {5,6,2,7,89,2,1,90,55,70};
sort(v.begin(), v.end(),
[pois](const int i, const int j) { return(poids[i] < pois[j]); });
//Lambda function C++11
```

285

285

### Exemple d'utilisation de transform

Affectation des éléments de la séquence s2, par application de la fonction sin aux éléments de la séquence s1

```
std::vector<int> s1(10);
std::list<int> s2(10);
std::generate(s1.begin(), s1.end(), rand);
std::transform(s1.begin(), s1.end(), s2.begin(), sin);
```

286

286

### • Exo :

```
#include <list>
#include <algorithm>
#include <iostream>
int main()
{
 std::list<int> lili(4);
 std::ostream_iterator<int> os(std::cout, "yo ");
 std::list<int>::iterator it=lili.begin();
 std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 *it=1; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 *it+=2; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 *it+=3; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 std::insert_iterator<std::list<int> > ins(lili, it);
 *ins=4; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 *ins=8; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 *ins+=5; std::copy(lili.begin(), lili.end(), os); std::cout<<"\n";
 return 0;
}
```

287

287

0yo 0yo 0yo 0yo  
1yo 0yo 0yo 0yo  
2yo 0yo 0yo 0yo  
2yo 3yo 0yo 0yo  
2yo 3yo 4yo 0yo 0yo  
2yo 3yo 4yo 8yo 0yo 0yo  
2yo 3yo 4yo 8yo 5yo 0yo 0yo

288

288

## Effective C++

- Du nom des ouvrages de Scott Meyer
- Option de compilation `-Weffc++`

289

289