

D'autres conteneurs

- `#include<slist>` : 'slist', liste chaînée simple.
- `#include<stack>` : 'stack', dérivé de 'deque'. Méthodes ajoutées :
 - `push(val)` -> ajout
 - `pop()` -> retrait + lecture du sommet
 - `top()` -> lecture du sommet.

D'autres conteneurs (2)

- `#include <queue>` : 'queue', dérivé de 'deque'. Méthodes ajoutées :
 - `push(val)`
 - `pop()`.

Exemple

```
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    stack<int> pile;
    pile.push(1);
    pile.push(2);
    cout << pile.top() << " == "
         << pile.pop() << endl;

    return 0;
}
```

Conteneurs et itérateurs

- Les autres types d'itérateurs fournis par les conteneurs sont :
 - itérateur sur des valeurs constantes
 - itérateur inversé
 - itérateur inversé sur des valeurs constantes.
- On emploie 'classiquement' `begin()` et `end()`.
- On emploie `rbegin()` et `rend()` pour les variantes inversées.

Exemples

```
// Déclarations
list<int>::iterator it;
list<int>::const_iterator cit;
list<int>::reverse_iterator rit;
list<int>::const_reverse_iterator rcit;

// Itérateur constant
for (cit = l.begin(); cit != l.end(); ++cit)
{
    ...
}

// Itération inversée
for (rit = l.rbegin(); rit != l.rend(); ++rit)
{
    ...
}
```

Les algorithmes de la STL

- Ce sont des briques logicielles génériques capables d'opérer sur divers conteneurs de la STL.
- Ce sont des templates.
- Ils sont hautement paramétrables.
- Les itérateurs font le lien avec les conteneurs.
- Ils sont fiables.
- Réduction drastique du code écrit.

Les algorithmes de la STL (2)

- `#include <algorithm>`
- De façon générale, ils utilisent un itérateur de début et un itérateur de fin pour parcourir le conteneur.
- Il y a des algorithmes qui modifient les conteneurs et d'autres qui ne le font pas.
- Pour ce cours, nous verrons des algorithmes 'simples'.

“non-mutating algorithms”

binary_search

- Recherche binaire dans un conteneur.
- Basée sur '<'.
'<' est un opérateur de comparaison qui renvoie true si l'élément à gauche est strictement inférieur à l'élément à droite.
- Renvoie un 'bool'.

```
list<int> l;  
l.push_back(1);  
l.push_back(5);  
l.push_back(10);  
l.push_back(12);  
  
bool res = binary_search(l.begin(), l.end(), 10);
```

count

- Compte le nombre d'éléments égaux à une valeur donnée.

```
multiset<int> ensemble;

for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 100; ++j)
        ensemble.insert(j);

cout << count(ensemble.begin(),
              ensemble.end(),
              50)
      << endl;
```

equal

- Comparaison élément à élément de deux conteneurs (renvoie bool).
- Les conteneurs **peuvent** être différents (ex: liste et vecteur) !

```
list<int> l;  
vector<int> v;  
  
(...)  
  
bool res = equal(l.begin(), l.end(),  
                v.begin(), v.end());
```

find

- Recherche un élément.
- Renvoie un itérateur sur le premier élément égal à la valeur recherchée.

```
deque<double> vals;  
vals.push_back(1); vals.push_back(2);  
vals.push_back(3); vals.push_back(4);  
  
deque<double>::iterator it1, it2;  
it1 = find(vals.begin(), vals.end(), 3);  
it2 = find(vals.begin(), vals.end(), 5);  
  
// it1 -> vals[2]  
// it2 -> vals.end();
```

“Sorting algorithms”

min / max

- min et max renvoient le min/max entre deux valeurs.
- min_element et max_element opèrent sur un conteneur.

```
int i = max(3, 5);
```

```
vector<int> v(3);
```

```
v[0] = 1; v[1] = 2; v[2] = 3;
```

```
i = max_element(v.begin(), v.end());
```

```
int tab[] = {1, 2, 3};
```

```
i = min_element(tab, tab + 3);
```

sort / is_sorted

- 'sort' effectue un tri (opérateur <).
- Cet algorithme modifie un conteneur.
- 'is_sorted' renvoie un 'bool' pour dire si un conteneur est trié.

```
int tab[] = {3, 8, 1, 2, 9, 0, 28, 19};  
  
cout << is_sorted(tab, tab + 8) << endl; // false  
  
sort(tab, tab + 8);  
  
cout << is_sorted(tab, tab + 8) << endl; // true
```

includes

- Test d'inclusion entre éléments de conteneurs (qui peuvent être différents) (renvoie un 'bool').
- Les conteneurs **doivent** être triés !

```
set<int> s1;  
set<int> s2;
```

```
(...)
```

```
bool s2_IncluDans_s1 = includes(s1.begin(), s1.end(),  
                               s2.begin(), s2.end());
```

“Mutating algorithms”

copy

- Effectue une copie vers un autre conteneur.
- Forme = itérateurs début/fin + itérateur d'écriture.
- Il faut que le conteneur de destination soit de taille suffisante !

```
vector<int> v1(2);  
vector<int> v2;  
  
v1[0] = 1; v1[1] = 2;  
v2.resize(v1.size());  
copy(v1.begin(), v1.end(), v2.begin());
```

Avant de continuer ...

- Nous venons de voir que la fonction 'copy' a besoin d'un itérateur d'écriture (output iterator) afin d'écrire les éléments dans le conteneur de destination.
- Problème : il faut tenir compte des problèmes de taille. En effet, 'copy' ne va pas ajouter d'éléments au conteneur de destination.
- Si les tailles ne sont pas bonnes : **crash**.
- Solution : utiliser un itérateur qui insère des éléments.

Itérateurs évolués

- `#include <iterator>`
- `istream_iterator` et `ostream_iterator` permettent d'itérer sur des flux. Applications : 'copier' des valeurs vers un fichier, 'copier' des valeurs depuis un fichier, ...
- `front_insert_iterator` et `back_insert_iterator` permettent d'insérer automatiquement des éléments à un conteneur qui supporte les opérations d'ajout propres aux séquences (`push_front` / `push_back`)
- `insert_iterator` permet d'insérer depuis un itérateur quelconque dans un conteneur supportant l'opération d'insertion (`insert`).

Mise en oeuvre

```
// Initialisations
list<int> l;
deque<int> d;
for (int i = 0; i < 100; ++i) l.push_back(i);

// Copie l -> d
back_insert_iterator<deque<int> > bi(d);
copy(l.begin(), l.end(), bi);

// Affichage de d
ostream_iterator<int> oi(cout, "\n");
copy(d.begin(), d.end(), oi);

// Sauvegarde 'inversée' sur un fichier, séparateur = ' '
ofstream fic("test.txt");
ostream_iterator<int> ofi(fic, " ");
copy(d.rbegin(), d.rend(), ofi);
fic.close();
```

Mise en oeuvre (2)

```
// Initialisations
```

```
list<string> l;  
ifstream fic("fichier.txt");
```

```
// Itérateurs
```

```
istream_iterator<string> debut(fic);  
istream_iterator<string> fin;  
back_insert_iterator<list<string> > bi(l);
```

```
// Lecture du fichier -> l
```

```
copy(debut, fin, bi);
```

```
// Lecture du fichier -> l
```

```
copy(istream_iterator<string>(fic),  
    istream_iterator<string>(),  
    back_insert_iterator<list<string> >(l));
```

Version
détaillée

Version
courte

copy_n

- Copie 'n' éléments à partir d'un itérateur de départ.

```
list<double> l1;  
list<double> l2;
```

```
(...)
```

```
copy_n(l1.begin(), l1.size(), l2.begin());
```

merge

- Fusionne 2 ensembles de valeurs **triées** en 1 seul ensemble trié.

```
// Deux tableaux ...
int t1[] = {1, 3, 5, 7};
int t2[] = {2, 4, 6, 8};

// Fusion dans une liste ...
list<int> l;
merge(t1, t1 + 4, t2, t2 + 4,
      back_insert_iterator<list<int> >(l));

// Fusion à l'écran ...
merge(t1, t1 + 4, t2, t2 + 4,
      ostream_iterator<int>(cout, " "));
```

replace

- Remplace les occurrences d'une valeur par une autre.

```
// Création (avec une erreur)
list<string> tokens;
tokens.push_back("Schumacher");
tokens.push_back("est");
tokens.push_back("le meilleur.");

// Rectification de l'erreur
replace(tokens.begin(), tokens.end(),
        "Schumacher", "Montoya");

// Affichage
copy(tokens.begin(), tokens.end(),
      ostream_iterator<string>(cout, " "));
```

Calcul de sommes

- `accumulate` prend une valeur de départ (typiquement 0). Il effectue la somme des éléments.
- Nous verrons la semaine prochaine comment le paramétrer pour faire autre chose que des sommes ...

```
int tab[] = {1, 2, 3, 4, 5};  
  
cout << accumulate(tab, tab + 5, 0) << endl;
```

Opérations ensemblistes

- `set_union`, `set_intersection`, `set_difference`.
- Forme :
 - itérateurs du conteneur 1
 - itérateurs du conteneur 2
 - itérateur pour écrire le résultat.

```
set<double> s1;  
set<double> s2;  
(...)  
  
set<double> res;  
insert_iterator<set<double> > ii(res, res.begin());  
set_intersection(s1.begin(), s1.end(),  
                s2.begin(), s2.end(), ii);
```

Quelques autres ...

- `reverse`

```
int tab[] = {1, 2, 3, 4, 5};  
reverse(tab, tab + 5);
```

- `fill`, `fill_n`

```
fill(tab, tab + 5, 0);  
vector<int> v(5);  
fill_n(v.begin(), 5, 0);
```

- `unique`

```
set<int> s;  
s.insert(10); s.insert(10); s.insert(10);  
unique(s.begin(), s.end());
```