

Présentation de la STL

- STL = Standard Template Library.
- 3 axes:
 - les conteneurs
 - les algorithmes
 - les itérateurs.
- Ces 3 axes de la STL permettent de développer rapidement des algorithmes complexes en assemblant des briques de base génériques.
- Ex: fusionner 2 listes en les triant et placer le résultat dans un fichier: **1 ligne ...**

Présentation de la STL (2)

- L'emploi des templates rend possible de telles constructions.
- Il n'existe pas d'équivalents dans les langages 'dits récents' (Java, C#). La STL rend le C++ unique.
- Les langages récents offrent des conteneurs évolués, des itérateurs (limités) et quelques algorithmes de base, sans jamais égaler la STL.
- <http://www.sgi.com/tech/stl/>

Les conteneurs

- Templates de types de données abstraits (TDA).
- On va pouvoir paramétrer divers TDAs (vecteurs, listes, dictionnaires, ...) pour n'importe quel type.
- Les conteneurs sont des templates de classes.
- Les conteneurs de la STL sont éprouvés et fiables. Réimplémenter ceux-ci est coûteux et favorise l'introduction d'erreurs. Typiquement, 95% du code sera juste. Employer la STL est bénéfique.

Les conteneurs (2)

- Chaque conteneur possède ses propres avantages inconvénients. On retiendra les critères suivants:
 - coût des insertions / suppressions
 - coût des accès aux éléments.
- On choisira son conteneur en fonction du besoin de façon à minimiser la somme des coûts des opérations effectuées sur celui-ci.
- Voir la complexité algorithmique.

Les vecteurs (vector)

- `#include <vector>`
- Méthodes importantes:
 - `size()`, `empty()`, `clear()`, `resize(taille)`
 - `front()`, `back()`
 - `push_front(val)`, `push_back(val)`
 - `pop_front()`, `pop_back()`
- Opérateurs: `=`, `==`, `<`, `[]`.

Les vecteurs (vector)

- Avantages:
 - accès aléatoire aux éléments: $O(1)$
 - ajout / suppression d'éléments à la fin: $O(1)$
 - gestion de la mémoire automatique.
- Inconvénients:
 - ajout / suppression au début et au milieu: $O(n)$ (nécessite de réallouer tout le tableau).

Vecteurs: exemple

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

using namespace std;

int main()
{
    vector<int> v1;
    v1.push_back(1); v1.push_back(2); v1.push_back(3);

    vector<int> v2(3); // Taille initiale -> 3
    v2[0] = 1; v2[1] = 2; v2[2] = 3;

    cout << (v1 == v2) ? "Ok" : "Appeller l'asile"
         << endl;

    return 0;
}
```

Les listes (list)

- `#include <list>`
- Méthodes importantes:
 - `size()`, `empty()`, `clear()`, `resize(taille)`
 - `front()`, `back()`
 - `push_front(val)`, `push_back(val)`
 - `pop_front()`, `pop_back()`, `remove(val)`
- Opérateurs: `=`, `==`, `<`.
- Spécialisations: `sort()`, `unique()`, `reverse()`.
- Insertions / suppressions au milieu: nécessite d'avoir vu les itérateurs ...

Les listes (list) (2)

- Avantages:
 - liste doublement chaînée
 - ajout / suppression: $O(1)$.
- Inconvénients:
 - accès aux éléments: $O(n)$
 - emploi d'opérations spécialisées à la place d'algorithmes STL (sort, unique).

Les listes: exemple

```
#include <list>
#include <iostream>

using namespace std;

int main()
{
    list<int> l;
    l.push_back(2);
    l.push_back(1);
    l.push_back(3);
    l.sort();
    l.reverse();

    // Affiche '3 1'
    cout << l.front() << ' ' << l.back() << endl;

    return 0;
}
```

'Double end queues' (deque)

- `#include <deque>`
- Se comporte comme un vecteur, à la différence que l'ajout et la suppression d'éléments au début se fait en temps constant ($O(1)$).
- 'deque' est parfois plus avantageux que 'vector', surtout dans le cas où l'on a besoin d'un accès aux éléments efficace et que l'on ajoute/supprime toujours aux extrémités.

Deque: exemple

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    deque<int> v1;
    v1.push_back(1); v1.push_back(2); v1.push_back(3);

    deque<int> v2(3); // Taille initiale -> 3
    v2[0] = 1; v2[1] = 2; v2[2] = 3;

    cout << (v1 == v2) ? "Ok" : "Appeller l'asile"
         << endl;

    return 0;
}
```

Les ensembles (set)

- `#include <set>`
- Chaque valeur d'un ensemble est unique.
- Méthodes utiles:
 - `insert(val)`, `erase(val)`
 - `find(val)` (renvoie un itérateur ...)
 - `clear()`
 - `size()`, `empty()`.
- Opérateurs: `=`, `==`, `<`.

Les ensembles (set) (2)

- Les opérations sont toutes en $O(n)$ (ajout, suppression, accès).
- En interne, les éléments sont toujours triés par ordre croissant.

Les ensembles: exemple

```
#include <set>
#include <iostream>

using namespace std;

int main()
{
    set<int> ens;
    ens.insert(1); ens.insert(1);
    ens.insert(2); ens.insert(2);
    ens.erase(2);

    // Affiche '1'
    cout << ens.size() << endl;

    return 0;
}
```

Les dictionnaires (map)

- `#include <map>`
- Association clé/valeur (clés uniques).
- Méthodes utiles:
 - `size()`, `empty()`, `clear()`
 - `insert(val)`, `erase(val)`, `find(val)`.
- Opérateurs: `=`, `==`, `<`, `[]`.
- `[]` va nous permettre d'accéder à un élément par sa clé. Il va aussi nous permettre l'ajout.

Les dictionnaires: exemple

```
#include <map>
#include <iostream>

using namespace std;

int main()
{
    // Paramètre 1: type clé, paramètre 2: type valeur
    map<string, string> annu;

    agenda["Aristide"] = "06-12-34-56-78";
    agenda["Paul"] = "03-86-38-12-34";

    cout << "Numéro de Paul: " << agenda["Paul"]
         << endl;

    return 0;
}
```

Variantes sur les 'set' et 'map'

- 'set' garanti l'unicité des valeurs.
- 'map' garanti les clés.
- Si l'on veut au contraire des valeurs ou clés multiples, il faut employer 'multiset' et 'multimap'.
- Définis dans les mêmes en-têtes que 'set' et 'map'.
- L'emploi est identique.

Les itérateurs

- Il s'agit d'une généralisation des pointeurs. Ce sont des objets qui 'pointent' sur d'autres objets.
- Ils servent de lien entre les conteneurs et les algorithmes (génériques) car ils permettent de s'abstraire du TDA.
- Les itérateurs permettent de se déplacer dans un TDA comme avec l'arithmétique des pointeurs.
- Il y a différents types d'itérateurs.

Les types d'itérateurs

- Input iterators: n'autorisent que la lecture.
- Output iterators: n'autorisent que l'écriture.
- Ces deux premières classes n'autorisent pas l'emploi d'algorithmes où plusieurs passes sont nécessaires.
- Forward iterators: input + output + possibilité de passer à un élément suivant (**++**).
- Bidirectional iterators: forward + possibilité de passer à un élément précédent (**--**) (**ex: list**).
- Random iterators: autorisent toutes les opérations de l'arithmétique des pointeurs (**++**, **+=**, **...**) (**ex: vector**).

Débuter avec les itérateurs

- Chaque conteneur fourni un type d'itérateur. Ex: `vector<int>::iterator`
- Chaque conteneur fourni deux itérateurs, accessibles par les méthodes:
 - `begin()`: itérateur sur le premier élément
 - `end()`: itérateur sur le premier élément qui n'appartient pas au conteneur.
- `end() <->` premier élément hors d'un tableau. Il peut être employé comme condition d'arrêt dans un parcours des éléments d'un conteneur.

Exemple sur les itérateurs

```
#include <vector>

using namespace std;

int main()
{
    list<int> l;
    l.push_back(1); (...) l.push_back(100);

    list<int>::iterator it;
    for (it = l.begin(); it != l.end(); ++it)
    {
        (*it) = (*it) + 1;
    }

    return 0;
}
```