

LE LANGAGE C++ MASTER I LA STL

Jean-Baptiste.Yunes@univ-paris-diderot.fr
U.F.R. d'Informatique
Université Paris Diderot - Paris 7

2019–2020

- La STL : une bibliothèque de modèles de classes et fonctions
 - conteneurs
 - algorithmes et objets fonction
 - itérateurs et allocateurs
 - chaînes
 - flux



LES CONTENEURS

- `<vector>` : tableaux dynamiques 1D
- `<list>` : listes doublement chaînées
- `<deque>` : files à double accès
- `<queue>` : files simples
- `<stack>` : piles
- `<map>` : tableaux associatifs
- `<set>` : ensembles
- `<bitset>` : ensemble de booléens ou de bits

- `<vector>` : c'est un `template` du namespace `std`;
- il contient des types
 - `value_type` (type utilisé à l'instanciation du `vector`)
 - `iterator` : qui se **comporte comme** un `pointeur value_type *`
 - + beaucoup d'autres...

- `<vector>` : c'est un `template` du namespace `std`;
- il contient des fonctions membres permettant d'obtenir des itérateurs
 - `begin()` : pointe sur le 1er élément
 - `end()` : pointe sur le suivant du dernier
 - `rbegin()` : pointe sur le dernier
 - `rend()` : pointe sur le précédent du premier

- `<vector>` : c'est un `template` du namespace `std`;
- il contient des fonctions et opérateurs d'accès aux éléments
 - `operator[] (size_type)` : accès non contrôlé
 - `at (size_type)` : accès contrôlé
(`out_of_range`)
 - `front ()` : premier élément
 - `back ()` : dernier élément

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

using namespace std;

int main()
{
    vector<string> v;
    v.resize(2);
    v[0] = "un";
    v[1] = "deux";
    cout << v[0] << ', ' << v[1] << endl;
    v[2] = "trois";
    return 0;
}
```



```
$ ./test
un,deux
Segmentation fault
$
```



```
#include <un,deux
#include <terminate
using namespace std;
int main()
{
    vector<string> v;
    v.resize(2);
    v[0] = "un";
    v[1] = "deux";
    cout << v[0] << ', ' << v[1] << endl;
    v.at(2) = "trois";
    return 0;
}
```

```
$ ./test
```

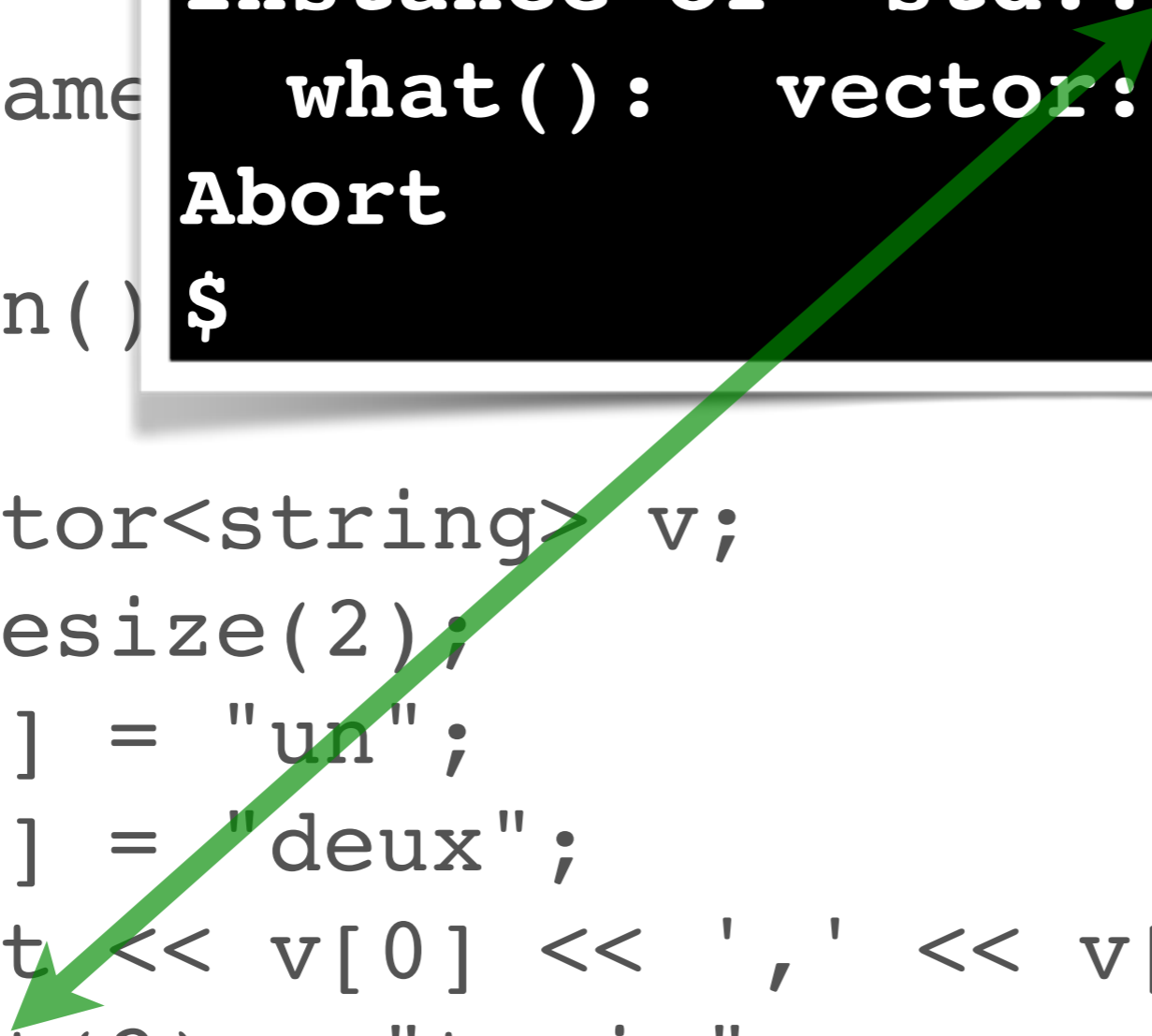
```
un,deux
```

```
terminate called after throwing an
instance of 'std::out_of_range'
```

```
what(): vector::_M_range_check
```

```
Abort
```

```
$
```



- `<vector>` : c'est un `template` du namespace `std`;
- il contient des constructeurs
 - `vector<T>(size_type n, T val=T())`
 - créé un vecteur de taille `n`, les éléments tous initialisés à `val`
 - `vector<T>(iterator<T> i1, iterator<T> i2)`
 - créé un vecteur avec les éléments `[i1, i2[`
 - `vector<T>(const vector<T> &v)`
 - créé un vecteur par copie

- `<vector>` : c'est un `template` du namespace `std`;
- il contient des opérateurs/fonctions d'affectation
 - `operator=(const vector &)`
 - `assign(iterator i1, iterator i2)`
 - `assign(size_type n, const T &val)`

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

int main() {
    int t[] = { 1, 2, 3, 4 };
    std::vector<int> v;
    v.assign(t+1,t+4);
    for (
        std::vector<int>::iterator it = v.begin();
        it<v.end();
        it++) {
        std::cout << *it << std::endl;
    }
}
```

```
$ ./test
2
3
4
$
```

- `<vector>` : c'est un `template` du namespace `std` et qui peut-être vu comme une pile
 - il contient des fonctions de manipulation de pile
 - `push_back(const T&);`
 - `pop_back();` : attention `pop_back` ne renvoie pas la valeur!!! Il faut consulter la valeur avant de dépiler (`back()`).

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

using namespace std;

int main()
{
    vector<string> maPile;
    maPile.push_back("un");
    maPile.push_back("deux");
    maPile.push_back("trois");
    maPile.push_back("quatre");
    while (!maPile.empty()) {
        cout << maPile.back() << endl;
        maPile.pop_back();
    }
    return 0;
}
```

```
$ ./test
quatre
trois
deux
un
$
```

- `<vector>` : c'est un `template` du namespace `std` et qui peut-être vu comme une liste
 - il contient des fonctions de manipulation de liste
 - `insert(iterator position, const T &element)` ajoute l'*élément* avant la *position*
 - `insert(iterator position, size_type n, const T&element)` ajoute *n* copies de l'*élément*
 - `erase(iterator position)` supprime l'élément à la *position* indiquée
 - `erase(iterator p1, iterator p2)` efface la séquence `[p1, p2[`
 - `clear()` efface tout

- `<vector>` : c'est un `template` du namespace `std`
 - il contient des attributs de taille
 - `size_type size()` : nombre d'éléments
 - `bool empty()` : est-il vide ?
 - `size_type max_size()` : + grande taille possible
 - `void resize(size_type n, T val=T())`
redimensionne avec initialisation des nouveaux
 - `size_type capacity()` : capacité actuelle avant redimensionnement automatique
 - `void reserve(size_type n)` : réserve de la place pour n éléments sans les initialiser

- `<vector>` : c'est un template du namespace `std`
- il contient des fonctionnalités annexes
 - `swap(vector &)` : pour échanger deux vecteurs
 - `bool operator==(const vector<T> &v1, const vector<T> &v2)`
 - `bool operator<(const vector<T> &v1, const vector<T> &v2)` : ordre lexicographique

- À quoi sert, par exemple, `value_type` ???
 - permet d'écrire des templates indépendants des types inclus dans d'autres types!

```
template <typename T>
typename T::value_type f(T v) {
    typename T::value_type x;
    x = 0;
    return x;
}
```

Voilà un bon usage de ce mot-clé

```
class A {
public:
    int value;
    A(int v=666) { value = v; }
};
```

```
ostream &operator<<(ostream &o, const A& a) {
    o << a.value;
    return o;
}
```

```
int main() {
    vector<A> v(5,A());
    cout << v[3].value << endl;
    cout << f(v) << endl;
    return 0;
}
```

LES SÉQUENCES

- les séquences sont des structures ordonnées, on y trouve :
 - `vector` : plutôt un tableau
 - `list` : optimisé pour l'insertion/suppression mais pas d'indexage
 - `deque` : optimisée pour les parcours dans les deux sens
- et les dérivées
 - `stack`
 - `queue`
 - `priority_queue`

LES CONTENEURS ASSOCIATIFS

- `<map>` est une simple séquence de paires (clé,valeur)
- les clés doivent être comparables (opérateur `<`)
 - sinon c'est `hash_map`
- les types inclus sont :
 - `key_type`
 - `map_type`
 - `pair<key_type, map_type> value_type`

- `<map>` est une simple séquence de paires (clé,valeur)
 - `iterator find(key_type &)`
 - `size_type count(key_type &)`
 - `pair<iterator,bool> insert(const value_type &)`
 - `void erase(iterator)`
 - `size_type erase(const key_type &)`
 - `void clear()`
 - `size_type size()`
 - `size_type max_size()`
 - `bool empty()`

Un type défini par
le template!

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

using namespace std;

int main()
{
    map<string,int> table;
    table["un"] = 1;
    table["deux"] = 2;
    table["trois"] = 3;
    map<string,int>::iterator i = table.find("deux");
    if (i==table.end()) cout << "pas ";
    cout << "trouve" << endl;
    return 0;
}
```

```
$ ./test
trouve
$
```

- `<set>` est une `map` réduite aux seules clés

ALGORITHMES ET OBJETS FONCTION

LES OBJETS FONCTION
LES FONCTEURS

- En surchargeant l'opérateur fonctionnel $()$ on a déjà vu comment obtenir des objets qui pouvaient être considérés comme des fonctions
 - ce sont des foncteurs ou objets fonction
- La STL en fait une utilisation intensive en liaison avec les algorithmes

- La STL définit quelques grandes classes de foncteurs :
 - les fonctions unaires :
 - `template <class Arg, class Ret>`
`unary_function`
 - les fonctions binaires
 - `template <class A1, class A2, class Ret>`
`binary_function`

```
#include <algorithm>
using namespace std;

class successor :
    public unary_function<int,int> {
public:
    int operator()(int v) {
        return v+1;
    }
};

int main() {
    successor s;
    cout << s(4) << endl;
    return 0;
}
```

```
$ ./test
5
$
```

- La STL définit quelques foncteurs dont
 - les prédicats qui sont simplement des foncteurs renvoyant un booléen
 - `equal_to()` / `not_equal_to()`
 - `greater()` / `less()` / `greater_equal()` / `less_equal()`
 - `logical_and()` / `logical_or()` / `logical_not()`


```
#include <algorithm>
using namespace std;

class A {
private:
    int value;
public:
    A(int value) { this->value = value; }
    bool operator==(const A &a) const {
        return value==a.value;
    }
};

int main() {
    equal_to<int> egal;
    cout << egal(4,4) << endl;
    equal_to<A> egal2;
    cout << egal2(A(4),A(444)) << endl;
    return 0;
}
```

```
$ ./test
1
0
$
```

- La STL définit quelques foncteurs dont
 - les fonction arithmétiques
 - `plus()` / `minus()`
 - `multiplies()` / `divides()`
 - `modulus()`
 - `negate()`
 - On trouve aussi les liaisons/adaptateurs/
inverseurs

LES ALGORITHMES

- Les algorithmes qui ne modifient pas les séquences
 - `for_each()`
 - `find()`
 - `find_if()`
 - `find_first_of()`
 - `adjacent_find()`
 - `count()`
 - `count_if()`
 - `mismatch()`
 - `equal()`
 - `search()`
 - `find_end()`

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

using namespace std;

class print : public unary_function<int,void> {
private:
    ostream &o;
public:
    print(ostream &o) : o(o) { }
    void operator()(int v) { o << v; }
    void endl() { o << endl; }
};

int main()
{
    vector<int> v(3);
    v[0] = 0; v[1] = 1; v[2] = 2;
    print printer(cout);
    for_each(v.begin(),v.end(),printer);
    printer.endl();
    return 0;
}
```

```
$ ./test
012
$
```

- Bien que ces algorithmes soient classés dans ceux qui ne modifient pas la séquence, il est à noter que cela n'interdit pas de modifier les éléments de la séquence...
 - Mais on devrait utiliser `transform()`

```
class print : public unary_function<int,void> {
private:
    ostream &o;
public:
    print(ostream &o) : o(o) { }
    void operator()(int v) { o << v; }
    void endl() { o << endl; }
};
```

```
void addOne(int &v) { v++; }
```

```
int main()
{
    vector<int> v(3);
    v[0] = 0; v[1] = 1; v[2] = 2;
    print printer(cout);
    for_each(v.begin(),v.end(),printer); printer.endl();
    for_each(v.begin(),v.end(),addOne);
    for_each(v.begin(),v.end(),printer); printer.endl();
    return 0;
}
```

```
$ ./test
012
123
$
```

```
class commencePar : public unary_function<string,bool> {
private:
    char c;
public:
    commencePar(char c) : c(c) { }
    bool operator()(string s) { return s[0]==c; }
};
```

```
int main()
{
    vector<string> v(4);
    v[0] = "abracadabra"; v[1] = "blablabla";
    v[2] = "c'est nul"; v[3] = "dodo";
    commencePar commenceParC('c');
    vector<string>::iterator i;
    i = find_if(v.begin(),v.end(),commenceParC);
    if (i==v.end()) {
        cout << "y'a rien" << endl;
    } else {
        cout << *i << endl;
    }
    return 0;
}
```

```
$ ./test
c'est nul
$
```


- Les algorithmes qui modifient les séquences
 - `transform()`
 - `copy()` / `copy_backward()`
 - `swap()` / `iter_swap()`
 - `replace()` / `replace_if()` / `replace_copy()` / `replace_copy_if()`
 - `fill()` / `fill_n()`
 - `generate()` / `generate_n()`
 - `remove()` / `remove_if()` / `remove_copy()` / `remove_copy_if()`
 - `unique()` / `unique_copy()`
 - `reverse()` / `reverse_copy()`
 - `rotate()` / `rotate_copy()`
 - `random_shuffle()`

```

#include <vector>
#include <iostream>

void print(const std::vector<int> &v) {
    for (std::vector<int>::const_iterator it =
it<v.end(); it++) {
        std::cout << *it << ' ';
    }
    std::cout << std::endl;
}

int main() {
    int t[] = { 1, 2, 3, 4, 5, 6, 7 };
    std::vector<int> v;
    v.assign(t,t+7);
    print(v);
    std::random_shuffle(v.begin(),v.end());
    print(v);

    std::random_shuffle(t,t+7);
    for (int i=0; i<7; i++) {
        std::cout << t[i] << std::endl;
    }
}

```

```

$ ./test
1 2 3 4 5 6 7
5 3 7 2 6 4 1
3
5
6
2
4
1
7
$

```

- Les tris
 - `sort()`
 - `stable_sort()`
 - `partial_sort()` / `partial_sort_copy()`
 - `nth_element()`
 - `lower_bound()` / `upper_bound()`
 - `equal_range()`
 - `binary_search()`
 - `merge()` / `inplace_merge()`
 - `stable_partition()`

```

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

class print : public unary_function<string, void>
public:
    void operator()(string v) {
        cout << v << endl;
    }
};

int main()
{
    vector<string> v(3);
    v[0] = "c'est nul"; v[1] = "abracadabra"; v[2] = "bool";
    print printer;
    for_each(v.begin(), v.end(), printer);
    sort(v.begin(), v.end());
    for_each(v.begin(), v.end(), printer);
    return 0;
}

```

```

$ ./test
c'est nul
abracadabra
bool
abracadabra
bool
c'est nul
$

```

version avec comparateur
implicite : operateur <

```

class print : public unary_function<string, void>
public:
    void operator()(string v) {
        cout << v << endl;
    }
};

bool comparaison(string s1, string s2) {
    return s1[2] < s2[2];
}

int main()
{
    vector<string> v(3);
    v[0] = "c'est nul"; v[1] = "abracadabra"; v[2] = "bool";
    print printer;
    for_each(v.begin(), v.end(), printer);
    sort(v.begin(), v.end(), comparaison);
    for_each(v.begin(), v.end(), printer);
    return 0;
}

```

```

$ ./test
c'est nul
abracadabra
bool
c'est nul
bool
abracadabra
$

```

version avec fonction
(ou binary-function)

- Les opérations ensemblistes
 - `includes()`
 - `set_union()`
 - `set_intersection()`
 - `set_difference()`
 - `set_symmetric_difference()`
- Les comparaisons
 - `min()`
 - `max()`
 - `min_element()`
 - `max_element()`
 - `lexicographical_compare()`

- Les permutations
 - `next_permutation()`
 - `prev_permutation()`
- Les tas
 - `make_heap()`
 - `push_heap()`
 - `pop_heap()`
 - `sort_heap()`

ITÉRATEURS ET ALLOCATEURS

- Nous n'en dirons que le strict que minimum
- Les itérateurs permettent de se déplacer dans une séquence
 - de manière similaire à l'arithmétique des pointeurs
- Les allocateurs correspondent aux classes qui permettent de réaliser des allocations
 - il s'agit d'abstractions permettant de détacher les constructions de la STL des contingences matérielles

LES CHÂÎNES



- **string** est une simple séquence de caractères...
 - elle est toutefois différente d'une séquence ordinaire
 - car certaines opérations désirées sont particulières
- Noter que les méthodes disponibles sont bien trop nombreuses pour être exposées ici
 - mais aucune n'est particulièrement surprenante, donc

consultez la documentation!

LES FLUX

- Les deux classes de bases sont
 - `ios_base`, et sa sous-classe
 - `ios`
- Toutes deux ne sont pas directement instanciables mais contiennent essentiellement les attributs et actions de base de la gestion des flux :
 - état du flux
 - propriétés de formatage du flux

- Les fonctions membres les plus importantes sont :
 - `bool good()` : tout est ok
 - `bool bad()` : le flux est dans un état irrécupérable
 - `bool fail()` : une erreur est apparue, consulter `bad()` pour connaître sa gravité
 - `bool eof()` : la fin du flux a été atteinte
 - `void clear()` : on remet l'état à une valeur par défaut
 - `sync_with_stdio(bool b=true)` // statique

- Sous la classe `ios` on trouve deux classes :
 - `istream`
 - la classe de base de tous les flux de lecture
 - `ostream`
 - la classe de base de tous les flux d'écriture

- pour `istream` on `trouve`

- `get`
- `getline`
- `ignore`
- `peek`
- `read`
- `>>`
- `tellg`
- `seekg`

- pour `ostream` on `trouve`

- `put`
- `write`
- `<<`
- `tellp`
- `seekp`
- `flush`

- Sous la classe `istream` on trouve

- `iostream`

- `ifstream`

- `istringstream`

- Sous la classe `ostream` on trouve

- `iostream`

- `ofstream`

- `ostringstream`

- Sous la classe `iostream` on trouve

- `fstream`

- `stringstream`

- Ces différentes classes se distinguent essentiellement par la source (pour les `istream`) ou par la destination (pour les `ostream`) de l'opération :
- `fstream/ifstream/ofstream` : fichier
- `stringstream/istringstream/stringstream` : mémoire
- donc par leur constructeur...

- Quelques exemples :

```
ofstream sortie("monFichier.txt");  
sortie << "Bonjour" << endl;  
sortie.write("Cuicui", 6);  
sortie.close();
```

```
ifstream entree("monFichier.txt");  
entree >> s;  
cout << s << endl;  
entree.close();
```

- Quelques exemples :

```
int d;  
string s = "28192";  
istringstream entree(s);  
entree >> d;  
cout << d << endl;  
entree.close();
```

```
ostringstream sortie();  
sortie << 2009;  
cout << sortie.c_str() << endl;  
sortie.close();
```

- Le formatage consiste à envoyer dans le flux un contenu qui n'est pas destiné à l'affichage mais simplement à modifier le comportement du flux
- il s'agit des manipulateurs de flux
 - `<iomanip>`

- boolalpha / noboolalpha
- dec / hex / oct
- endl / ends
- fixed / scientific
- flush
- internal
- left / right
- setbase
- setfill
- setiosflags / resetiosflags
- setprecision
- setw
- showbase / noshowbase
- showpoint / noshowpoint
- showpos / noshowpos
- skipws / noskipws
- unitbuf / nounitbuf
- uppercase / nouppercase
- ws

```
#include <iomanip>
```

```
bool b = true;
```

```
cout << boolalpha << b << hex << 12;
```

```
cout << setw(12) << right << setfill('_');
```

```
cout << 6789 << endl;
```

```
$ ./test
truec_____1a85
$
```

LES ITÉRATEURS DE FLUX

- Il existe deux itérateurs de flux :
 - `ostream_iterator`
 - `istream_iterator`
- ils servent à itérer une opération d'entrée ou de sortie sur une séquence...
 - il s'agit (par exemple) d'afficher chaque élément en utilisant un séparateur


```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
```

```
using namespace std;
```

```
int main()
```

```
{
```

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

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

```
    copy(v.begin(),
```

```
          v.end(),
```

```
          ostream_iterator<int>(cout, ":"));
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

```
$ ./test
```

```
0:1:2:
```

```
$
```