La **programmation orientée objet** et le langage **C++**

Pablo Rauzy

rauzy@enst.fr

pablo.rauzy.name/teaching.html#epu-cpp

EISE4 @ Polytech'UPMC

17 octobre 2014 — Cours 4

- Nouveautés du C++ par rapport au C.
- Programmation orientée objet, encapsulation.
- Les classes en C++.
- Surcharge des opérateurs.
- Héritage.
- Modèles.
- Exceptions.

→ On a fait le tour du langage.

→ Aujourd'hui, on s'intéresse à la STL.

- La librairie standard de C++ est composée d'une hiérarchie de modèles de classes et de modèles de fonctions.
- Elle implémente différents *conteneurs* de base, ainsi que des *itérateurs* et *algorithmes* conçus pour les manipuler.
- Et bien sûr, la classe `string`.
- La STL ne contient beaucoup de choses, mais pas tout ce que l'on est en droit d'attendre de la librairie standard d'un langage moderne (et pour cause). C'est pourquoi il existe des librairies généralistes additionnelles comme `Boost`.

- Documentation STL : `http://en.cppreference.com/w/cpp`.
- Boost : `http://www.boost.org/`.

- Nous allons voir les conteneurs principaux de la STL, mais n'hésitez pas à utiliser la documentation que vous pouvez trouver en ligne pour les autres si vous en avez besoin.
- La STL contient :
  - un conteneur simple : `pair` ;
  - des conteneurs séquentiels : `vector`, `list`, `deque`, `...` ;
  - des conteneurs associatifs : `set`, `map`, `hash_map`, `...` ;
  - des adaptateurs de conteneurs : `stack`, `queue`, `priority_queue`.

std::pair<T1, T2>

- Description :
  - `#include <utility>`
  - `template<typename T1, typename T2> struct pair;`
  - Sert à stocker deux membres, `first` et `second`, de type choisi.
- Les pairs sont comparables (opérateurs ==, !=, <, <=, >, >=) et assignables (opérateur =).
- La fonction `make_pair` créée une paire avec ses deux arguments.
- Ce conteneur est utilisé par la STL dans ses conteneurs associatifs.

Exemple : std::pair<T1, T2>

```
std::pair<std::string, int> p1("réponse", 42);

std::cout << "Premier élément : " << p1.first << std::endl;
std::cout << "Second élément : " << p1.second << std::endl;


template<typename T>
void swap (std::pair<T, T> &p)
{
  T tmp = p.first;
  p.first = p.second;
  p.second = tmp;
}

std::pair<char, char> p2 = std::make_pair('a', 'b');

swap(p2);

std::cout << "Premier élément : " << p2.first << std::endl;
std::cout << "Second élément : " << p2.second << std::endl;
```

std::vector<Elem>

- Description :
  - `#include <vector>`
  - `template<typename Elem> class vector;`
  - Tableau de taille dynamique.
  - Les éléments sont stockés de façon contiguë.
- Principales méthodes :
  - `at`, `operator []`
  - `front`, `back`
  - `size`, `capacity`, `resize`, `reserve`, `empty`
  - `push_back`, `pop_back`
  - `insert`, `erase`, `clear`
- Complexités algorithmiques :
  - Accès direct : $O(1)$.
  - Ajout ou retrait d'élément à la fin : $O(1)$.
  - Ajout ou retrait d'élément : $O(n)$.

Exemple : std::vector<Elem>

```
template<typename T>
void print_seq(const T& c)
{
    for (auto x : c) {
        std::cout << ' ' << x;
    }
    std::cout << '\n';
}

std::vector<int> v;
v.reserve(5);

v.push_back(1);
v.push_back(2);
v.insert(v.begin(), 3);

print_seq(v); // 3 1 2

v.front(); // 3
v.at(1); // 1
v.back(); // 2

v.size(); // 3
v.capacity(); // 5

v.clear();
v.empty(); // true
```

std::list<Elem>

- Description :
  - `#include <list>`
  - `template<typename Elem> class list;`
  - Liste doublement chaînées.
- Principales méthodes :
  - `front`, `back`
  - `size`, `max_size`, `empty`
  - `push_back`, `pop_back`, `push_front`, `pop_front`
  - `insert`, `erase`, `clear`
  - `remove_if`, `unique`, `sort`, `merge`
- Complexités algorithmiques :
  - Accès direct : non / $O(n)$.
  - Ajout ou retrait d'élément aux extrémité : $O(1)$.
  - Ajout ou retrait d'élément : $O(1)$.

Exemple : std::list<Elem>

```
std::list<int> l;

l.push_back(1);
l.push_back(2);
l.push_front(3);

print_seq(l); // 3 1 2

l.front(); // 3
l.back(); // 2

l.size(); // 3

l.clear();
l.empty(); // true
```

std::deque<Elem>

- Description :
  - `#include <deque>`
  - `template<typename Elem> class deque;`
  - File bidirectionnelle.
- Principales méthodes :
  - `at`, `operator []`
  - `front`, `back`
  - `size`, `max_size`, `empty`
  - `push_back`, `pop_back`, `push_front`, `pop_front`
  - `insert`, `erase`, `clear`
- Complexités algorithmiques :
  - Accès direct : $O(1)$.
  - Ajout ou retrait d'élément aux extrémités : $O(1)$.
  - Ajout ou retrait d'élément : $O(n)$.

Exemple : std::deque<Elem>

```
std::deque<int> d;

d.push_back(1);
d.push_back(2);
d.push_front(3);

print_seq(d); // 3 1 2

d.front(); // 3
d.at(1); // 1
d.back(); // 2

d.size(); // 3

d.clear();
d.empty(); // true
```

std::set<Key, Cmp = std::less<Key>>

- Description :
  - `#include <set>`
  - `template<typename Key, typename Cmp> class set;`
  - Ensemble.
- Principales méthodes :
  - `size`, `max_size`, `empty`
  - `insert`, `erase`, `clear`
  - `count`, `find`, `lower_bound`, `upper_bound`
- Complexités algorithmiques :
  - Insertion : $O(\log n)$.
  - Suppression : $O(\log n)$.
  - Recherche : $O(\log n)$.

Exemple : std::set<Key, Cmp = std::less<Key>>

```
std::set<int> s;

s.insert(3);
s.insert(2);
s.insert(3);
s.insert(1);

print_seq(s); // 1 2 3

s.erase(2);

print_seq(s); // 1 3

s.size(); // 2
(s.find(2) != s.end()); // false

s.clear();
s.empty(); // true
```

std::less<T>

- Description :
  - `#include <functional>`
  - `template <typename T> struct less;`
  - Foncteurs de comparaison.
- `bool operator()(const T& lhs, const T& rhs) const;`
- Appelle l'opérateur de comparaison < sauf si spécialisé.

Exemple : std::less<T>

```
class cstrless {
public:
  bool operator () (const char *a, const char *b) const
  {
    return strcmp(a, b) < 0;
  }
};

std::set<const char *, cstrless> s;

s.insert("foo");
s.insert("bar");
s.insert("foo");
s.insert("baz");

print_seq(s); // bar baz foo

s.erase("baz");

print_seq(s); // bar foo

s.size(); // 2
(s.find("foo") != s.end()); // true

s.clear();
s.empty(); // true
```

std::map<Key, Elem, Cmp = std::less<Key>>

- Description :
  - `#include <map>`
  - `template<typename Key, typename Elem, typename Cmp> class map;`
  - Dictionnaire.
- Principales méthodes :
  - `size`, `max_size`, `empty`
  - `insert`, `erase`, `clear`
  - `count`, `find`, `lower_bound`, `upper_bound`
- Complexités algorithmiques :
  - Insertion : $O(\log n)$.
  - Suppression : $O(\log n)$.
  - Recherche : $O(\log n)$.

Exemple : std::map<Key, Elem, Cmp = std::less<Key>>

```
template<typename T>
void print_assoc(const T& c)
{
    for (auto x : c) {
      std::cout  << x.first << ' ' << x.second << " - ";
    }
    std::cout << '\n';
}

std::map<int, std::string> dict;

dict.insert(std::make_pair(65, "toto"));
dict.insert(std::make_pair(3, "lala"));
dict.insert(std::make_pair(42, "foo"));

print_assoc(dict); // 3 lala - 42 foo - 65 toto -

dict.count(3) // 1

dict.erase(42);

print_assoc(dict); // 3 - lala - 65 toto -
```

std::stack<Elem, Cont = std::deque<Elem>>

- Description :
  - `#include <stack>`
  - `template<typename Elem, typename Cont> class stack;`
  - Pile.
- Principales méthodes :
  - `push`, `pop`, `top`
  - `empty`, `size`
- Complexités algorithmiques :
  - Empilage : $O(1)$.
  - Dépilage : $O(1)$.
  - Lecture : $O(1)$.

Exemple : std::stack<Elem, Cont = std::deque<Elem>>

```
std::stack<int> s;

s.push(1);
s.push(5);
s.push(3);
s.push(2);
s.push(3);

s.size(); // 5

s.top(); // 3

s.pop(); // 3

s.top(); // 2

s.size(); // 4
```

`std::queue<Elem, Cont = std::deque<Elem>>`

- Description :
  - `#include <queue>`
  - `template<typename Elem, typename Cont> class queue;`
  - File.
- Principales méthodes :
  - `push`, `pop`
  - `front`, `back`
  - `empty`, `size`
- Complexités algorithmiques :
  - Enfilage : $O(1)$.
  - Défiler : $O(1)$.
  - Lecture de la tête : $O(1)$.

Exemple : std::queue<Elem, Cont = std::deque<Elem>>

```
std::queue<int> q;

q.push(1);
q.push(5);
q.push(3);
q.push(2);
q.push(3);

q.size(); // 5

q.front(); // 1

q.pop(); // 1

q.front(); // 5

q.size(); // 4
```

std::priority_queue<E, Cont = std::deque<E>, Cmp = std::less<E>>

- Description :
  - `#include <queue>`
  - `template<typename Elem, typename Cont, typename Cmp> class priority_queue;`
  - File à priorités.
- Principales méthodes :
  - `push`, `pop`
  - `front`, `back`
  - `empty`, `size`
- Complexités algorithmiques :
  - Enfilage : $O(logn)$.
  - Défiler : $O(logn)$.
  - Lecture de la tête : $O(1)$.

Exemple : std::priority_queue<E, Cont = std::deque<E>, Cmp = std::less

```
std::priority_queue<int> pq;

pq.push(1);
pq.push(5);
pq.push(3);
pq.push(2);
pq.push(3);

pq.size(); // 5

pq.front(); // 5

pq.pop(); // 5

pq.front(); // 3

pq.size(); // 4
```

- La STL fourni 5 types d'itérateurs, qui sont défini par les opérations qu'on peut effectuer dessus.
  - `InputIterator`, `OutputIterator`
  - `ForwardIterator`
  - `BidirectionalIterator`
  - `RandomAccessIterator`
- Un pointeur est un itérateur valide.
- Dans les slides suivant, la sémantique des opérations sur les itérateurs est la même que celle des opérations équivalente sur les pointeurs.

- Les conteneurs séquentiels de la STL sont capable de retourner un itérateur avec les méthodes `begin` et `end`.

InputIterator

- Soit `i`, `j` deux instances du type `It` qui est un `InputIterator`.
  - `i == j`, `i != j`
  - `*i`
  - `i->m`
  - `++i`

Exemple : InputIterator

```
template <typename InputIt, typename T>
InputIt find_assoc (InputIt first, InputIt last, const T& value)
{
  while (first != last && *first != value)
    ++first;
  return first;
}

std::vector<int> v;
v.push_back(1);
v.push_back(3);
v.push_back(42);
v.push_back(13);

find(s.begin(), s.end(), 42);
```

- Soit `r` une instance du type `It` qui est un `OutputIterator`, et `o` une valeur compatible.
  - `*r = o`
  - `++r`
  - `*r++`

Exemple : OutputIterator

```
template <typename InputIt, typename OutputIt>
OutputIt copy (InputIt first, InputIt last, OutputIt result)
{
    while (first != last)
      *result++ = *first++;
    return result;
}

int data[100];
std::vector<int> newdata(100);

copy(data, data+100, newdata.begin());
```

- Soit `i` une instance du type `It` qui est un `ForwardIterator`.
  - `InputIterator`
  - `++i`
  - `i++`
  - `*i++`

Exemple : ForwardIterator

```
template <typename FwdIt, typename T>
void replace (FwdIt first, FwdIt last, const T& old_v, const T& new_v)
{
   while (first != last)
   {
      if (*first == old_v)
          *first = new_v;
      ++first;
   }
}

std::vector<int> v;
v.push_back(1);
v.push_back(3);
v.push_back(42);
v.push_back(1);
v.push_back(13);

replace(v.begin(), v.end(), 1, 2);

print_seq(v); // 2 3 42 2 13
```

BidirectionalIterator

- Soit `i` une instance du type `It` qui est un `BidirectionalIterator`.
  - `ForwardIterator`
  - `--i`
  - `i--`
  - `*i--`

Exemple : BidirectionalIterator

```
template <typename BidirIt, typename OutputIt>
OutputIt reverse_copy (BidirIt first, BidirIt last, OutputIt result)
{
   while (first != last)
      *result++ = *--last;
   return result;
}

std::list<int> l;
l.push_back(1);
l.push_back(3);
l.push_back(42);
l.push_back(1);
l.push_back(13);

std::vector<int> v(l.size());

reverse_copy(l.begin(), l.end(), v.begin());

print_seq(v); // 13 1 42 3 1
```

- Soit `a`, `b`, `i`, et `r` des instances du type `It` qui est un `RandomAccessIterator`.
  - `BidirectionalIterator`
  - `r += n`, `r -= n`
  - `i + n`, `n + i`
  - `b - a`
  - `i[n]`
  - `a < b`, `a <= b`, `a > b`, `a >= b`

Exemple : RandomAccessIterator

```
template <typename RndAccIt>
void mixup (RndAccIt first, RndAccIt last)
{
   while (first < last)
   {
      std::iter_swap(first, first + (std::rand() % (last - first)));
      ++first;
   }
}

std::list<int> l;
l.push_back(1);
l.push_back(3);
l.push_back(42);
l.push_back(1);
l.push_back(13);

mixup(l.begin(), std::next(l.begin(), 3));

print_seq(v); // 42 1 3 1 13
```

- La STL fourni beaucoup d'algorithmes standards qui sont implémentés dans des modèles de fonctions non-membres et peuvent donc s'appliquer aux différents conteneurs de la STL pour lesquels ils font sens.

- `#include <algorithm>`

Séquences - non-modifiants

- `for_each`
- `count`, `count_if`
- `find`, `find_if`
- `equal`

Exemple : count_if

```
bool even (int v)
{
  return 0 == (v % 2);
}

std::list<int> l;
l.push_back(1);
l.push_back(3);
l.push_back(42);
l.push_back(68);
l.push_back(13);

count_if(l.begin(), l.end(), even);
```

Séquences - modifiants

- `copy`, `copy_if`
- `fill`
- `transform`, `generate`
- `remove`, `remove_if`
- `replace`, `replace_if`
- `reverse`, `reverse_copy`
- `rotate`, `rotate_copy`
- `unique`

Exemple : remove_if

```
bool even (int v)
{
  return 0 == (v % 2);
}

std::list<int> l;
l.push_back(1);
l.push_back(3);
l.push_back(42);
l.push_back(68);
l.push_back(13);

remove_if(l.begin(), l.end(), even);

print_seq(l); // 1 3 13
```

Partitionnement

- `partition`
- `stable_partition`

Exemple : partition

```
bool even (int v)
{
  return 0 == (v % 2);
}

void display (int v)
{
  std::cout << v << ' ';
}

std::list<int> l;
l.push_back(1);
l.push_back(3);
l.push_back(42);
l.push_back(68);
l.push_back(13);

for_each(l.begin(), l.end(), display); // 1 3 42 68 13

std::list<int>::iterator it = partition(l.begin(), l.end(), even);

for_each(l.begin(), l.end(), display); // 68 42 3 1 13

for_each(l.begin(), it, display); // 68 42

for_each(it, l.end(), display); // 3 1 13
```

- `sort`
- `stable_sort`
- `partial_sort`, `partial_sort_copy`

- `lower_bound`, `upper_bound`
- `binary_search`

Exemple : sort

```
std::list<int> l;
l.push_back(1);
l.push_back(42);
l.push_back(3);
l.push_back(68);
l.push_back(13);

for_each(l.begin(), l.end(), display); // 1 3 42 68 13

sort(l.begin(), l.end());

for_each(l.begin(), l.end(), display); // 1 2 13 42 68
```

Ensembles

- `merge`
- `includes`
- `set_difference`, `set_intersection`, `set_union`

Exemple : includes

```
std::set<int> s1, s2;
s1.insert(1);
s1.insert(42);
s1.insert(3);
s1.insert(68);
s1.insert(13);

s2.insert(1);
s2.insert(6);
s2.insert(13);

includes(s1.begin(), s1.end(), s2.begin(), s2.end()); // false

s2.erase(6);

includes(s1.begin(), s1.end(), s2.begin(), s2.end()); // true
```

Tas

- `make_heap`
- `push_heap`
- `pop_heap`
- `sort_heap`

Exemple : Tas

```
std::vector<int> v{ 3, 1, 4, 1, 5, 9 };

print_seq(v); // 3 1 4 1 5 9

make_heap(v.begin(), v.end());

print_seq(v); // 9 5 4 1 1 3

pop_heap(v.begin(), v.end());

int a = v.back(); // 9
v.pop_back();

print_seq(v); // 5 3 4 1 1

// Si besoin d'un tas, utiliser une priority_queue est bien mieux !
```

Min et max

- `min`, `max`
- `min_elements`, `max_elements`

Exemple : min_elements

```
std::vector<int> v{ 3, 1, 4, 1, 5, 9 };

min_element(v.begin(), v.end()); // 1
```

Documentation

- http://en.cppreference.com/w/cpp

C'est tout pour aujourd'hui :)

- Cours précédents
- Standard Template Library
- Conteneurs
  - `std::pair<T1, T2>`
  - `std::vector<Elem>`
  - `std::list<Elem>`
  - `std::deque<Elem>`
  - `std::set<Key, Cmp = std::less<Key>>`
  - `std::map<Key, Elem, Cmp = std::less<Key>>`
  - `std::stack<Elem, Cont = std::deque<Elem>>`
  - `std::queue<Elem, Cont = std::deque<Elem>>`
  - `std::priority_queue<E, Cont = std::deque<E>, Cmp = std::less<E>>`
- Itérateurs
  - `InputIterator`
  - `OutputIterator`
  - `ForwardIterator`
  - `BidirectionalIterator`
  - `RandomAccessIterator`
- Algorithmes
  - Séquences - non-modifiants
  - Séquences - modifiants
  - Partitionnement
  - Tris
  - Ensembles
  - Tas
  - Min et max
- Documentation