Cours C++

Jean-Baptiste Mouret
mouret@isir.fr

ISIR - Université Paris 6
http://www.isir.fr

Cours 8

Foncteurs

En C, on utilise souvent des pointeurs sur fonction :

- passer une fonction en argument d'une autre (e.g. qsort)
- pour faire des **callbacks**

```
#include <stdlib.h>
typedef float (*fun_t)(float);
float add1(float x) {
  return x + 1;
}
float sub2(float x) {
  return x - 2;
}
void apply(float* tab, int n, fun_t f) {
  for (int i = 0; i < n; ++i)
    tab[i] = (*f)(tab[i]);
}
int main() {
  float* x = (float*)calloc(10, 0);
  apply(x, 10, add1);
  apply(x, 10, sub2);
  return 0;
}
```

En objet (e.g. Java), on préfère passer des objets

```
interface Func {
    public float method(float);
}
class Add1 : implements Func {
    public float method(float x) { return x + 1; }
}
class Sub2 : implements Func {
    public float method(float x) { return x + 1; }
}

public class Function
{
    static void apply(Func f, ArrayList<float> v) {
        for (int i = 0; i < v.size(); ++i)
            f.method(v);
    }
    static void main(String[] args)
    {
        ArrayList<float> a = new ArrayList<float>();
        apply(a, new Add1());
        apply(a, new Sub2());
    }
}
```

En C++, on fait des **functors** (foncteurs) : des objets avec un opérateurs ()

```
struct Add1 { // struct = class mais public par defaut
  float operator()(float x) {
    return x + 1;
  }
};
struct Sub2 { // struct = class mais public par defaut
  float operator()(float x) {
    return x - 2;
  }
};
// utilisation : ressemble a une fonction
int main() {
  Sub2 sub;
  Add1 add;
  float v = sub(2); // v = 0
  float y = add(2); // y = 3
}
```

```
#include <iostream>
struct Functor {
  virtual float operator()(float x) const = 0;
};
struct Add1 : public Functor {
  float operator()(float x) const {
    return x + 1;
  }
};
struct Sub2 : public Functor {
  float operator()(float x) const {
    return x - 2;
  }
};
void apply(float* tab, int n, const Functor& f) {
  for (int i = 0; i < n; ++i)
    tab[i] = f(tab[i]);
}
int main() {
  float x[] = {1, 2, 3, 4};
  apply(x, 4, Add1());
  apply(x, 4, Sub2());
  return 0;
}
```

```
#include <iostream>
struct Add1 {
  float operator()(float x) const {
    return x + 1;
  }
};
struct Sub2 {
  float operator()(float x) const {
    return x - 2;
  }
};
template<typename F>
void apply(float* tab, int n, const F& f) {
  for (int i = 0; i < n; ++i)
    tab[i] = f(tab[i]);
}
int main() {
  float x[] = {1, 2, 3, 4};
  apply(x, 4, Add1());
  apply(x, 4, Sub2());
  return 0;
}
```

STL – Standard Template Library

Les conteneurs en C++

- Quatre types beaucoup utilisés pour écrire des algorithmes :
  - std :: vector<T> : tableau redimensionnables
  - std :: list <T> : liste chaînée
  - std :: set<T> : ensemble
  - std :: map<T> : table associative
- Pas d'arbres ni de graphes
- Opérations typiques :
  - accès à un élément
  - supprimer un élément
  - ajouter un élément
  - rechercher un élément dans la collection
- Chaque type diffère par les temps d'exécution de chaque opération (aucun conteneur n'est parfait)

std :: vector<T> : principales méthodes

- v.push_back(o) : ajout d'un élément o de type T **à la fin**
- v.pop_back() : suppression de l'élément de **la fin**
- **void** clear () : supprime tous les éléments
- **operator**[](size_i i) : renvoie l'élément à la case i
- Doc : `http://www.sgi.com/tech/stl/Vector.html`

```
#include <vector>
int main() {
  std::vector<int> v(10); // taille 10
  v.push_back(42);
  v[0] = 24;
  for (size_t i = 0; i < v.size(); ++i)
    std::cout << v[i] << std::endl;
  return 0;
}
```

std :: vector<T> : caractéristiques

- **Ajout à la fin :** rapide
- **Suppression à la fin :** rapide
- **Suppression au milieu :** lent (oblige à tout décaler)
- **Accès/changement à/d' un élément à un index donné (accès aléatoire)** : rapide

➔ À ne pas utiliser quand on doit souvent :

- enlever un élément au début **et** à la fin
- supprimer beaucoup d'éléments « au milieu »

➔ À utiliser dans : tous les autres cas

Liste chaînée

- On doit parcourir dans l'ordre

➔ concept des itérateurs

```
#include <list>
#include <iostream>
int main() {
  std::list<int> my_list;
  my_list.push_back(42);
  my_list.push_back(4242);

  for (std::list<int>::const_iterator it = my_list.begin();
       it != my_list.end(); ++it)
    std::cout << *it << std::endl;
  std::list<int>::iterator it = my_list.begin();
  *it = 2; // changement du premier element
  return 0;
}
```

Itérateurs

- std :: list <T>::iterator : type de l'iterateur
- std :: list <T>::const_iterator : type de l'iterateur constant
- ++it : itérateur suivant
- --it : itérateur précédent
- it ->methode() : méthode de l'objet pointé par it
- my_list.begin() : itérateur sur le premier elémément
- my_list.end() : itérateur sur le dernier elémément (pointeur "nul")
- my_list. insert (iterator pos, **const** T& t) : insert t avant pos
- Doc (listes) : `http://www.sgi.com/tech/stl/List.html`

Le parcours avec des itérateurs fonctionne aussi avec des vecteurs.

```
int main() {
  std::list<int> my_list;
  my_list.push_back(42);
  my_list.push_back(4242);

  for (std::list<int>::const_iterator it = my_list.begin();
       it != my_list.end(); ++it)
    std::cout << *it << std::endl;
  std::list<int>::iterator it = my_list.begin();
  *it = 2; // changement du premier element
  return 0;
}
```

```
int main() {
  std::vector<int> my_list;
  my_list.push_back(42);
  for (std::vector<int>::const_iterator it = my_list.begin();
       it != my_list.end(); ++it)
    std::cout << *it << std::endl;

  return 0;
}
```

... on peut donc faire des algorithmes génériques

```cpp
#include <list>
#include <vector>
#include <iostream>

template<typename T>
void print(const T& my_list) {
  for (typename T::const_iterator it = my_list.begin();
       it != my_list.end(); ++it)
    std::cout << *it << std::endl;
}
int main() {
  std::list<int> my_list;
  print(my_list);

  std::vector<int> my_vec;
  print(my_list);
  return 0;
}
```

... et encore plus générique

```
#include <list>
#include <vector>
#include <iostream>
template<typename T>
void print(const T& i1, const T& i2) {
  for (std::list<int>::const_iterator it = i1;
       it != i2; ++it)
    std::cout << *it << std::endl;
}
int main() {
  std::list<int> my_list;
  print(my_list.begin(), my_list.end());

  std::vector<int> my_vec;
  print(my_list.begin(), my_list.end());
  return 0;
}
```

Conclusion : listes

	std : :list	std : :vector
Ajout à la fin	O(1)	O(1)
Ajout au début	O(1)	O(n)
Accès aléatoire	O(n)	O(1)
Suppression à la fin	O(1)	O(1)
Suppression au début	O(1)	O(n)
Suppression après un élément	O(1)	O(n)
Recherche	O(n)	O(n)
Classement	O(n log(n))	O(n log(n)))

- O(1) : rapide
- O(n) : lent (linéairement dépendant de la taille de la liste)

Parcours avec des itérateurs.

Remarques

- Pas de conteneurs avec des références
- std :: vector<A> a(10) : appelle 10 fois le constructeur par défaut de A
- std :: vector<A*> a(10) : il faut allouer chaque A avec new ... et delete
- On n'a vu qu'une partie des méthodes de chaque classe...

std :: set<T> : introduction

- On souhaite représenter un ensemble **non ordonné**
- Un ensemble ne contient pas d'éléments dupliqués
- Opérations qui doivent être rapides :
  - ajout d'un élément
  - suppression d'un élément
  - **savoir si un élément appartient à l'ensemble**

std :: set<T>

- Représente l'ensemble par un arbre de recherche

➔ **Nécessite qu'un opérateur < soit défini pour T** (ou une fonction de comparaison doit être fournie)

- (n = taille de l'ensemble)
- **void** insert (**const** T& t) : ajout d'un élément ➔ temps = O(log(n))
- iterator find (**const** T& t) : recherche d'un élément ➔ temps = O(log(n))
- doc : `http://www.sgi.com/tech/stl/set.html`

```cpp
#include <list>
#include <set>
#include <iostream>

int main() {
  std::set<int> s;
  s.insert(42); s.insert(42); s.insert(24);
  // parcours
  for (std::set<int>::const_iterator it = s.begin();
       it != s.end(); ++it)
    std::cout << *it << std::endl;

  // recherche
  std::set<int>::iterator it = s.find(42);
  if (it != s.end())
    std::cout << "trouve la reponse" << std::endl;
  else
    std::cout << "je cherche encore" << std::endl;
  return 0;
}
//24
//42
//trouve la reponse
```

```cpp
// avec une classe quelconque
#include <list>
#include <set>
#include <iostream>
class A {
public:
  A(int x) : _x(x) {}
  bool operator<(const A& o) const { return _x < o._x ; }
  int x() const { return _x; }
protected:
  int _x;
};

int main() {
  std::set<A> s;
  s.insert(A(42)); s.insert(A(42)); s.insert(A(24));
  // parcours
  for (std::set<A>::const_iterator it = s.begin();
       it != s.end(); ++it)
    std::cout << it->x() << std::endl;
}
```

```cpp
// avec une fonction de comparaison
#include <list>
#include <set>
#include <iostream>
class A {
public:
  A(int x) : _x(x) {}
  int x() const { return _x; }
protected:
  int _x;
};
bool compare(const A& a1, const A& a2) {
  return a1.x() < a2.x();
}

int main() {
  std::set<A, compare> s;
  s.insert(A(42)); s.insert(A(42)); s.insert(A(24));
  // parcours
  for (std::set<A>::const_iterator it = s.begin();
       it != s.end(); ++it)
    std::cout << it->x() << std::endl;
}
```

```
// avec un foncteur de comparaison
#include <list>
#include <set>
#include <iostream>
class A {
public:
  A(int x) : _x(x) {}
  int x() const { return _x; }
protected:
  int _x;
};
struct Comparator {
  bool operator()(const A& a1, const A& a2) {
    return a1.x() < a2.x();
  }
};

int main() {
  std::set<A, Comparator> s;
  s.insert(A(42)); s.insert(A(42)); s.insert(A(24));
  // parcours
  for (std::set<A>::const_iterator it = s.begin();
       it != s.end(); ++it)
    std::cout << it->x() << std::endl;
}
```

Table associative (Map)

- Une table associative associe une **valeur** (un objet) à une **clé** (un autre objet)
- Exemples :
  - élément : étudiant (nom, prénom, etc.) / clé : numéro d'étudiant
  - élément : abonné téléphonique / clé : numéro de téléphone
- Les types de la valeur et de la clé peuvent être différents
- Exemple :
  - ma_table une table associant une chaine de caractères à un entier
  - ajouter un couple clé/valeur : `ma_table["42"] = 22;`
  - accès par clé : `ma_table["42"]` ➔ renvoie 22
- Représente l'ensemble par un arbre de recherche

➔ **Nécessite qu'un opérateur $<$ soit défini pour T** (ou fonction de comparaison)

Map : opérations rapides (log(n))

- Ajout de couple clé/valeur : ma_table[key] = val
- Accès par clé : ma_table[key]
- Savoir si une clé est dans la table : find (cf std :: set)
- Récupérer et itérer sur l'ensemble des clés : itérateurs
  - it->first : clé
  - it->second : élément
- doc : `http://www.sgi.com/tech/stl/Map.html`

```cpp
#include <map>
#include <string>
#include <iostream>
int main() {
  typedef std::map<std::string, int> month_t;
  typedef month_t::iterator month_iterator_t;
  month_t months;
  months["january"] = 31;
  months["february"] = 28;
  months["march"] = 31;
  std::cout << months["january"] << std::endl;
  month_iterator_t it = months.find("february");
  if (it != months.end())
    std::cout << it->first << "=>"
              << it->second << std::endl;
}
// 31
// february=>28
```

```cpp
#include <map>
#include <string>
#include <iostream>
class A {
public:
  A(int x) : _x(x) {}
  int x() const { return _x; }
protected:
  int _x;
};
int main() {
  typedef std::map<std::string, A*> month_t;
  typedef month_t::iterator month_iterator_t;
  month_t months;
  months["january"] = new A(31);
  months["february"] = new A(28);
  std::cout << months["january"]->x() << std::endl;
  month_iterator_t it = months.find("february");
  if (it != months.end())
    std::cout << it->first << "=>"
              << it->second->x() << std::endl;
  for (month_iterator_t it = months.begin(); it != months.end();
       ++it)
    delete it->second;
}
```

```
// classes utilisateurs pour les cles : definir un functor
#include <map>
#include <string>
#include <iostream>
class A {
public:
  A() {}
  A(const std::string& x) : _x(x) {}
  const std::string& x() const { return _x; }
protected:
  std::string _x;
};
struct compare_a {
  bool operator()(const A& a1, const A& a2) {
    return a1.x() < a2.x();
  }
};
int main() {
  typedef std::map<A, int, compare_a> month_t;
  typedef month_t::iterator month_iterator_t;
  month_t months;
  months[A("january")] = 31;
  months[A("february")] = 28;
  std::cout << months[A("january")] << std::endl;
  month_iterator_t it = months.find(A("february"));
}
```

Algorithmes génériques

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
int main()
{
  std::vector<int> v(5);
  std::fill(v.begin(), v.end(), 42);// rempli
  std::sort(v.begin(), v.end());//tri
  v.push_back(22);
  std::cout << std::count(v.begin(), v.end(), 42) << std::endl;
  std::cout << *std::max_element(v.begin(), v.end()) << std::endl;
  //supprime les 42
  v.erase(std::remove(v.begin(), v.end(), 42), v.end());
  // affiche v
  std::copy(v.begin(), v.end(),
            std::ostream_iterator<int>(std::cout, "\n"));
  return 0;
}
```

http://www.cplusplus.com/reference/algorithm/

Typedef

```
using namespace std;
// une map, cle = entier, donnees = paire de vecteurs
map<int, pair<vector<int>, vector<float> > my_map;
// iterer dessus...
for (map<int, pair<vector<int>, vector<float> >::const_iterator
     it = my_map.begin(); it != my_map.end(); ++it)
      std::cout << it->first << std::endl;
```

Solution : typedef

```
using namespace std;
// une map, cle = entier, donnees = paire de vecteurs
typedef pair<vector<int>, vector<float> > data_t;
typedef map<int, data_t> map_t;
map_t  my_map;
// iterer dessus...
for (map_t::const_iterator it = my_map.begin();
      it != my_map.end(); ++it)
      std::cout << it->first << std::endl;
```

```
int main(int argc, char **argv)
{
  typedef gen::EvoFloat<30, Params> gen_t;
  typedef phen::Parameters<gen_t, FitZDT2<Params>, Params> phen_t;
  typedef eval::Eval<Params> eval_t;
  typedef boost::fusion::vector<stat::ParetoFront<phen_t, Params>
stat_t;
  typedef modif::Dummy<> modifier_t;
  typedef ea::Nsga2<phen_t, eval_t,
    stat_t, modifier_t, Params> ea_t;

  ea_t ea;

  run_ea(argc, argv, ea);

  return 0;
}
```

Typename

Deux usages :

```
template <typename T>
//....
```

```
template <typename T>
void test(const T& t) {
  for (typename T::const_iterator it = t.begin();
       it != t.end(); ++it)
    std::cout << *it << std::endl;
}
```

Règle : devant un nom de type dépendant, on doit mettre typename.

Typename : raison

```
template<typename T>
void test(const T& t) {
  T::iterator it = t.begin(); // ne compile pas
}
```

Ambiguité avec les attributs de classes (static)

```
class X {
  typedef ZZ iterator;
};
class Z {
  static int iterator;
};
// =>
X::iterator x;
Z::iterator = 42;
```

Boost

Boost

- La STL est très puissante
- ... mais elle ne fait pas tout
- Boost :
  - ensemble de bibliothèques qui complètent la STL
  - souvent l'antichambre du standard C++
  - 80 bibliothèques, de taille et de complexité très variables
- `http://www.boost.org`

Exemples de bibliothèques utiles

- Asio : réseau portable (sockets, etc.)
- Assign : remplir des conteneurs facilement
- Filesystem : accès portable au système de fichier (chdir, etc.)
- GIL : traitement d'image
- Graph : graphes et algorithmes sur les graphes
- Lexical cast : convertir un entier en chaine de char. (et inversement), etc.
- Program Options
- Random
- Regex : expressions régulière
- Serialization : lire/écrire des objets dans des streams
- Smart pointers : plus besoin de faire de delete
- Test : tests unitaires
- uBLAS : algèbre linéaire

Assign

```
#include <boost/assign/std/vector.hpp>
#include <boost/assign/list_of.hpp>
//...
{
  using namespace boost::assign;
  std::vector<int> values;
  values += 1,2,3,4,5,6,7,8,9;
  // surcharge de l'operateur ,!

  std::list<int> primes = list_of(2)(3)(5)(7)(11);

  std::map<int, int> next =
      map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);

}
```

boost::shared_ptr

```
#include <boost/shared_ptr.hpp>
struct C {
  void test() const { std::cout<<"42"<<std::endl; }
};

void f(boost::shared_ptr<C> c)
{
  // comme un pointeur
  c->test();
  // copie du pointeur
  boost::shared_ptr<C> c2 = c;
  c2->test();
}
int main() {
  boost::shared_ptr<C> x(new C);
  f(x);
  return 0;
}//x automatiquement libere !
```

boost :: lexical_cast <>()

```
#include <boost/lexical_cast.hpp>

int x = 42;
std::string s = "4242";

std::string s2 = boost::lexical_cast<int>(s);

int x2 = boost::lexical_cast<std::string>(x);
```

Exercices

- Programmez une fonction générique qui prend un conteneur d'objets de type T et renvoie l'élement le plus courant (le plus d'occurences) :

```
int c[] = {1, 2, 3, 4, 2, 4};
std::vector<int> v(c, c+6);
int m = most_frequent(v);
```

Programmez une fonction générique qui prend une liste de points, un point et une fonction de distance et qui renvoie le plus proche voisin du point passé en argument (utiliser std :: sort ()).

```
#include <vector>
struct Point {
  float x, y;
  Point(float xx, float yy) : x(xx), y(yy) {}
};
template<typename T>
float dist2(const T& p1, const T& p2) {
  return (p1.x - p2.x) * (p1.x - p2.x)
    + (p1.y - p2.y) * (p1.y - p2.y);
}
int main()
{
  std::vector<Point> l;
  l.push_back(Point(4, 2));
  l.push_back(Point(2, 3));
  l.push_back(Point(3, 1));
  stp::pair<Point, Point> p = nearest_neighbor(l, Point(0, 4),
                                               dist2<Point>);
  return 0;
}
```