Université de Nice-Sophia Antipolis
Polytech ELEC4

Lundi 17 mars 2020
Durée: 2h

Fonctions anonymes
Travaux Dirigés – Séance n. 11

---

1 Introduction

Les *fonctions anonymes*, ou encore *lambda expressions*, est la notion fondamentale des langages de programmation *fonctionnels*. Dans ces langages, comme Lisp, une fonction est une valeur *fonctionnelle*, de la même façon que 10.6 est une valeur réelle, ou `"bonjour"` est valeur de type chaîne de caractères.

Par exemple, une fonction qui met en relation deux valeurs de type quelconque et leur somme peut être notée sous la forme : $(x, y) \to x + y$. Dans la mesure où cette fonction ne possède aucun nom, elle est dite *anonyme*.

Le langage C++, depuis la version C++11, introduit la notion de fonction anonyme. Dans ce TD, nous verrons comme définir une fonction anonyme et l'utiliser (dans un cadre limité). Par la suite, vous utiliserez l'option `--std=c++14` du compilateur `g++`.

2 Déclaration

La valeur fonctionnelle précédente, celle qui additionne, par exemple deux réels, sera dénotée en C++ :

```
[] (double x, double y) { return x+y; }
```

On remarque que les types des paramètres `x` et `y` doivent être mentionnés, mais on peut omettre le type du résultat de la fonction anonyme dans la mesure où le compilateur peut le déterminer par *inférence*. Toutefois, il est toujours possible d'indiquer le type du résultat de la fonction anonyme avec la syntaxe :

```
[] (int x, double y) -> double { return x+y; }
```

En C++, le corps d'une lambda expression ne se limite pas à une simple expression. Puisque C++ est un langage impératif, un bloc d'instructions, avec des déclarations de variables, peut être défini, comme dans les deux exemples suivants :

```
[] (double x) { assert(x>=0);  return sqrt(x); }

[] () {
    std::cout << "n = ";
    int n;
    std::cin >> n;
    std::cout << n*n << std::endl;
  };
```

C++ est un langage typé. Si on souhaite affecter une valeur fonctionnelle à une variable, il faudra indiquer son type. Quel est donc le type d'une valeur fonctionnelle ?

Son type peut être vu comme celui d'une classe dans laquelle l'opérateur `()` a été (re)défini. Par exemple, le type du lambda qui additionne deux réels peut s'écrire :


```
class fonc_sum {
public:
  double operator()(double x, double y) { return x+y; } ;
};
```

Mais si on veut faire la même chose pour la multiplication, il faudra définir une nouvelle classe spécialisée pour cette nouvelle opération. Cela risque de devenir rapidement fastidieux. Heureusement, la bibliothèque standard C++, propose le type générique `std::function` (inclure `<functional>`) pour représenter les valeurs fonctionnelles. Par exemple, si on souhaite affecter les trois valeurs fonctionnelles précédentes à trois variables `f1`, `f2` et `f3`, on pourra écrire :

```
std::function<double(double, double)> f1 =
                    [] (double x, double y) -> double { return x+y; };

std::function<double(double)>  f2 = [] (double x) {
  assert(x>=0);
  return sqrt(x);
};

std::function<void()> f3 = [] () {
  std::cout << "n = ";
  int n;
  std::cin >> n;
  std::cout << n*n << std::endl;
};
```

Mais, il sera aussi possible de laisser au compilateur la tâche de la détermination automatique du type de `f1`, `f2` et `f3` avec l'utilisation du mot-clé **auto**. On écrira alors de façon simplifiée :

```
auto f1 = [] (double x, double y) -> double { return x+y; };

auto f2 = [] (double x) { assert(x>=0); return sqrt(x); };

auto f3 = [] () {
    std::cout << "n = ";
    int n;
    std::cin >> n;
    std::cout << n*n << std::endl;
  };
```

Notez que l'exécution des fonctions anonymes précédentes désignées par les variables `f1`, `f2` et `f3` se fait simplement par l'utilisation des noms de ces variables :

```
std::cout << f1(3.4, -9.76) << std::endl;
std::cout << f2(9.0) << std::endl;
f3();
```

exercice 1) À l'aide d'une fonction `main`, testez les déclarations et les appels précédents.

Pour vérifier si une variable `f` désigne une valeur fonctionnelle, il suffit de tester `(f)`. Par exemple,

```
std::function<int(char)> f;
...
if (f) {
  // ok, f désigne une valeur fonctionnelle => on peut appliquer f
  std::cout  << f('z') << std::endl;
}
```

3 Utilisation

Quel est l'intérêt des *fonctions anonymes* ? Il est multiple. Puisque les *fonctions anonymes* sont des valeurs, on pourra les utiliser comme paramètres effectifs d'autres fonctions, mais surtout comme valeur résultat d'une autre fonction. En d'autres termes, on pourra écrire des fonctions qui fabriquent des fonctions. C'est ce dernier aspect qui donne toute la puissance à la programmation fonctionnelle. Toutefois, on ne verra ce dernier point que très partiellement car cet aspect dépasse le cadre de cet enseignement.

3.1 Paramètre de type fonctionnel

Dans un premier temps, on va s'intéresser à la possibilité de passer une fonction en paramètre. En C, on pouvait déjà le faire à l'aide des pointeurs sur fonction.

Imaginons qu'on veuille écrire une fonction C qui calcule l'aire d'une fonction continue sur un intervalle $[a, b]$ par la méthode des rectangles. On écrit la fonction `aire` suivante, avec comme 4ème paramètre une fonction paramétrique :

```
/*
 * Antécédent : a⩽b et n⩾1
 * Rôle : calcule l'aire de la fonction continue f
 *        sur l'intervalle [a,b]
 * Algorithme : méthode des rectangles, n est le nb de rectangles calculés
 */
double aire(double a, double b, int n, double (*f)(double)) {
  assert(a<=b);
  double largeurRect=(b-a)/n;
  double x=a+largeurRect/2;
  double aire = 0;

  for (int i=1; i<=n; i++, x+=largeurRect)
    // aire = ∑_1^{i-1}(x_{i+1} − x_i) × f((x_i + x_{i+1})/2)
    aire += largeurRect*f(x);
  // aire = ∑_1^{n}(x_{i+1} − x_i) × f((x_i + x_{i+1})/2)
  return aire;
}
```

Pour calculer l'aire de $cos(x) + x$ sur l'intervalle $[0, \pi/2]$, ou celle de x sur $[2, 3]$ (☺ CPGE), on écrira par exemple :

```
double f1(double x) { return cos(x)+x; }
double f2(double x) { return x; }
...
printf("aire = %f\n", aire(0,M_PI/2, 1000, f1));
printf("aire = %f\n", aire(2,3, 1000, f2));
```

Dans le code précédent, on voit que les deux appels à la fonction `aire` nécessitent les déclarations des deux fonctions `f1` et`f2` à passer en paramètre. En C++, avec la notion de *fonction anonyme*, cela devient inutile puisqu'il suffira de passer *directement* la valeur fonctionnelle en paramètre.

Dans l'en-tête de la méthode `aire`, le 4ème paramètre pourra être défini de 4 façons différentes, soit comme en C par un pointeur sur fonction, soit par un template, soit par un `std::function`, soit simplement par **auto**.

exercice 2) Récrivez en C++ la fonction `aire` avec son 4ème paramètre formel déclaré selon les 4 possibilités précédentes, et testez l'appel de votre fonction `aire` avec des valeurs fonctionnelles effectives appropriées.

La bibliothèque standard propose de nombreuses méthodes qui prennent en paramètre des fonctions anonymes. Nous allons en voir quelques unes.

Commençons par la méthode `find_if` (de la bibliothèque `algorithm`) qui recherche dans un conteneur, entre deux positions (représentées par deux itérateurs), le premier élément qui satisfait une condition définie par une fonction anonyme de prototype **bool `f(const T &x)`**. Si l'élément est trouvé, la méthode renvoie sa position, sinon elle renvoie l'itérateur de fin de conteneur. Par exemple, l'appel :

```
#include <algorithm>
...
std::find_if(v.begin(),v.end(), [] (int x) { return x<0; });
```

recherche, dans le conteneur d'entiers `v`, la position du premier entier négatif.

exercice 3) Écrivez un programme qui déclare un vecteur d'entiers que vous initialiserez aux valeurs de votre choix. À l'aide de la méthode `find_if`, recherchez et affichez la valeur de la 1ère occurrence d'un entier pair. Vous traiterez le cas où aucun élément ne satisfait la condition.

La méthode `transform` permet d'appliquer une fonction anonyme de prototype **`R f(const T &x)`** sur tous les éléments d'un conteneur entre deux positions (représentées par deux itérateurs) et d'affecter les résultats dans le conteneur à partir d'une troisième position. La méthode renvoie la position qui suit le dernier élément transformé.

L'exemple suivant transforme tous les caractères de la chaîne `s` en majuscule.

```
std::string s = "hello";
std::transform(s.begin(), s.end(), s.begin(),
               [](const unsigned char &c) { return std::toupper(c); });
```

exercice 4) Testez le code précédent. Puis, modifiez-le pour ne mettre en majuscule que la première lettre.

exercice 5) À l'aide de la méthode `transform`, transformez votre vecteur d'entiers précédent de telle façon que chaque entier soit multiplié par 2.

La méthode `for_each` est similaire à l'énoncé *foreach* du langage C++. Elle applique une fonction anonyme de prototype **void `f(const T &x)`** sur tous les éléments d'un conteneur entre deux positions (représentées par deux itérateurs). Par exemple, ci-dessous, l'appel à la méthode `for_each` écrit sur la sortie standard tous les éléments du conteneur `v` :

```
#include <algorithm>
...
std::for_each(v.begin(),v.end(), [] (int x) { std::cout << x;});
```

exercice 6) Testez le code précédent sur votre vecteur d'entiers.

exercice 7) À l'aide la méthode `for_each`, récrivez l'exercice qui consiste à transformer le vecteur d'entiers en multipliant chacun des entiers par 2.

exercice 8) La méthode `for_each` renvoie comme résultat une fonction anonyme (celle passée en paramètre). Utilisez la méthode `for_each` pour qu'elle multiplie par 2 chacun des entiers du vecteur et qu'elle renvoie la somme de tous les entiers qu'il contient (après multiplication).

La méthode `sort` permet de trier les éléments d'un conteneur entre deux positions représentées par deux itérateurs. Cette méthode de tri garantit une complexité en $\mathcal{O}(n \log_2 n)$.

La relation d'ordre par défaut est `<` qui trie de façon croissante. Cet opérateur doit être applicable sur les éléments du conteneur.

exercice 9) Triez de façon croissante votre vecteur d'entiers.

La méthode `sort` admet un troisième paramètre, une fonction anonyme dont le prototype est **bool** `cmp(const T1 &a, const T2 &b)`, qui donne la relation d'ordre à appliquer pour comparer les éléments.

exercice 10) Triez de façon décroissante votre vecteur d'entiers.

exercice 11) Créez un vecteur de `Rectangle` que vous trierez de façon décroissante selon la taille de leur surface. La fonction anonyme utilisera la méthode `surface`.

exercice 12) Faites à nouveau ce tri, mais en utilisant l'opérateur `>` que vous définirez dans la classe `Rectangle`.

exercice 13) Définissez également l'opérateur `<` et triez le vecteur de rectangles, de façon croissante, cette fois, sans utiliser de fonction anonyme.

Notez que la classe `std::list` possède une méthode `sort` qui trie les éléments d'une liste selon les mêmes principes.

4 Capture

Dans les exemples précédents, les fonctions anonymes accèdent uniquement à leurs paramètres ou à leurs variables locales, qu'on appelle *occurrences liées*. Si une fonction anonyme doit accéder à une variable déclarée par ailleurs (*i.e.* à l'extérieur du lambda), se pose la question de la valeur de cette variable au moment de l'exécution de la fonction anonyme. Ces variables sont appelées *occurrences libres*.

Par exemple, soit le lambda qui renvoie le produit de son paramètre x (occurrence liée) et de la variable n (occurrence libre) définie par ailleurs :

```
(x) → x * n
```

Si la valeur de n est déterminée au moment de la définition du lambda, on parle de *portée statique* ou *lexicale*. En revanche, si on utilise la valeur que possède *une* variable n au moment de l'exécution de la fonction anonyme f, on parle de *portée dynamique*.

En C++, la portée est *statique*, et le lambda contrôle l'accès aux occurrences libres grâce à la paire de crochets `[]` en tête de déclaration. Entre les crochets, on indique les occurrences libres que l'on veut *capturer*.

- `[]` aucune occurrence libre n'est capturée ;
- `[&]` capture toutes occurrences libres par *référence* ;
- `[=]` capture toutes occurrences libres par *copie* et qui ne peuvent être modifiées ;
- `[&x,y]` capture `x` par référence et `y` par copie ;
- `[this]` capture **this**, et donne accès à toute la classe.

On a vu qu'une fonction anonyme était représentée par le type `std::function`. Chaque lambda est implémenté par une classe particulière qui (re)définit l'opérateur `()`. Lors de son instanciation, les occurrences libres capturées sont passées en paramètre du constructeur et mémorisées dans la classe.

exercice 14) Testez le code suivant et modifiez-le pour qu'il fonctionne :

```
int n=10;
std::function<int(int)> f =  [] (int x) { return n*x; };
std::cout << f(3) << std::endl;
```


5 Résultat de type fonctionnel

Une caractéristique fondamentale de la programmation fonctionnelle est la possibilité pour une fonction de fournir comme résultat une autre fonction, permettant ainsi de créer *dynamiquement* une nouvelle fonction. Cette propriété est au cœur des langages fonctionnels, et permet en particulier de mettre en œuvre la composition ou la curryfication de fonctions, ou encore la programmation par continuité. Dans ce qui suit nous donnerons deux simples exemples : composition de fonctions et un générateur de générateurs.

5.1 Composition

En mathématiques, la composition de deux fonctions $f : \mathcal{T} \to \mathcal{R}$ et $g : \mathcal{R} \to \mathcal{Z}$, est la fonction notée $g \circ f : \mathcal{T} \to \mathcal{Z}$, telle que $(g \circ f)(x) = g(f(x))$.

Écrivons la fonction `compose` qui renvoie la composition de 2 fonctions qui vont de **double** vers **double**. Cette fonction pourra s'écrire :

```
std::function<double(double)>
compose(std::function<double(double)> f, std::function<double(double)> g)
{
  return [&f,&g] (double x) { return g(f(x)); };
}
```

On pourra, par exemple, déclarer la variable `gof`, composition des fonctions $x + 1$ et *cos*, et appliquer `gof` comme suit :

```
auto gof = compose([] (double x) {return x+1;},
                   [] (double x) {return cos(x);});
//
std::cout << gof(3) << std::endl; // = -0.653644
```

exercice 15) Testez le code précédent.

Si on souhaite appliquer la composition sur des fonctions avec des ensembles de départ et d'arrivée autres que **double**, il faut rendre générique la fonction `compose`.

exercice 16) Modifiez la fonction `compose` précédente pour la rendre générique, de telle façon que le code suivant puisse s'exécuter sans erreur :

```
auto gof = compose( [] (int *t) {return *t;}, [] (double x) {return x*x;});
int t[] = { 3, 4, 5 };
std::cout << gof(t) << std::endl;
```

exercice 17) Écrivez une fonction qui permet de composer n fois une même fonction.

5.2 Générateurs

On souhaite écrire un générateur qui produit une suite croissante d'entiers positifs. Chaque appel au générateur produit l'entier suivant. Pour faire ce travail, en C, on peut écrire la fonction `generer` suivante en utilisant la déclaration de variable locale statique :

```
int generer() {
  static int x=0;
  return x++;
}
```

exercice 18) Écrivez la fonction précédente, et affichez les résultats de plusieurs appels consécutifs.

Toutefois, si l'on souhaite générer plusieurs suites, il faudra écrire autant de fonctions `generer` que suites désirées. D'où l'idée de fabriquer un générateur de générateurs de suite de nombres entiers.

On souhaite donc écrire une fonction anonyme qui renvoie une fonction anonyme qui, à chacun de ses appels, fournit le prochain nombre entier de la suite. Si la première fonction anonyme s'appelle `creer_generateur`, on pourra créer 2 générateurs `g1` et `g2` et produire 2 suites en les appelant successivement `g1` ou `g2` :

```
std::function<int()> g1 = creer_generateur();
auto g2 = creer_generateur(); // le compilateur déduit le type par inférence

std::cout << g1() << std::endl; // ⇒ 0
std::cout << g1() << std::endl; // ⇒ 1
std::cout << g2() << std::endl; // ⇒ 0
std::cout << g1() << std::endl; // ⇒ 2
std::cout << g2() << std::endl; // ⇒ 1
```

Écrivons le générateur. C'est une fonction anonyme qui n'a pas de paramètre et qui renvoie une fonction anonyme qui n'a pas non plus de paramètre, mais qui renvoie un entier. Le type de `creer_generateur` peut dont être donné par déclaration suivante :

```
std::function<std::function<int()>()> creer_generateur;
```

Écrivons cette fonction anonyme. Elle possède un compteur, une variable locale, et renvoie une fonction anonyme qui incrémente ce compteur :

```
std::function<std::function<int()>()> creer_generateur = [] () {
  int cpt = 0;
  return [cpt] () mutable { return cpt++; };
};
```

Le mot-clé **mutable** est nécessaire, car la fonction anonyme renvoyée doit prendre une copie de `cpt` et le modifier par incrémentation, ce qui n'est pas autorisé par défaut.

exercice 19) Testez le générateur de générateurs précédent.

exercice 20) Modifiez le générateur de générateurs précédent, afin qu'il produise des générateurs de nombres tirés aléatoirement sur l'intervalle $[0; n[$. La valeur n est passée en paramètre du générateur de générateurs. Par défaut $n = 1000$. D'autre part, chaque générateur produit possède sa propre suite avec son propre germe.

6 Problèmes

exercice 21) Reprenez la classe `Dico` de la feuille de TD précédente et utilisez une `std::list` générique à la place de la `std::map`. Vous mémoriserez chaque mot et son nombre d'occurrences dans une `std::pair`. La liste des mots sera écrite sur la sortie standard de façon décroissante (*i.e.* de Z à A).

---

exercice 22) Reprenez la classe `Matrice` de la feuille de TD précédente. Dans le `main`, déclarez et initialisez une matrice `m`. Appliquez la méthode `std::for_each` pour écrire sur la sortie standard les éléments de `m` compris entre 2 positions données par deux itérateurs.

exercice 23) Appliquez la méthode `std::transform` pour multiplier par 2 tous les éléments de `m` par deux.