ISIMA

Expressions lambda

(PARTIE VII – C++11/14)

Bruno Bachelet

Loïc Yon

Algorithmes génériques

- Algorithme générique ⇒ algorithme à trou
  - `std::sort(v.begin(),v.end(),std::greater<int>());`
  - Dernier paramètre = foncteur ou pointeur de fonction

- Implémentation à l'aide de la généricité
  - ```
    template <typename IT,typename COMP>
    void sort(const IT & debut,const IT & fin,
              const COMP & comparer);
    ```
  - `comparer()` ⇒ appel opérateur «`()`» si foncteur
  - `comparer()` ⇒ appel fonction si pointeur de fonction

- Souvent, créer un foncteur est fastidieux
  - Trouver un nom
  - Ecrire la classe
  - Pour un usage souvent ponctuel
    ```

Nouveau concept

- Expression lambda
  - Permet l'écriture d'une fonction à la volée
  - Pour un usage ponctuel
  - Fonction «anonyme»
  - Fonction «contextualisée» (cf. mécanisme de capture)

- Exemple
  - ```
    sort(v.begin(),v.end(),
         [] (int x,int y) { return x>y; });
    ```
  - Tri par ordre décroissant

- Syntaxe: `[`*capture*`]` `(`*arguments*`)` `->` *retour* `{`*code*`}`
  - Arguments, retour, code = éléments d'une fonction normale
    - Remarque: utilisation de la nouvelle syntaxe de retour de fonction
  - Capture = liste des variables du contexte «capturées» par la lambda
    ```

Lambda sans capture

- `[] (int x,int y) { return x>y; }`
  - Type de retour déduit automatiquement
    - A condition que tous les retours soient du même type
  - Equivalent à: `[] (int x,int y) -> bool { return x>y; }`

- Equivalent au foncteur suivant
  - ```
    struct Comparer {
     bool operator () (int x,int y) const
     { return x>y; }
    };
    ```
  - Remarque: opérateur «`()`» constant

- Ou à la fonction suivante
  - `inline bool comparer(int x,int y) { return x>y; }`

Code source complet: `cpp11_lambda.cpp`
    ```

Implémentation des lambdas

- Implémentation libre des lambdas
  - Souvent sous la forme d'un foncteur
  - Mais pour les lambdas sans capture, une fonction suffit
- Dépend donc du compilateur
  ⇒ impossible de connaître *a priori* le type d'une lambda
- Mais possibilité de stocker une lambda dans une variable
  - `auto f = [] (int x,int y) { return x>y; };`
  - `if (f(v[i],v[j])) ...`
- Et aussi de «capter» le type d'une lambda
  - `typedef decltype(f) lambda_t;`
  - `typeid(lambda_t).name()` ⇒ *main::{lambda(int, int)#2}* (g++ 4.8.3)

Capture de variables *(1/5)*

- Une lambda peut utiliser des variables de son contexte
  ⇒ mécanisme de «capture»

- Exemple: filtrer les valeurs d'un échantillon
  - `std::replace_if(v.begin(),v.end(),filtre,-1);`
  - Prédicat «`filtre`» vérifié ⇒ remplacement par *-1*

- Capture des données de l'intervalle du filtre

```
int min = ...;
int max = ...;
...
auto filtre = [min,max] (int x)
              { return (x<min || x>max); };
```

- Variables capturées listées dans «`[]`»
  - Variable utilisée sans être capturée ⇒ erreur

Capture de variables *(2/5)*

- Deux types de captures
  - `[`*variable*`]` ⇒ capture par copie
  - `[&`*variable*`]` ⇒ capture par référence

- Capture par copie = copie de la variable capturée
  - Modification de la variable dans le contexte
    ⇒ aucun impact dans la lambda

- Exemple

```
int min = 5; int max = 7;
...
auto filtre = [min,max] (int x)
              { return (x<min || x>max); };
...
min = 3; max = 10;
...
replace_if(v.begin(),v.end(),filtre,-1);
```
⇒ filtre [5;7]

Capture de variables *(3/5)*

- Capture par référence = référence sur la variable capturée
  - Evite la copie (important pour les gros objets)
  - Modification de la variable dans le contexte ⇒ impact dans la lambda
  - Attention à la durée de vie des variables capturées par référence

- Exemple

```
int min = 5; int max = 7;
...
auto filtre = [&min,&max] (int x)
              { return (x<min || x>max); };
...
min = 3; max = 10;
...
replace_if(v.begin(),v.end(),filtre,-1);
```

⇒ filtre [3;10]

Capture de variables *(4/5)*

- Capture automatique possible
  - Variable utilisée ⇒ variable capturée
  - Seules les variables utilisées dans la lambda sont capturées
  - `[]` ⇒ aucune capture
  - `[=]` ⇒ capture automatique par copie
  - `[&]` ⇒ capture automatique par référence

- Exemple: capture automatique par copie

```
int min = 5;
int max = 7;
...
auto filtre = [=] (int x)
              { return (x<min || x>max); };
```

Capture de variables *(5/5)*

- Capture de «`this`»
  - `[this]` ⇒ capture du pointeur de l'objet du contexte

- Exemple

```
class Statistique {
 private:
  int min_;
  int max_;

 public:
  ...
  void filtrer(vector<int> & v) const {
   auto filtre = [this] (int x) {
    return (x < this->min_ || x > this->max_);
   };

   replace_if(v.begin(),v.end(),filtre,-1);
  }
};
```

Foncteur et capture *(1/2)*

- Lambda avec capture ⇒ implémentation d'un foncteur

- Exemple d'une capture par copie

```
class FoncteurCopie {
 private:
  int min;
  int max;

 public:
  FoncteurCopie(int a,int b) : min(a),max(b) {}

  bool operator () (int x) const {
   return (x<min || x>max);
  }
};
```

Foncteur et capture *(2/2)*

- Exemple d'une capture par référence

```
class FoncteurRef {
 private:
  int & min;
  int & max;

 public:
  FoncteurRef(int & a,int & b) : min(a),max(b) {}

  bool operator () (int x) const {
   return (x<min || x>max);
  }
};
```

Lambda constante

- Remarque: l'opérateur «**( )**» du foncteur est constant

- Rappel: dans une méthode constante…
  - Les attributs deviennent constants
  - Mais attention au cas des pointeurs/références
  - Les pointeurs/références sont constants mais pas les objets référencés !

- Par défaut, une lambda est «constante»
  ⇒ implémentation d'un foncteur avec opérateur «**( )**» constant

- Lambda constante
  ⇒ les variables capturées par copie sont constantes
  - Car les variables deviennent des attributs du foncteur
  - Capture par copie ⇒ attribut valeur ⇒ variable capturée constante
  - Capture par référence ⇒ attribut référence ⇒ variable capturée modifiable

Lambda non constante

- Lambda non constante ⇒ mot-clé «**mutable**»

  ⇒ Modification possible des variables capturées par copie

  ⇒ Foncteur avec opérateur «**()**» non constant

- Exemple: produire des nombres pairs

```
int cpt = 32;
...
auto gen = [cpt] (void) mutable {
 cpt += 2;
 return cpt;
};
...
std::generate(v.begin(),v.end(),gen);
```

- Attention: une lambda peut être un objet non constant
  - `template <typename LAMBDA>`
    `void algo(const LAMBDA &)` ⇒ erreur possible

Abstraction du type de fonction

- Trois manières de modéliser une fonction
  - Pointeur de fonction
    - Une méthode est considérée comme une fonction dont le 1er argument est le pointeur de l'objet
  - Foncteur
    - Objet avec opérateur « **( )** »
  - Lambda
    - Type inconnu
    - Implémentation comme fonction ou foncteur
- Types différents, mais même manière d'être appelés
- Comment faire abstraction de ces trois types ?
- Objectif: algorithme recevant indifféremment en paramètre un pointeur de fonction, un foncteur ou une lambda

Abstraction de fonction par généricité *(1/2)*

- 1ère approche: abstraction par un paramètre générique
  - Avantage: très efficace
    - Instanciation adaptée au type de modélisation
  - Inconvénient: difficile de contrôler le paramètre
    - Comment être sûr qu'il représente bien une fonction ?

- Passage par référence constante ?
  - `template <typename F> void algo(const F & fonc);`
  - Problème pour les lambdas/foncteurs non constants

- Passage par référence non constante ?
  - `template <typename F> void algo(F & fonc);`
  - Problème pour les *rvalues* ou les pointeurs de fonction
  - Et souvent une lambda est une *rvalue*: `algo([…] (…) {…});`

Abstraction de fonction par généricité *(2/2)*

- Passage par copie ⇒ inefficacité

- Solution: passage par référence universelle
  - `template <typename F> void algo(F && fonc);`
  - Accepte des lambdas/foncteurs constants ou non
  - Accepte des *lvalues* ou des *rvalues*

- Exemple

```
template <typename IT,typename GEN>
void generate(const IT & debut,const IT & fin,
              GEN && generer) {
 for (IT it = debut; it != fin; ++it)
  *it = generer();
}
```

Abstraction de fonction par adaptation

- 2nde approche: abstraction par un adaptateur ⇒ **`std::function`**
  - Avantage: meilleur contrôle du paramètre
    - Tout type ne peut pas être converti en «**`std::function`**»
  - Inconvénient: surcoût à l'exécution
    - L'abstraction est réalisée par héritage

- Exemple
  - ```
    template <typename IT,typename RET>
    void generate(const IT & debut,const IT & fin,
                  const std::function<RET(void)> & generer) {
     for (IT it = debut; it != fin; ++it)
      *it = generer();
    }
    ```
  - ```
    generate(v.begin(),v.end(),std::function<int(void)>(gen));
    ```
    - Remarque: conversion explicite nécessaire

«Rappel» de syntaxe

- Remarque sur l'utilisation de «`std::function`»
  - Forme peu courante du paramètre: `std::function<int(void)>`
  - `int (*)(void)` ⇒ pointeur de fonction
  - `int(void)` ⇒ ???

- Explication (valide en C++03)
  - `int(double,double)` ⇒ fonction
  - `int (*)(double,double)` ⇒ pointeur de fonction
  - `int (&)(double,double)` ⇒ référence de fonction

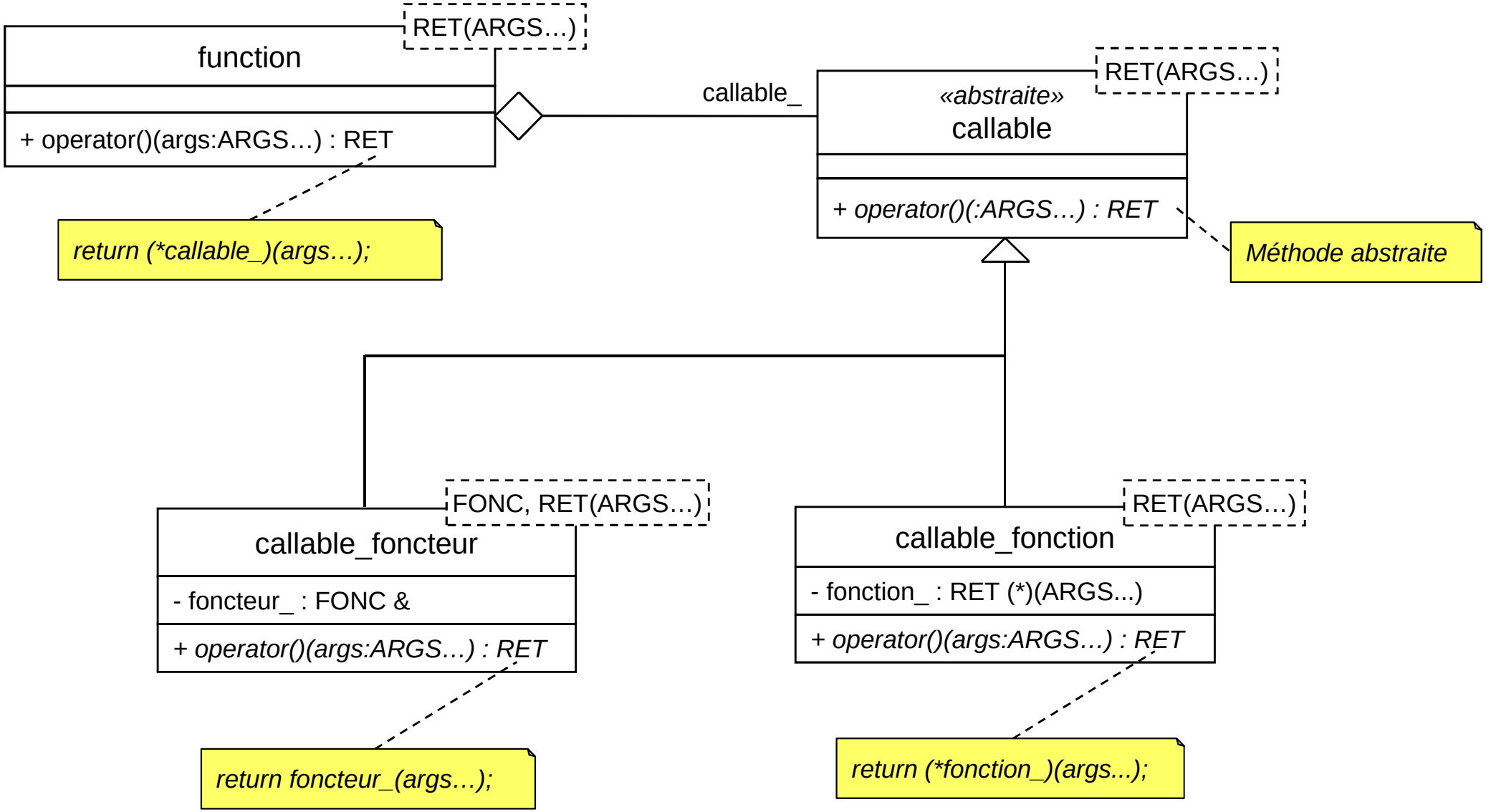
- Syntaxe rarement utilisée car l'intérêt est limité
  - `typedef int fonction_t(double,double);` ⇒ OK
  - `fonction_t x;` ⇒ OK (mais fonction sans corps)
  - `fonction_t x = f;` ⇒ erreur (affectation impossible)

Implémentation de «std::function» (1/6)

- Comment fonctionne «`std::function`» ?
- Attention: il s'agit d'une ébauche !
- Utilisation du patron de conception «Adaptateur / *Adapter*»
  - Implémentation par délégation
- Adaptateur: classe «`function`»
  - Agrège un objet «`callable`»
  - Délégation de l'opérateur «`()`»
- Adapté: classe abstraite «`callable`»
  - Représente un foncteur, une fonction ou une lambda ⇒ sous-classes

Code source complet: `cpp11_function.hpp`

Implémentation de «std::function» (2/6)



Implémentation de «std::function» (3/6)

- Adapté: classe abstraite
  - `callable<RET(ARGS...)>`

- Déclaration primaire (pour imposer un seul paramètre)
  - `template <typename> class callable;`

- Spécialisation (pour forcer la signature du paramètre)

```
template <typename RET,typename... ARGS>
class callable<RET(ARGS...)> {
 public:
  virtual RET operator () (ARGS... args) const = 0;
  virtual ~callable(void) {}
};
```

Implémentation de «std::function» (4/6)

- Adapté: sous-classe pour foncteur
  - `callable_foncteur<FONC,RET(ARGS...)>`

- Déclaration
  - `template <typename,typename> class callable_foncteur;`

  - ```
    template <typename FONC,typename RET,typename... ARGS>
    class callable_foncteur<FONC,RET(ARGS...)>
    : public callable<RET(ARGS...)> {
     private:
      FONC & foncteur_;

     public:
      callable_foncteur(FONC & f) : foncteur_(f) {}

      RET operator () (ARGS... args) const
      { return foncteur_(args...); }
    };
    ```
    ```

Implémentation de «std::function» (5/6)

- Adapté: sous-classe pour fonction
  - `callable_fonction<RET(ARGS...)>`

- Déclaration
  - `template <typename> class callable_fonction;`

  - ```
    template <typename RET,typename... ARGS>
    class callable_fonction<RET(ARGS...)>
    : public callable<RET(ARGS...)> {
     private:
      RET (*fonction_)(ARGS...);

     public:
      callable_fonction(RET (*f)(ARGS...)) : fonction_(f) {}

      RET operator () (ARGS... args) const
      { return (*fonction_)(args...); }
    };
    ```
    ```

Implémentation de «std::function» (6/6)

- Adaptateur

```
template <typename RET,typename... ARGS>
class function<RET(ARGS...)> {
 private:
  std::unique_ptr< callable<RET(ARGS...)> > callable_;

 public:
  template <typename FONC> function(FONC && fonc)
  : callable_(new callable_foncteur< remove_ref<FONC>,
                                     RET(ARGS...)
                                    >(fonc)) {}

  function(RET (*fonc)(ARGS...))
  : callable_(new callable_fonction<RET(ARGS...)>(fonc)) {}

  RET operator () (ARGS... args) const
  { return (*callable_)(args...); }
};
```