Ch. 7 - Les fonctions en C++

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

11 octobre 2021

Table des matières

Introduction

Caractéristiques

1. Même concept modulaire qu'en `C`
2. Possède des paramètres et un retour
   - `sqrt` prend en paramètre un flottant et retourne un flottant
3. Identifiées par leur nom et leurs paramètres
   - Les règles d'appel sont appliquées sur ces caractéristiques
   - *Pas* le type de retour
4. Concept indépendant de la POO
   - Fonctions *membres* (méthodes : `C++`)
   - Fonctions indépendantes
5. Plus qu'une fonction mathématique
   - Effectue un travail
   - Possibilité de modifier les paramètres
   - Peut ne rien retourner (`void`)

Arguments par défaut

- Jusqu'à présent, une fonction était appelée avec le même nombre d'arguments que son prototype en requérait
- C++ permet de spécifier la valeur de certains paramètres s'ils sont omis
  - `double f(int x = 0) ...`
  - `double d = f(); //same as f(0)`
- Les valeurs par défaut des paramètres sont spécifiés dans la déclaration de la fonction
  - Si séparation déclaration / implémentation et spécification dans les deux cas : erreur
- Très pratique pour les constructeurs de classe
  - Cf. Ch. 4

Contraintes

Règles

1. Les arguments par défaut ne peuvent utiliser des variables locales
   - En particulier les autres paramètres
2. Les arguments par défaut ne peuvent pas utiliser `this`
3. Les arguments par défaut sont les derniers de la liste de paramètres

- `void f(int i = 5, long l, int j = 3);`
- Appel de `f(10, 20)`
- Exécute `f(5, 10, 20)` ou `f(10, 20, 3)` ?

Fichier `param-def.cpp`

```
1  int k = 2;
2
3  void f(int i = 3)
4  {
5      cout << i << endl;
6  }
7
8  //void g(int n, int m = n * 2) {}
9
10 void h(int n, int m = k * 2, int p = 3)
11 {
12   cout << n << "␣" << m << p << endl;
13 }
14
15 int main()
16 {
17     f(2);
18     f();
19   //g(2);
20   h(2);
21 }
```

En C++

- Les pointeurs de fonctions « classiques » restent valides en C++
- `std::function` est un wrapper de pointeur de fonction
  - `std::function<ReturnType (parameters)> my_ptr`
  - `std::function<void (int)> ptr:ptr` est une fonction prenant en paramètre un `int` et retournant un `void`
- On peut créer des pointeurs de fonction membres (aka méthodes)
  - Utilisation de l'opérateur de résolution de portée `::`
- Le premier paramètre est *toujours* la classe de l'objet sur lequel on appelle la fonction
  - `std::function<void (Rectangle&, double)> f = &Rectangle::setX`
- On peut aussi utiliser une syntaxe transparente à l'aide de templates

Fonction indépendante

- Fichier `fct-ptr.cpp`

```
1  int fwd(std::function<int (int, int)> f, int a, int b)
2  {
3    return f(a, b);
4  }
5
6  int add(int a, int b)
7  {
8    return a + b;
9  }
10
11 int main()
12 {
13   cout << fwd(add, 2, 3) << endl;
14 }
```

Fonction membre

- Fichier `fct-ptr.cpp`

```
1  struct A
2  {
3    int i;
4    A(int i) : i(i) {}
5    int add(int j) { return i += j; }
6  };
7
8  int fwd(function<int (A&, int)> member, A& a, int j)
9  {
10   return member(a, j);
11 }
12
13 int main()
14 {
15   A a(2);
16   cout << fwd(&A::add, a, 3) << endl;
17 }
```

Passage d'argument

Passage par adresse

- On ne transmet pas une copie du paramètre, mais son adresse
  - « Comme en `Java` » pour les objets
- Permet d'émuler un passage par référence
  - En `C` pur, pas d'autre solution

Inconvénients par rapport aux références

- Plus « risqué »
  - Hygiène de programmation plus stricte
- Syntaxe « moins transparente »

- Parfois (rarement) pas d'autre choix en `C++`

Passage par référence

- On ne transmet pas une copie de l'objet, mais l'objet lui-même
- Utilisation du caractère `&` après le type
  - Ce paramètre est transmis par référence
  - Le standard ne spécifie pas leur implémentation (souvent des pointeurs constants)
- Offre des gains de performances
- Diverses conséquences
  - Synchronisation
  - Immédiats
  - Pas de conversions possibles à l'appel

Exemple

```
void swap(int&, int&);
```

Exemple

- Fichier `swap-ref.cpp`

```
1  void swap(int& x, int& y)
2  {
3    cout << "Entering swap : " << x << " " << y << endl;
4
5    int tmp = y;
6    y = x;
7    x = tmp;
8
9    cout << "Exiting swap : " << x << " " << y << endl;
10 }
11
12 int main()
13 {
14   int i = 1;
15   int j = 2;
16
17   cout << "Before call : " << i << " " << j << endl;
18   swap(i, j);
19   cout << "After call : " << i << " " << j << endl;
20 }
```

Retour d'une fonction

- Comme pour le passage de paramètre, le retour d'une fonction peut être effectué
  - par valeur (par défaut) : `int f();`
  - par adresse : `int* f();`
  - par référence : `int& f();`

Attention

- Ne créez pas de pointeurs / références vers des temporaires
- Ils vont « pendouiller » (dangling)

Illustration

- Fichier `return.cpp`

```
1  string f1()
2  {
3      string s = "Hello World!";
4      return s; //returns a copy of s
5  }
6
7  string& f2() {
8      string s = "Hello World!"; string & rs = s;
9      return rs;
10 }
11
12 string* f3() {
13     string s = "Hello World!"; string * rs = &s;
14     return rs;
15 }
16
17 int main() {
18     cout << f1() << endl;
19     cout << f2() << endl; //undefined behaviour
20     cout << *(f3()) << endl; //undefined behaviour
21 }
```

Fonctions lambda

Les lambdas

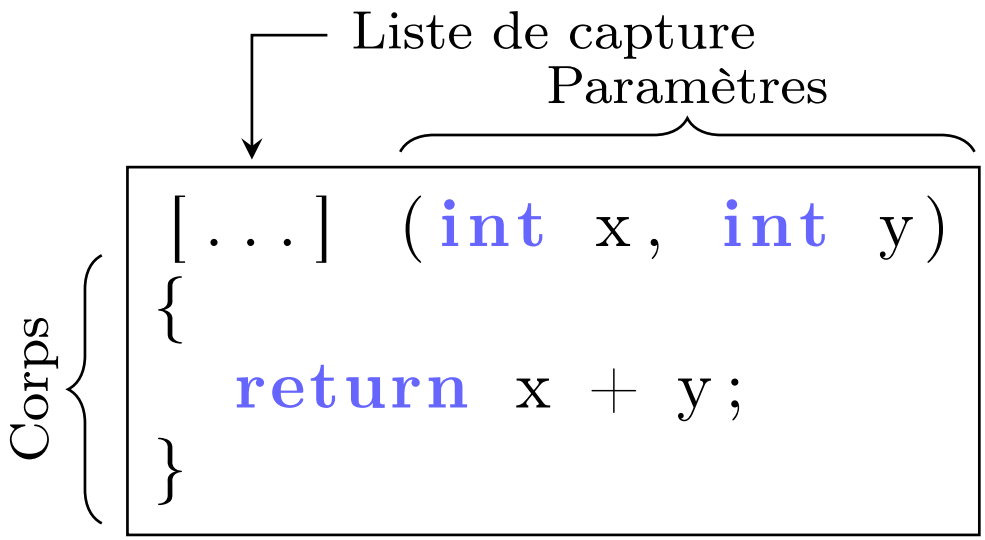
- Concept C++ uniquement

Idée de base

- Écrire des fonctions (locales) à la volée

- Motivation : écrire une fonction indépendante est « trop verbeux » quand les fonctions sont destinées à un usage unique
- Compilé comme un objet fonction avec une surcharge de l'opérateur `()`
  - Cf. Ch. 8
- On peut construire des lambdas
  - à la volée et les passer comme paramètres d'une fonction
  - en les affectant dans une variable pour les utiliser plusieurs fois au sein d'un bloc
    - Le type de la variable est systématiquement déterminé par `auto`

Syntaxe



- Les paramètres et le corps du fonction sont spécifiés comme ceux d'une fonction « habituelle »
- La liste de capture possède une syntaxe particulière, mais peut être vide

Exemple

- Fichier `lambda.cpp`
- `std::for_each` (`algorithm.h`) est une fonction appliquant une fonction donnée à tous les éléments d'un conteneur itérable

```
1  int main()
2  {
3    vector<int> v = {1, 2, 3, 4, 5};
4    for_each(v.begin(), v.end(), [](int& i) { i++; });
5    for_each(v.begin(), v.end(), [](int i) { cout << i << endl; });
6  }
```

- C'est « court »

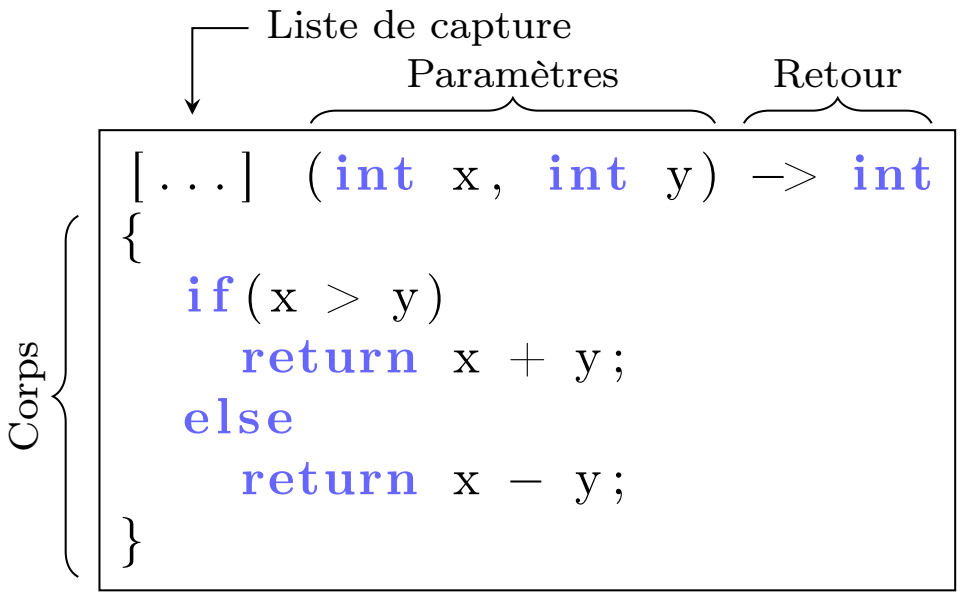
Remarque

- Avoir un code lisible est primordial
- Lambda courtes

Déduction du type de retour

- Dans l'exemple précédent, on n'a pas dû préciser le type de retour
- Il est « déduit » par le compilateur
- Si ce n'est pas possible (si `auto` ne le permet pas), il faut le préciser

Syntaxe



Liste de capture

- Par défaut, rien en dehors des paramètres de la lambda ne peut être utilisé dans son corps
- La liste de capture permet d'inclure des éléments « extérieurs »

Syntaxe

- `[x]` : la variable `x` est passée par valeur
- `[&x]` : la variable `x` est passée par référence
- `[=]` : toutes les variables du bloc de déclaration sont passées par valeur
- `[&]` : toutes les variables du bloc de déclaration sont passées par référence

- Possibilité de combinaison
  - Par exemple : `[x, &y]`

Exemple

- Fichier `lambda-ret.cpp`

```
1  struct A { int i; };
2
3  int main()
4  {
5    A a; a.i = 1;
6    A b; b.i = 2;
7
8    auto f = [&a, b] (int i) //generic lambda
9      {
10       int k = a.i + b.i + i;
11       a.i += 3;
12       //b.i += 3; //error, b is read-only
13
14       return k;
15     };
16
17   cout << f(4) << endl;
18   cout << a.i << " " << b.i << endl;
19 }
```

Initialisation dans la liste de capture

- En C++14, un élément de la liste de capture peut être initialisé

```
1  int main()
2  {
3    int x = 4;
4    auto f = [&r = x, x = x + 1]()->int
5         {//r is a x-reference, x is incremented
6             r += 2;
7             return x + 2;
8         };//();
9
10   int k = f(); //comment that and uncomment stuff before
11
12   cout << x << endl; //6
13   cout << k << endl; //7
14 }
```

Rappel

- Avoir un code lisible est primordial

Règles d'appel

Surdéfinition

- On parle de surdéfinition (overloading) quand un même symbole possède plusieurs significations
- Le choix du symbole dépend du contexte

Exemple : `a + b`

- Addition entière
- Addition flottante

- En C++, on peut surdéfinir des fonctions
  - Des règles d'appel sont mises en œuvre pour le choix de la fonction à appeler en cas « d'ambiguïté »
- En C, seul ne nom de la fonction à appeler intervient dans la recherche
  - Le seul overload existant est celui avec les opérateurs arithmétiques et le types de base

Étapes dans l'appel d'une fonction en C++

1. Name lookup
   - On recherche la définition d'un symbole
   - Pour des fonctions, cette recherche dépend du type des arguments
   - Pour des templates, il faut déduire le type des arguments (cf. Ch. 13)
2. Si ces étapes produisent plus d'une correspondance (surdéfinition), il faut résoudre l'ambiguïté
   - Overload resolution
   - En pratique, on cherche la « meilleure correspondance »
   - Les conversions ont un « score » (rank)
     - Les conversions de meilleur score sont privilégiées
     - Si on a le choix entre deux conversions de même score : erreur
3. Si elle ne peut pas être résolue : erreur
   - Aucune définition meilleure qu'une autre

Choix de fonction à appeler

- Idée : le compilateur cherche « la meilleure » correspondance possible

Règles d'appel

1. Correspondance exacte
   - Tous les types sont distingués
   - `const` intervient uniquement dans le cas de pointeurs et références
2. Correspondance de type « promotion »
   - Promotion entière, promotion flottante
3. Autres conversions
   - Conversion entière, conversion flottante, conversion `void*`, conversion définie par l'utilisateur (C++), etc.

Exemple

- Fichier `surdef.cpp`

```
1  int f(int i) {
2    cout << "Integer " << i << endl;
3    return 0;
4  }
5
6  //double f(int i) {} //return type matters not
7  //int f(const int i) {} //cv-qualifier lost
8
9  int f(double d) {
10   cout << "Double " << d << endl;
11   return 0;
12 }
13
14 int f(int i, int j) {
15   cout << "Integers " << i << " and " << j << endl;
16 }
17
18 int main() {
19   int k = 1;
20   f(k);
21   double d = 2.1;
22   f(d);
23   f(k,d);
24 }
```

- Impossible en C

rvalue et cv-qualifiers

Conversions de rvalue cv-qualifiée

- Une lvalue de type `T` qui n'est ni une fonction ni un tableau peut être convertie en rvalue.
  1. Si ce n'est pas une classe, le type de la rvalue est la version cv non qualifiée de `T`
  2. Sinon, le type de la rvalue est `T`

Conséquence

- Les cv-qualifiers sont perdus sur les arguments explicites
- Pas sur `this`

Exemple

- Fichier `const-call.cpp`
- Ne vous souciez pas de `struct`
  - « Comme une classe » (cf. Ch. 4)

```
1  struct A
2  {
3      void brol() const { cout << "A::brol() const" << endl; }
4      void brol() { cout << "A::brol()" << endl; }
5
6      void brol2(A) { cout << "A::brol2(A)" << endl; }
7
8      //ERROR : cv-qualifier lost
9      //void brol2(const A) { cout << "A::brol2(const A)" << endl; }
10 };
11
12 int main()
13 {
14     A a = A();//creates an A (rvalue to lvalue conv)
15     const A ca = A();//creates an A (rvalue to lvalue conv)
16
17     a.brol();  //brol
18     ca.brol(); //brol const
19
20     a.brol2(a);
21     a.brol2(ca);
22 }
```

Contraintes références

- Une fonction `void f(int&);` est appelée avec une référence
- *Doit* être une lvalue

Conséquences

- Pas d'immédiat
- Pas d'expression
- Pas de conversion ou tronquage

- Idée : sinon, on pourrait changer la valeur d'un temporaire, écrire `a*x+b = y;`, etc.
- Différent avec `const`

Exemple

- Fichier `ref-cstr.cpp`

```
1  void f(int&){}
2
3  int main()
4  {
5    const int n = 15;
6    int q;
7    f(q);
8    //f(2*q+3);//not a lvalue
9    //f(3);
10   //f(n);//wrong cv-qualifier
11
12   float x;
13   //f(x);//no truncation
14
15   short k;
16   //f(k);//no conversion
17 }
```

Exceptions

- Le prototype d'une fonction peut prendre par référence des paramètres constants

Exemple

- `void f(const int& n);`

- Ici, il est prévu que les paramètres soient constants, et donc non modifiables
- Permet un appel de `f` sur *toute* expression entière
  - `f(2)`, `f(3 * n)`, etc.
  - Conversions placées dans des variables temporaires transmises par référence
  - Risque de modification de temporaire supprimé

Exemple

- Fichier `surdef-ref.cpp`

```
1  void f(int & i)
2  {
3    cout << "Ref " << i << endl;
4  }
5
6  void f(const int & i)
7  {
8    cout << "Ref cst " << i << endl;
9  }
10
11 int main()
12 {
13   int n = 3;
14   const int m = 5;
15   f(n);
16   f(3);
17   f(int{4});
18   f(4 * n);
19   f(4 * m);
20   f(m);
21 }
```

Le cas des pointeurs

- Fichier `surdef-ptr.cpp`

```
1  //void brol(char*) { cout << "Brol char" << endl; }
2  void brol(double*) { cout << "Brol double" << endl; }
3  void brol(void*) { cout << "Brol void" << endl; }
4  void brol(int*) { cout << "Brol int" << endl; }
5  //void brol(int* const) { cout << "int const" << endl; } //try to uncomment that
6  void brol(const int*) { cout << "Const int" << endl; }
7
8  int main()
9  {
10   char * ptc;
11   double * ptd;
12   void* ptv;
13
14   brol(ptc);// char, try when remove brol(char*) -> void
15   brol(ptd);// double
16   brol(ptv);//remove brol(void*) for this call : ERROR (no conv)
17
18   int n = 3;
19   const int p = 5;
20
21   brol(&n);
22   brol(&p);
23 }
```