Les fonctions Lambda de C++11

Ou les fonctions qui n'en sont pas

Un commentaire

Avec la spécification récente **C++11**, autrement dit la version qui fait évaluer grandement le langage C++, est apparu l'expression **lambda**. On les appelle aussi les **closures**. Mais qu'est-ce c'est exactement ?

1. Définition

La fonction lambda est une fonction **inline** créée directement dans votre code, éventuellement passée directement dans les paramètres d'une de vos méthodes. Au lieu d'écrire une fonction que vous appelez dans les paramètres d'une méthode, vous placez ainsi directement le corps de cette **fonction anonyme** dans les paramètres d'entrées de votre fonction/méthode. Cela vous évite de créer une fonction à part et de l'appeler.

2. Exemple d'utilisation

Vous souhaitez créer une méthode qui permet l'affichage d'un widget, associé à une animation graphique. Exemple : dès que le widget s'affiche, un effet d'animation sera rendu (on peut imaginer faire *trembler* notre

bouton à l'affichage). On peut penser embarquer le mécanisme d'animation dans une fonction lambda, pourquoi pas asynchrone.

On peut aussi imaginer créer une fonction qui envoie des mails, dans ses paramètres une fonction lambda permettrait de contrôler la validité du mail. Etc.

Les cas possibles sont très nombreux.

3. Avantages et pourquoi c'est une bonne chose

– La fonction est inline et donc pas d'appel en langage machine (pas de call) à une fonction avec les contraintes bien connues de travaux sur la pile à l'appel et au retour : **plus de performance**.

– La fonction **est privée à votre code**, elle ne sera pas publiée dans un fichier header.

– Pour des fonctions courtes, elle évite la création de fonctions, de publier une déclaration, **code plus concis**.

4. Syntaxe basique

```
1  #include <iostream>
2  int main() {
3     auto p = [] () { cout << "Je suis une fonction lam
4     p(); // Appel de la fonction
5     return 0;
6  }
```

auto : Permet de créer une variable automatique, le type est déduit de l'initialisation.

[] : Dit au compilateur qu'il s'agit d'un type lambda et peut comporter des spécifications concernant le passage des paramètres qui vont suivre.

() : On décrit ici les paramètres de la fonction lambda (ici rien, void).

{ ... } : Le code de la fonction lambda.

5. Autres exemples

```
1  // 1.
2  #include <iostream>
3  int main() {
```

```
    int a = 2;
    auto p = [&a] () { return a*a; };
    p(); // Appel de la fonction
    return 0;
}
```

Ici on déclare une variable locale déclarée sur la pile : a, qu'on initialise avec la la valeur 2. La syntaxe [&a] indique que la fonction lambda utilisera a par adresse. La fonction multiplie la donnée et retourne un entier. A noter que le compilateur résout le retour automatiquement, le retour par défaut d'une fonction lambda est void. On peut également spécifier le type de retour avec la syntaxe suivante : [] () -> int { return a*a; };

```
// 2.
#include <iostream>
int main() {
    int aa = 2;
    int bb = 3;
    auto ajouter = [=] (int aa, int bb) { return aa
    ajouter(1,2);

    return 0;
}
```

Autre exemple : Nous avons ici une fonction lambda qui attend 2 paramètres en entrée du type entier. On remarque dans le spécificateur de format d'entrée, **[=]** qui indique que les paramètres suivant sont passés par copie.

[&] : passage par adresse

[=] : passage par copie

[&aa,=] : 1er paramètre passé par adresse, second par copie

6. Exemple d'utilisation avec la STL

Voici une illustration de tri grâce à la classe vector.

```
vector v;
v.push_back( 50 );
v.push_back( -10 );
v.push_back( 20 );
v.push_back( -30 );

    // lambdas
std::sort( v.begin(), v.end(), [](int a, int b) { re
for( int t = 0; t &lt; v.size(); t++ ) {
       cout << v[t] << " ";
}
```

Ci-dessus on initialise un vector avec certaines valeurs. Dans la fonction de tri en dernier paramètre on appelle la fonction lambda au sein du passage des paramètres.

7. Conclusion

Les lambdas manquaient cruellement au C++, c'est un artifice de programmation bien utile. Java avait déjà son équivalent avec ses méthodes anonymes et *inner*, JavaScript avec ses *closures*, Objective-C avec les *codes blocks*.

Si cet article vous a plu, n'hésitez pas à le partager en cliquant sur les boutons ci-dessous.

Au sujet du formateur

Bertrand Leclercq

Anime réguliérement la formation C+ perfectionnement chez ORSYS et chez Dassault Systèmes

CONTACTER LE FORMATEUR

C++ avancé, *Tutoriaux*

c++ ▸ *Tutoriaux* ▸

- One Comment
- 0 Pings & Trackbacks

adonis simo
février 11, 2017 à 11:24

Merci a vous pour cet article tres constructif, j'en recherchai depuis