Paul Clas 1846912

Travail personnel :

Les fonctions lambda en C++. Capture et fermeture

LOG 2410 Conception Logiciel

Présenté à :

François Guibault

Remis le

7 Avril 2019

Département de génie informatique et génie logiciel

Polytechnique Montréal

1. Introduction

Imaginez que vous avez un bottin téléphonique et vous voulez être capable d'utiliser des fonctions pour faire des recherches dans celui-ci. Nos cours de programmation orientée objet à Polytechnique Montréal en INF1005C et INF1010 nous ont bien appris à créer nos propres fonctions qui prennent en paramètres un string et retourne tous les numéros de téléphone associé à ce string. Quelquefois, c'est l'unique fonctionnalité de la classe que l'utilisateur veut pouvoir utiliser; l'option de recherche à travers une liste, tableau, etc. Il existe alors beaucoup de solution potentiel que nous pourrions implémenter pour permettre une telle fonctionnalité. Une des solutions les plus simples à implémenter et qui a l'avantage d'être inline, c'est-à-dire directement dans le code, sont les fonctions lambda. La solution à ce problème que nous allons étudier dans ce rapport sont les fonctions lambda, leurs captures et leur classe fermeture.

Il s'agira donc de définir les fonctions lambda introduit dans la spécification de C++11 et améliorer dans les plus récente spécifications. Nous explorerons les motivations derrières celles-ci ainsi que leur fonctionnement grâce à la leur capture et leur classe fermeture. Nous décrirons l'utilité des classes de fermeture, les différents modes de capture et les modes à privilégiés. Enfin, nous discuterons des problèmes potentiel qui peuvent découler du mauvais choix de capture et des règles qui devraient être suivit pour la capture des fonction lambda.

2. Les fonctions lambda en C++

Utilisé dans plusieurs langages de programmation comme Java, C# et Python. Les fonctions lambdas nous viennent des maths grâce à l'invention de « Lambda Calculus » de Alonzo Church dans les années 1930 [Sussman, 1975]. La fonction lambda est une fonction anonyme, sans nom, qui construit une fermeture, un objet capable de capturer des variables dans sa portée. À part cela, une fonction lambda se comporte comme une fonction classique. Celle-ci peut recevoir des arguments et retourner ou non une valeur. Il existe plusieurs façons d'écrire des fonctions lambda toutefois, comme nous le

préciserons plus tard, comme nous utilisons les fonctions lambda avec les algorithmes de la bibliothèques STL, il est nécessaire de respecter la signature imposée par ces algorithmes.

Voici des exemples de la structure de déclarations à respecter lors de l'implémentation de fonction lambda :

```
[capture](paramètres) mutable exception attribute->ret{ body };
// Exemple Déclaration complète

[capture] {paramètres}->ret {body}
// Déclaration d'un lambda const: les objets capturés par copie ne peuvent pas
// être modifiés

[capture] {paramètres} {body}
// Déclaration dans type de retour. le type de retour de la fermeture
// operator() est déduite d'après le body:
//          - Si le body est composé de la mention return unique, le type
//              de retour est le type de l'expression retournée
//          - Si le body n'est pas composé de la mention return, alors
//              le type de retour est void.

[capture]{body}
// Déclaration avec la liste des paramètres omis: la fonction ne prend aucun
// argument, comme si la liste des paramètres est {}
```

Figure 1 : Exemple de déclaration classique de fonction lambda

a. Utilité de la classe de fermeture

Une fonction lambda n'est qu'en fait qu'une simple expression qui n'existe que dans le code source d'un programme. Une fonction lambda n'existe pas, à proprement parler, durant le temps d'exécution. En faites, celle-ci génère un objet durant le temps d'exécution et c'est cet objet que l'on appelle une fermeture. Une fonction lambda est une instance de la classe fermeture.

La classe fermeture est donc créée pour chaque instanciation d'une fonction lambda. Pour ce faire la fermeture surcharge l'operator() pour qu'elle agisse tout comme une fonction. Lorsque la classe est construite, toutes les variables dans l'environnement

lexical de la fonction sont passés dans le constructeur et stocké en mémoire comme des variables membres. L'environnement lexical d'une fonction est l'ensemble des variables non locales qu'elle a capturé, soit par valeur (c'est-à-dire par copie des valeurs des variables), soit par référence (c'est-à-dire par copie des adresses mémoires des variables) [Belz, 2019]. D'ailleurs nous allons discuter plus tard dans ce rapport des différences entre ces deux modes de capture grâce à la classe fermeture.

L'utilisation du champ lexical ainsi que l'utilisation des algorithmes de la bibliothèques STL et la surcharge d'operator() ressemble beaucoup à une autre abstraction de fonction que nous avons appris la session dernière en programmation orientée-objet comme des foncteurs. Pour ne pas rentrer trop dans les détails car il ne s'agit pas du sujet de ce rapport, on pourrait dire que les fonctions lambda et les foncteurs sont beaucoup plus similaires que différent. L'analyse du code assembleur par le compilateur révèle peu de différence dans le nombre de « mov » et de registre utilisé. [Sussman, 1975]. Plus de recherches approfondies semble démontré que les fonctions lambda sont légèrement plus rapides mais prendrais un peu plus de bytes sur la pile(stack) [Sutter, 2019]. Pour un langage C++ qui prône la rapidité d'exécution ceci pourrait avoir des répercussions La différence entre les fonctions lambda et les foncteurs se situent principalement au niveau de la facilité d'utilisation des lambda (parce qu'elles sont définies inline et sont facile à lire) et de l'ancienneté des foncteurs par rapport aux fonctions lambda.

L'utilité de la classe fermeture est que celle-ci n'a pas besoin d'être explicitement construite. Grâce à la fermeture, le type des membres n'a pas nécessairement besoin d'être spécifié. Si les membres sont stockés en mémoire par copie, leur « nom » n'a pas besoin d'être spécifié. Un constructeur qui initialise les membres de la clôture non pas besoin d'être défini. Il est a noté que la metafonction « remove reference » convertit les références directement à leur valeur. Ceci est fait avec la réutilisabilité comme concept de design comme cela les membres dans la fermeture sont toujours stockés dans la fermeture par copie à part si spécifié en référence grâce à l'utilisation de « extern ».

L'utilisation de la classe fermeture est un choix de conception important en C++ pour l'utilisation des fonctions lambda car elle permet aux variables de l'environnement lexical d'être utilisés dans le corps de la fonction lambda [Samko, 2006]. Grâce à la fermeture nous pouvons utiliser toutes les fonctions lambdas et algorithme de la STL. Étant donné que les variables locales peuvent être utilisé dans le corps dans la fonction lambda, la prochaine section permet de comprendre comment la classe fermeture capture les variables. L'utilité de la classe fermeture est fondamentalement de permettre la capture.

b. Différence entre les différents modes de capture

C++ étant un langage de programmation intensif sur la performance, l'utilisation de la fermeture permet beaucoup de flexibilité par rapport aux variables que l'on pourra capturer avec les fonctions lambda. Par défaut, une fonction lambda est en totale isolation du reste du code. Une fonction lambda ne peut pas accéder aux variables de la fonction dans laquelle elle est écrite. L'exception à cette règle est lorsqu'on utilise la zone de capture pour dire à quels objets la fonction lambda peut accéder. La capture avec les fonctions lambda est assurée grâce aux spécifications donnée dans les crochets [ ]. Le tableau suivant montre toutes les options disponibles lors de l'implémentation de fonction lambda.

[]	Ne capture rien.
[&]	Capture les variables par référence.
[=]	Capture les variables par copie.
[=, &varB]	Capture les variables par copie mais capture la variable varB par référence.
[varB]	Capture varB par copie.
[this]	Capture le pointeur de la classe encapsulante.

Tableau 1: Option de capture pour les fonctions lambda

Il est a noté que la dernière option de capture ne nécessite pas de définir par copie ou référence, la capture du pointeur this se fait par défaut en copie. Cela permet de ne pas à devoir faire une distinction entre des variables locales et le champ de la classe lors de l'écriture de la fonction lambda. On peut accéder aux deux.

On parle donc de zone de capture car la fonction lambda a capturé notre variable, afin de l'utiliser. On continu l'analyse des différentes modes de capture de la fonction lambda en s'intéressant à la capture par copie et par référence. En effet on remarque les fonctions classiques, les foncteurs et les fonctions lambda se comporte exactement pareil lorsqu'il n'y a rien à capturer.

i. La capture par copie

```cpp
#include <iostream>

int main()
{
    bool const estNumeroCanadien{ true };
    int const numbTelephone{ 5146549856 };
    string const nom{ "Jason Bourne" };

    auto lambda = [estNumeroCanadien, nom, numbTelephone]() -> void
    {
        std::cout << "Le numero est canadien: " << std::boolalpha << un_booleen << "." << std::endl;
        std::cout << "Le numero de telephone de  "<< nom << " est " << numbTelephone <<" ."<< std::endl;
    };

    lambda();

    return 0;
}
```

Figure 2 : Code qui démontre une capture par copie

Cette capture est dite par valeur, ou par copie, car chaque variable est copiée. Il est a noté que si l'on supprime estNumeroCanadien de la zone de capture, le code ne compile plus sur Visual Studio. Apparemment, cette erreur de compilation est seulement attribuable à Visual Studio 2017 et n'arrive pas lorsqu'on compile sur CygWin. Selon la norme, il semble y avoir effectivement certains cas où ce n'est pas obligatoire de capturer une variable pour l'utiliser.

On peut également capturer plusieurs variables à la fois comme l'exemple précédant le démontre, en reprenant l'idée du bottin Téléphonique.

Il faut faire remarquer que nous ne pouvons pas, par défaut, modifier une variable capturée par copie. Il faudra alors utiliser le mot-clef mutable. On placera mutable après la liste des arguments et avant le type de retour, comme montré dans la Figure 1 au début du rapport, pour rendre la modification possible.

ii. La capture par référence

Le principe est exactement le même, mais cette fois, on a affaire directement à la variable capturée et non à sa copie et pour ce faire, on rajoute l'esperluette & devant. Cela permet, comme pour les paramètres, d'éviter une copie potentiellement lourde, ou bien de sauvegarder les modifications faites à la variable.

```
#include ...
using namespace std;


int main() {

    map<string, int> bottinTelephone;
    double n, m, value;
    string key, testKey;
    cin >> n;

    for_each(values.begin(), values.end(), [&](string& key, string& value) {
        bottinTelephone.insert(pair<string, double>(key, value));
    });


    for_each(cbegin(bottinTelephone), cend(bottinTelephone), [key, &value](string testKey) -> void
    {
        if (testkey == bottinTelephone->first)
        {
            cout << bottinTelephone->first << " " << bottinTelephone->second<< endl;
        }
    });

    return 0;
}
```

Figure 3 : Code qui démontre une capture par référence

Dans cet exemple de code, nous avons implémenter deux fonctions lambda. La première créer un map<string, int> bottinTelephone avec un nom comme key et le numéro de téléphone. Le deuxième for_each prends un nom rentré en std::in dans la console et retourne le numéro de téléphone associé à ce nom.

c. Les modes à privilégier

Le problème de design majeur des fonctions lambda est comment gérer les références pour les variables définie à l'extérieur de l'environnement lexical de la fonction lambda. Les deux façons vues précédemment, soit le passage par copie et le passage par référence ont leur propre problème potentiel.

Passer des variables locales par références créer des fermetures de « seconde classe » : cette fermeture est inutilisable en dehors de la fonction qui l'a créé. Cette situation est problématique car une des utilisations des fonctions lambda est la fonction de « callback ». Une expression lambda qui ne réfère pas à une variable pas référence est toujours utilisable dans le contexte du programme. Toutefois, stocké des variables par copie a son propre lot de problème. Par exemple, c'est possible de créer des fonctions lambda qui ne peuvent pas être utilisées en dehors de la déclaration de la fonction, même si les copies des variables sont stockées dans la fermeture. Des variables locales utilisés est stockés en copie dans la fermeture sont des pointeurs vers d'autre variable locale. [Samko, 2006].

On peut donc conclure qu'il n'y a pas de mode à privilégier à strictement parler car cela dépend de l'utilisation que l'on veut en faire. Le type de paramètre que l'on veut utiliser n'a d'importance que si celui fait entrainerait un cout trop grand de calcul par copie. Typiquement cependant, si nous voulions changer la variable, nous utiliserions la capture par valeur avec mutable. Pour de l'efficacité et rapidité, nous passerions la variable par référence en mettant un const devant.

d. Problèmes potentiels et Règles à suivre

Un des problèmes des lambda en C++ c'est que celle-ci ne sont pas capable de résoudre des problèmes de hiérarchie d'appel (upward funarg ou downward funarg), c'est-à-dire quand des variables locales sont capturés par référence par la fonction lambda et que celle-ci survivent (par exemple quand elles sont retournées par la fonction ou alors qu'on les stocke ailleurs). Lorsque la fonction Lambda capture la variable par référence, le langage C++ stocke l'adresse de la référence dans le stack et puisque le langage C++ ne possède pas de « garbage collector » si la variable est réutilisée plus tard elle n'existera plus (« undefined ») [Samko, 2006].

Un autre problème avec l'utilisation des lambda et si nous avons besoin de partager des variables avec deux fonctions lambda. Par exemple si nous voulions utiliser à la fois les fonctionnalités de la STL avec une fonction lambda pour incrémenter et décrémenter sur les mêmes variables capturées alors la capture par copie ne marcherait pas. Il faut alors utiliser le pointeur std ::shared_ptr.

Lors de la capture par référence, la fonction lambda est capable de modifier la variable locale en dehors de la fonction lambda, ce qui est intuitif puisqu'on passe la variable par référence mais ce si veut également dire que si l'on veut retourner une fonction lambda d'une autre fonction, nous ne devrions pas utiliser la capture par référence car la référence ne sera pas valide après que la fonction retourne son type.

La syntaxe de la capture dans le corps de la fonction lambda peut engendrer des problèmes potentiels. Voici un tableau qui résume la bonne syntaxe pour écrire des captures. Ceci constitue aussi les règles à suivre lors de l'écriture de fonction lambda.

Passage par copie		
[=]{};	Fonctionne	Capture par copie par défaut
[=,&i]{};	Fonctionne	Capture par copie par défaut, i capturé par référence
[=, *this]{};	Erreur	

[=, this]{};	Erreur	
Passage par référence		
[&]{};	Fonctionne	Capture par référence de défaut
[&, i]{};	Fonctionne	Référence par défaut et i est capturé par copie
[&, &i]{};	Erreur	
[&, this{};	Fonctionne	Équivalent à [&]
[&, this, i]{}	Fonctionne	Équivalent à [&,i]{}

Tableau 2.0 : Syntaxe pour la capture dans les fonctions lambda.

Finalement, dans notre expérimentation avec les fonctions lambdas, nous ne pouvons pas utiliser deux fois la même capture. Ainsi [i, i]{} n'est pas valide, tout comme [this, *this]{} [Sussman, 1975].

3. Conclusion

Les lambdas sont des fonctions anonymes, destinées à être utilisées localement. Les fonctions lambdas sont générées inline. Elles réduisent la taille du code, le rendent plus lisible et évitent de créer des fonctions à portée globale alors qu'elles sont à usage local. Les fonctions lambda créent une classe fermeture qui permet de capturer de façon très flexible des variables. Les paramètres peuvent être explicitement typés ou automatiquement déduits en utilisant auto.

La véritable nouveauté des lambdas, par rapport aux fonctions, est la zone de capture, qui permet de manipuler, par référence et/ou copie des objets de la portée englobante. On peut capturer explicitement ou bien certaines variables, ou capturer toutes celles de la portée englobante. La syntaxe de ces fonctions est très importante et beaucoup de « run time errors » peuvent être évité en cernant le bon mode de capture soit par valeur ou par référence. Il faut aussi faire attention lors de la réutilisation des variables passés dans la capture car dépendant de leur passage par référence ou copie, celle-ci ne pourrait pas encore exister après la sortie de la fonction.

L’étude des fonctions lambda nous a permis de jeter également un regard plus critique sur les foncteurs. Il serait intéressant d’étudier la différence fonctionnel entre les foncteurs et les fonctions lambda et de dresser les situations où il serait plus avantageux d’utiliser l’une ou l’autre de ces fonctions.

4. Bibliographie

CPPreference, Lambda functions (depuis C++11) https://fr.cppreference.com/w/cpp -/language/lambda, Avril 2019.

Guillaume Belz, Les foncteur et les fonctions Lambda, http://guillaume.belz.free .fr/doku.php?id=predicats, Avril 2019.

Herb Sutter. GotW #58: Nested functions. http://www.gotw.ca/gotw/058.htm, Avril 2019.

Herb Sutter, Elements of Modern C++ Style, https://herbsutter.com/elements-of-modern-c-style/, Avril 2019.

Jaakko Jarvi, Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4), 29 février 2008.

Oleg Kiselyov. Genuine lambda-abstractions in C++. http://okmij.org/ftp/c++-digest/#lambda-abstr, Avril 2019.

Sussman and Steele, Section 4: Some Implementation Issues, December 22, 1975, https://en.wikisource.org/wiki/Page:Scheme_An_interpreter_for_extended_lambda_calculus.djvu/22, Avril 2019.

Valentin Samko, ISO C++, A proposal to add lambda functions to the C++ standard, 23 février 2006