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 à 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 la plus récente spécification C++14. Nous explorerons les motivations derrières celles-ci ainsi que leur fonctionnement grâce à la leurs 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.

1. **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. 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 :

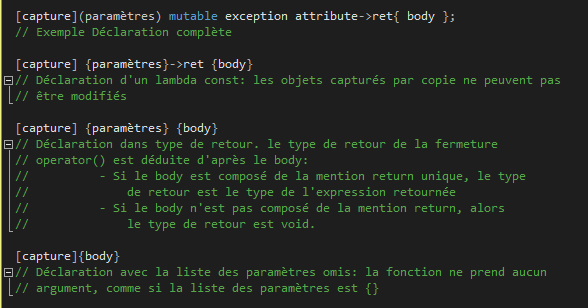


Figure 1 : Exemple de déclaration classique de fonction lambda

* 1. **Utilité de la classe de fermeture**

Une fonction lambda es 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) \*\*\*\*\*\*\*\*. D’ailleurs nous allons discuter plus tard dans ce rapport les différences entre ces deux modes de capture grâce à la classe fermeture.

L’utilisation du champs 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 de code assembleur par le compilateur révèle peu de différence dans le nombre de « mov » et de registre utilisé. 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)\*\*\*\*\*. 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 constance 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. 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.

* 1. **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. La capture avec les fonctions lambda est assurer 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 continu l’analyse des différentes modes de capture de la fonction lambda en s’intéressant à la capture par valeur 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.

* + 1. **La capture par copie**

Lors de la capture par copie/valeur

* + 1. **La capture par référence**

1. Both ways of storing variables into closures, by copy and by reference, are problematic in their own ways. Storing local variables by reference leads to closure objects being second-class: unusable outside the function which created them. This would be a dangerous default, since one of the anticipated common uses of lambda expressions is callbacks. A lambda expression which does not refer to any non-copied local variables is usable in any program
2. context.
   1. **Les modes à privilégier**

The major design issue with lambda expressions is how to handle references to variables defined outside of the body

of the lambda expression. Both of the design choices discussed above (storing the variables either by copy or by

reference) have their problems. Another viable design choice is to disallow references to local variables defined

outside of the lambda function’s body altogether, and allow the member variables of the closure to be explicitly

declared and initialized as part of the definition of the lambda expression.

Both ways of storing variables into closures, by copy and by reference, are problematic in their own ways. Storing local variables by reference leads to closure objects being second-class: unusable outside the function which created them. This would be a dangerous default, since one of the anticipated common uses of lambda expressions is callbacks. A lambda expression which does not refer to any non-copied local variables is usable in any program

context.

Storing local variables by copy has other problems. For example, it is possible to create lambda functions which cannot be used outside their enclosing function, even if copies of the variables are stored in the closures. One example of this is if a local variable used (and stored by copy) in the closure is a pointer to another local variable.

The pointee’s lifetime is shorter than the lifetime of the closure, and so a dangling pointer if the closure escapes its enclosing function. References do not have this problem, because the object referred to is copied into the closure if the reference is used in the lambda expression’s body. Slicing is also a possible problem, as a reference can refer to an object of a dynamic type derived from its static type; copying the referenced object using its static type would lead to only some of the actual object being copied. Copying of certain kinds of objects can also be very slow, which may not be expected to occur just from referring to a variable inside a lambda expression.

Note that some lambda functions will not compile with our chosen default. For example, streams are non-copyable

types, and thus the following code is erroneous:

ofstream os(”file.txt”);

hi(**int** i) {os << i;}

The correct definition is:

ofstream os(”file.txt”);

hi(**int** i) **extern**(os) {os << i;}

* 1. **Problèmes potentiels**

**Run time error detection**

**Most of such problems may be detected at run time by the implementation (many compilers already have options to add buffer overrun and other run time error detection). For example, if a lambda is defined in a function and its body refers to an object defined with automatic storage in that function, then implementation may store a counter of all the lambda objects of that type created by that function call or by copy construction, and if that counter is non zero when storage for local variables in that function is released, then the error may be reported at run time.**

**VOIR LE SLIDE SUR LINKEDIN**

The problem is that C++ lambdas are not able to solve the "[upwards funarg](https://en.wikipedia.org/wiki/Funarg_problem)" case, i.e. when a local variable is captured by reference by a lambda that survives the "owner" of the local (because it's stored somewhere or because it returned as function result).

What C++ does when a lambda captures a variable by reference is just storing the address of the referenced variable in a context structure that is used by the lambda code.

The variable itself however in your case is a local and lives in the stack frame where the lambda was created. If the lambda object is just called during the execution of someOtherFunction then things are fine, but if the lambda is instead **stored away** and survives the stack frame that created it (or to be more precise that created the captured variable referenced in the lambda), when it's executed will reference a variable that doesn't exist any more (undefined behavior).

Solving the "upwards funarg" problem in the general case requires a garbage collector, and C++ doesn't have one.

What you can do in some cases is capturing "by value", so the lambda will have its own private copy:

foo([counter]() mutable { counter++; })

In this case however if you want to change the captured copy you also need to use the mutablekeyword because ... well, just because this is what C++ requires if you want to modify a captured copy (captured copies are otherwise const objects in the body of the lambda).

Unfortunately if you need to share captured variables for example with two lambdas (e.g. creating both an "incrementer" and a "decrementer" on the same captured variable) the using a copy is not viable. What you can do for this is capturing by value a std::shared\_ptr to the variable and this will correctly replace the garbage collector for simple cases (not in the case of reference loops, however).

**Dangers and Benefits of Capture by Reference**

When you capture by reference, the lambda function is capable of modifying the local variable outside the lambda function--it is, after all, a reference. But this also means that if you return a lamba function from a function, you shouldn't use capture-by-reference because the reference will not be valid after the function returns.

* 1. **Règles à suivre**

1. **Conclusion**
2. **Bibliographie**
3. **Annexe**