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 lambdas. La solution à ce problème que nous allons étudier dans ce rapport sont les fonctions lambdas, 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 lambdas 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 lambdas :

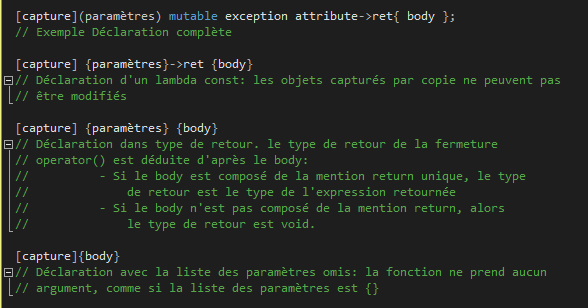


Figure 1 : Exemple de déclaration classique de fonction lambda

Well, the corporate marketoids of Sun/Oracle, Microsoft, Google etc. are to blame, because that's what they called these constructs in their languages (Java, C#, Go etc.). They often call "closures" what are supposed to be just lambdas. Or they call "closures" a particular technique they used to implement lexical scoping, that is, the fact that a function can access the variables that were defined in its outer scope at the time of its definition. They often say that the function "encloses" these variables, that is, captures them into some data structure to save them from being destroyed after the outer function finishes executing. But this is just made-up *post factum* "folklore etymology" and marketing, which only makes things more confusing, because every language vendor uses its own terminology.

And it's even worse because of the fact that there's always a bit of truth in what they say, which does not allow you to easily dismiss it as false :P Let me explain:

If you want to implement a language that uses lambdas as first-class citizens, you need to allow them to use symbols defined in their surrounding context (that is, to use free variables in your lambdas). And these symbols must be there even when the surrounding function returns. The problem is that these symbols are bound to some local storage of the function (usually on the call stack), which won't be there anymore when the function returns. Therefore, in order for a lambda to work the way you expect, you need to somehow "capture" all these free variables from its outer context and save them for later, even when the outer context will be gone. That is, you need to find the *closure* of your lambda (all these external variables it uses) and store it somewhere else (either by making a copy, or by preparing space for them upfront, somewhere else than on the stack). The actual method you use to achieve this goal is an "implementation detail" of your language. What's important here is the *closure*, which is the set of *free variables* from the *environment* of your lambda that need to be saved somewhere.

It didn't took too long for people to start calling the actual data structure they use in their language's implementations to implement closure as the "closure" itself. The structure usually looks something like this:

Closure {

[pointer to the lambda function's machine code],

[pointer to the lambda function's environment]

}

and these data structures are being passed around as parameters to other functions, returned from functions, and stored in variables, to represent lambdas, and allowing them to access their enclosing environment as well as the machine code to run in that context. But it's just a way (one of many) to *implement* closure, not *the* closure itself.

As I explained above, the closure of a lambda expression is the subset of definitions in its environment that give values to the free variables contained in that lambda expression, effectively *closing* the expression (turning an *open* lambda expression, which cannot be evaluated yet, into a *closed* lambda expression, which can then be evaluated, since all the symbols contained in it are now defined).

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

In recent days, I've twice found myself explaining the difference between lambdas and closures in C++11, so I figured it was time to write it up.

The term "lambda" is short for lambda expression, and a lambda is just that: an expression. As such, it exists only in a program's source code. A lambda does not exist at runtime.

The runtime effect of a lambda expression is the generation of an object. Such objects are known as closures.

Given

auto f = [&](int x, int y) { return fudgeFactor \* (x + y); };

the blue expression to the right of the "=" is the lambda expression (i.e., "the lambda"), and the runtime object created by that expression is the closure.

You could be forgiven for thinking that, in this example, f was the closure, but it's not. f is a copy of the closure. The process of copying the closure into f may be optimized into a move (whether it is depends on the types captured by the lambda), but that doesn't change the fact that f itself is not the closure. The actual closure object is a temporary that's typically destroyed at the end of the statement.

The distinction between a lambda and the corresponding closure is precisely equivalent to the distinction between a class and an instance of the class. A class exists only in source code; it doesn't exist at runtime. What exists at runtime are objects of the class type. Closures are to lambdas as objects are to classes. This should not be a surprise, because each lambda expression causes a unique class to be generated (during compilation) and also causes an object of that class type--a closure--to be created (at runtime).

* 1. **Différence entre les différents modes de capture**

An important design decision in creating a language extension for creating locally-defined functions is whether variables

from the enclosing scope can be used in the body of the lambda expression, and if so, how they should be

stored in the closure object. We have chosen to allow such uses, as many uses of lambda expressions (such as in

the multiply list() example above) require this feature. An alternative would be to completely disallow use of local

variables, in which case no support for closures would be necessary, only local classes. We consider the loss in expressiveness

severe enough to not pursue this alternative. Another alternative, suggested by Jeremy Siek, would be

to require all local variables to be stored in the closure to be declared or listed explicitly using some syntax in the

signature of the lambda function. We discuss this approach further in Section 5.3.

Given that local variables can be used transparently within the body of a lambda expression, another question is

how to store them in the closure object. We allow changing the default storage mode from a **const** copy to storing a

reference. Note that the actual implementation need not keep a reference to each particular local variable, if there is

a way to keep a pointer to the stack frame or an equivalent instead.

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;}

**How are Lambda Closures Implemented?**

So how does the magic of variable capture really work? It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function. A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding enviroment are passed into the constructor of the lambda function class and saved as member variables. This is, in fact, quite a bit like the idea of a [functor](https://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html) that is already possible. The benefit of C++11 is that doing this becomes almost trivially easy--so you can use it all the time, rather than only in very rare circumstances where writing a whole new class makes sense.

C++, being very performance sensitive, actually gives you a ton of flexibility about what variables are captured, and how--all controlled via the capture specification, []. You've already seen two cases--with nothing in it, no variables are captured, and with &, variables are captured by [reference](https://www.cprogramming.com/tutorial/references.html). If you make a lambda with an empty capture group, [], rather than creating the class, C++ will create a regular function. Here's the full list of options:

|  |  |
| --- | --- |
| [] | Capture nothing (or, a scorched earth strategy?) |
| [&] | Capture any referenced variable by reference |
| [=] | Capture any referenced variable by making a copy |
| [=, &foo] | Capture any referenced variable by making a copy, but capture variable foo by reference |
| [bar] | Capture bar by making a copy; don't copy anything else |
| [this] | Capture the this pointer of the enclosing class |

Notice the last capture option--you don't need to include it if you're already specifying a default capture (= or &), but the fact that you can capture the this pointer of a function is super-important because it means that you don't need to make a distinction between local variables and fields of a class when writing lambda functions. You can get access to both. The cool thing is that you don't need to explicitly use the this pointer; it's really like you are writing a function inline.

class Foo

{

public:

Foo () : \_x( 3 ) {}

void func ()

{

// a very silly, but illustrative way of printing out the value of \_x

[this] () { cout << \_x; } ();

}

private:

int \_x;

};

int main()

{

Foo f;

f.func();

}

**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. **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. In this approach, lambda expressions are very direct syntactic sugar for defining function objects. We demonstrate with an example, using invented syntax:

* 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).

* 1. **Règles à suivre**

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