WIKIPÉDIA

Fermeture (informatique)

Dans un langage de programmation, une **fermeture** ou **clôture** (en anglais : ***closure***) est une fonction accompagnée de son environnement lexical. 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)[1]. Une fermeture est donc créée, entre autres, lorsqu'une fonction est définie dans le corps d'une autre fonction et utilise des paramètres ou des variables locales de cette dernière.

Une fermeture peut être passée en argument d'une fonction dans l'environnement où elle a été créée (passée *vers le bas*) ou renvoyée comme valeur de retour (passée *vers le haut*). Dans ce cas, le problème posé alors par la fermeture est qu'elle fait référence à des données qui auraient typiquement été allouées sur la pile d'exécution et libérées à la sortie de l'environnement. Hors optimisations par le compilateur, le problème est généralement résolu par une allocation sur le tas de l'environnement.

Sommaire

Exemples

La fonction interne *ajoute10* a toujours accès à l'argument *nombre*, bien que l'appel à la fonction *ajouteur* soit terminé.

ActionScript

En ActionScript :

```
var ajouteur = function(nombre) {
    function ajoute(valeur) {
        return valeur + nombre;
    }
    return ajoute;
}

var ajoute10 = ajouteur(10);
ajoute10(1); // renvoie 11
```

C++

Depuis C++11 :

```
int main() {
    auto ajouteur = [](int i) {
        return ([=](int j) -> int {
            return i + j;
        });
    };
    auto ajoute10 = ajouteur(10);
    ajoute10(1); // renvoie 11
    return 0;
}
```

Avant C++11, le concept de clôture pouvait être implémenté avec des structures :

```
#include <iostream>

struct Ajouteur {
    int n;
    Ajouteur(int val): n(val) {}
    int operator ()(int rhs) {
        return n + rhs;
    }
};

// version utilisant les templates, en supposant que le paramètre 'val' est connu à la compilation
// si ce n'est pas le cas, les templates sont inutiles et l'autre version est la seule possible

template<int n>
struct AjouteurT {
    int operator ()(int rhs) {
        return n + rhs;
    }
};

int main(void) {
    Ajouteur ajoute10(10);
    std::cout << ajoute10(2) << std::endl;
    // si les optimisations sont désactivées, calcule 12 en additionnant deux variables

    AjouteurT<5> ajoute5;
    std::cout << ajoute5(1) << std::endl;
    // si les optimisations sont désactivées, calcule 6 en additionnant une constante (5) avec une variable
```

```
    return 0;
}
```

D'une manière générale, les templates permettent, en plus d'une écriture allégée dans certains cas bien spécifiques, de propager explicitement les constantes, et donc de certifier au compilateur que certains calculs peuvent et doivent être faits dès la compilation sans inquiétude de possibles effets de bord.

Les clôtures peuvent aussi être implémentées à travers des objets de bibliothèques populaires telles que boost. Par exemple `boost::function` couplé de `boost::bind` permet d'implémenter une clôture. Des clôtures plus simples peuvent aussi être implémentées à travers `boost::lambda`.

Le concept de clôture est aussi présent dans la meta-programmation (programmation template), on en trouve beaucoup dans `boost::mpl`. Ceci s'explique du fait que la programmation en langage template se rapproche du paradigme fonctionnel.

Cependant par définition une clôture peut faire référence à son environnement direct, ici la gestion de ce modèle nécessiterait que l'instruction appelante transmette la portée de son propre environnement, par exemple par un passage de référence ou de pointeurs. Cela complique la syntaxe et rend dangereux leur utilisation en fonction de leur durée de vie. Il existe de telles clôtures dans `boost::signals` et libsigc++ qui sont capables de savoir quand la structure appelante est supprimée, évitant alors de potentielles violations d'accès.

C#

En C# :

```
Func<int, int> ajouteur(int nombre) {
    return valeur => nombre + valeur;
}

var ajoute10 = ajouteur(10);
ajoute10(1); // renvoie 11
```

C, C++, Objective-C

Le principe de fermeture a été introduit par Apple au travers des blocs qui sont une extension non standard du C disponible sur OS X à partir de la version 10.6 « Snow Leopard » et sur iOS à partir de la version 4.0[2] :

```
#include <stdio.h>
#include <Block.h>

typedef int (^AjouteBlock) (int);

AjouteBlock ajouteur (int nombre) {
    return Block_copy( ^ int (int valeur) {
        return valeur + nombre;
    });
}

int main(void) {
    AjouteBlock ajoute10 = ajouteur(10);
    printf("%d",ajoute10(1)); // affiche 11

    // Release the block
    Block_release(ajoute10);

    return 0;
}
```

Avec une extension du compilateur gcc, on peut utiliser les fonctions imbriquées pour émuler les closures. Cela fonctionne tant que l'on ne sort pas de la fonction contenante. Le code suivant est donc invalide (la variable nombre n'existe que dans la fonction ajouteur, et sa valeur est perdue après l'appel) :

```c
#include "stdio.h"

void* ajouteur (int nombre)
{
    int ajoute (int valeur) { return valeur + nombre; }
    return &ajoute; // l'operateur & est facultatif car en C le nom d'une fonction est un pointeur dit statique dessus
}


int main(void) {

    int (*ajoute10)(int) = ajouteur(10);
    printf("%d", ajoute10(1));

    return 0;
}
```

Common Lisp

En Common Lisp :

```lisp
(defun ajouteur (nombre)
    (lambda (valeur)
        (+ nombre valeur)))

(defvar +10 (ajouteur 10))
(funcall +10 1) ; retourne 11
```

Delphi

En Delphi :

```delphi
type
    TAjoute = reference to function(valeur: Integer): Integer;

function ajouteur(nombre: Integer): TAjoute;
begin
    Result := function(valeur: Integer): Integer
    begin
        Result := valeur + nombre;
    end;
end;

var
    ajoute10: TAjoute;
begin
    ajoute10 := ajouteur(10);
    ajoute10(1); // renvoie 11
end.
```

Go

En Go :

```
func ajouteur(nombre int) func(int) int {
    return func(valeur int) int {
        return nombre + valeur
    }
}

ajoute10 := ajouteur(10)
ajoute10(1) // renvoie 11
```

Groovy

En Groovy, une fermeture se débute et se termine par une accolade. La fermeture *ajouteur* renvoie une fermeture anonyme :

```
def ajouteur = { nombre ->
    return { valeur -> valeur + nombre }
}
def ajoute10 = ajouteur(10)
assert ajoute10(1) == 11
assert ajoute10 instanceof groovy.lang.Closure
```

Haskell

En Haskell, on peut créer des fermetures à l'aide de fonctions anonymes, aussi appelées fonctions lambda (λ) :

```
ajouteur nombre = \valeur -> valeur + nombre
ajoute10 = ajouteur 10
main = print (ajoute10 1)
```

La curryfication permet de générer des fermetures directement par application partielle des arguments :

```
ajouteur nombre valeur = nombre + valeur
ajoute10 = ajouteur 10
main = print (ajoute10 1)
```

Ou encore, étant donné que les opérateurs sont eux-mêmes des fonctions :

```
ajoute10 = (+ 10) -- Application partielle de la fonction (+)
main = print (ajoute10 1)
```

JavaScript

En JavaScript :

```
function ajouteur(nombre) {
    function ajoute(valeur) {
        return valeur + nombre;
    }

    return ajoute;
}

var ajoute10 = ajouteur(10);
ajoute10(1); // renvoie 11

// ES6
```

```
let ajouteur =  nombre => valeur => valeur + nombre

let ajoute10 = ajouteur(10);
ajoute10(1); // renvoie 11
```

Lua

En Lua :

```
local function ajouteur(nombre)
    return function(valeur)
        return valeur + nombre
    end
end

local ajoute10 = ajouteur(10)
ajoute10(1) -- retourne 11
```

OCaml

En OCaml :

```
let ajouteur n =
  let ajoute v = n + v in
  ajoute;;

let ajoute10 = ajouteur 10;;
ajoute10 1;;
```

Grâce à la curryfication, toute fonction peut générer une fermeture lorsqu'on lui passe seulement une partie de ses arguments :

```
let ajouteur nombre valeur = nombre + valeur;;
let ajoute10 = ajouteur 10;;
ajoute10 1
```

Ou encore, étant donné que les opérateurs sont eux-mêmes des fonctions :

```
let ajoute10 = ( + )  10;;
ajoute10 1
```

On peut également donner d'autres exemples :

```
let creer_predicat_plus_grand_que = function
    seuil -> (fun x ->  x > seuil)
```

qui donne :

```
let sup10 = creer_predicat_plus_grand_que 10;;
sup10 12;; (* true  *)
sup10 8;;  (* false *)
```

OCaml permet également de capturer dans une fermeture une valeur modifiable en place (mutable). Par exemple, pour créer un compteur, on définit simultanément 3 fonctions :

```
let raz, inc, compteur = (* remise à zéro, incrémentation, interrogation *)
    let n = ref 0 in
    (function () -> n:=0),       (* raz = remise à zéro *)
    (function () -> n:= !n + 1), (* inc = incrémentation *)
    (function () -> !n)          (* compteur = interrogation *)
```

la variable mutable n est capturée dans l'environnement commun des 3 fonctions *raz, incr* et *compteur*, qui s'utilisent de la sorte :

```
compteur();;        (* renvoie 0 *)
inc();;             (* incrémente, ne renvoie rien *)
compteur();;        (* renvoie 1 *)
inc();inc();inc();; (* compteur vaut maintenant 4 *)
raz();;
compteur();;        (* renvoie 0 *)
n;;                 (* renvoie "Unbound value n" car n est encapsulée *)
```

Perl

En Perl :

```
sub ajouteur {
    my $valeur = shift;
    return sub { shift() + $valeur };
}

my $ajoute10 = ajouteur(10);
$ajoute10->( 1 ); # retourne 11
```

PHP

En PHP :

```
<?php
function ajouteur($nombre) {
    // on est obligé d'utiliser une fonction anonyme sinon PHP ne déclare pas
    // la fonction dans l'environnement actuel mais dans l'environnement global
    return function($valeur) use($nombre) {
        return $valeur + $nombre;
    };
}

$ajouter10 = ajouteur(10);
$ajouter10(1); // renvoie 11
?>
```

Il est important de noter qu'en PHP, une fonction n'a pas accès aux variables de l'environnement où la fonction est déclarée. Pour ce faire il faut utiliser use($nombre) comme ci-dessus.

Powershell

En PowerShell :

```
function ajouteur($nombre) {
    {
        param($valeur)
        $valeur + $nombre
```

```
    }.GetNewClosure()
}

$ajoute10 = ajouteur 10
&$ajoute10 1 # retourne 11
$ajoute10.GetType().FullName # retourne System.Management.Automation.ScriptBlock
```

Python

En Python :

```
def ajouteur(nombre):
    return lambda valeur: valeur + nombre

ajoute10 = ajouteur(10)
ajoute10(1) # retourne 11
```

En Python 3, le mot-clé `nonlocal` permet de modifier les variables non locales au lieu de les masquer par des variables locales :

```
def f():
    x = 0

    def g():
        nonlocal x
        x += 1
        return x

    return g

g = f()
g() # retourne 1
g() # retourne 2
g() # retourne 3
```

En Python 2, le mot-clé `nonlocal` n'existe pas, on ne peut donc modifier que des variables non locales de types mutables :

```
def f():
    d = {'x': 0}

    def g():
        d['x'] += 1
        return d['x']

    return g

g = f()
g() # retourne 1
g() # retourne 2
g() # retourne 3
```

Ruby

En Ruby :

```
def ajouteur(nombre)
    lambda {|valeur| valeur + nombre}
end

ajoute10 = ajouteur(10)
ajoute10.call(1) # retourne 11
```

Rust

En Rust, les fermetures capturent leur environnement en utilisant pour chaque variable le niveau de privilège le plus bas possible : par référence immuable, par référence mutable, ou enfin par valeur. Chaque fermeture possède un unique type anonyme. On décrit leur type générique à l'aide des traits (interfaces) `Fn`, `FnMut` et `FnOnce` selon que la fermeture prenne la capture par référence immuable, mutable ou par valeur respectivement.

Le mot-clé `move` permet de forcer la capture par valeur, par exemple lorsque la durée de vie de la fermeture risque d'excéder celle des variables. C'est le cas ici puisque `nombre` est alloué sur la pile.

On peut utiliser un type de retour générique (typage et surcharge statique et fermeture allouée sur la pile par défaut) :

```
1 fn ajouteur(nombre: i32) -> impl Fn(i32) -> i32 {
2     move |valeur| valeur + nombre
3 }
4
5 fn main() {
6     let ajoute10 = ajouteur(10);
7     println!("{}", ajoute10(1));
8 }
```

Il est aussi possible de convertir la fermeture en objet-trait (typage dynamique). La taille dynamique impose de placer l'objet derrière un pointeur pour le renvoyer :

```
fn ajouteur(nombre: i32) -> Box<dyn Fn(i32) -> i32> {
    Box::new(move |valeur| valeur + nombre)
}

fn main() {
    let ajoute10 = ajouteur(10);
    println!("{}", ajoute10(1));
}
```

Scala

En Scala :

```
def ajouteur(n: Int)(x: Int) = (x + n)

def ajoute10 = ajouteur(10)
```

Scheme

Le concept de fermeture a été précisé dans l'article de Sussman et Steele de 1975 cité dans l'introduction, qui a présidé à la naissance du langage Scheme. Contrairement à celles de Lisp (qui implémente les fermetures en 1978 dans les machines Lisp du MIT en MacLisp), les fonctions sont en Scheme des valeurs de première classe.

```
(define (ajouteur n)
    ; les variables + et n sont 'libres' dans la lambda ci-dessous, leurs valeurs seront cherchées dans l'environnement de
création de la lambda
    ; ce qui est capturé par la fermeture, ce n'est pas la valeur de n mais la liaison de n à une valeur dans
l'environnement de création
    (lambda (x) (+ x n))) ; la fonction qui à x associe x + n
```

```
> (ajouteur 10)
#<procedure>
> ((ajouteur 10) 1)
11
```

L'existence de fermetures permet d'associer à une fonction un environnement privé de variables, donc un état local et permet de programmer par envois de messages sans disposer d'une couche-objet ad-hoc.

Smalltalk

En Smalltalk :

```
| ajouter ajouter10 |
ajouter := [ :nombre | [ :valeur | nombre + valeur ] ].

ajouter10 := ajouter value: 10.
ajouter10 value: 1. "Retourne 11"
```

Swift

En Swift :

```
func ajouteur(_ nombre: Int) -> (Int) -> Int {
    return { nombre + $0 }
}

let ajoute10 = ajouteur(10)
print(ajoute10(5))
```

CoffeeScript

En CoffeeScript :

```
ajouteur = (n) ->
    (x) -> x + n

ajoute10 = ajouteur 10
console.log ajoute10(5)
```

WLangage

En WLangage, langage de WinDev[3,4]

```
PROCEDURE Ajouteur(LOCAL ajout est un entier) : Procédure
    PROCEDURE INTERNE Ajoute(n)
    RENVOYER n+ajout
    FIN
RENVOYER Ajoute

soit Ajoute10 = Ajouteur(10)
Trace(Ajoute10(1)) // Affiche 11

soit Ajoute100 = Ajouteur(100)
Trace(Ajoute100(1)) // Affiche 101
```

Notes et références

1. Sussman and Steele. « *Scheme: An interpreter for extended lambda calculus* ». « *[…] a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments.* » (Wikisource)
2. `(en)` « *Blocks Programming Topics* » (https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html).
3. Documentation officielle WINDEV (http://doc.pcsoft.fr/fr-FR/?1514075)
4. Blog d'un développeur (http://blog.ytreza.org/windev-ouverture-aux-closures)
