Chapitre 4
Possession et emprunt en Rust

1. Introduction

En Rust, on maîtrise totalement la gestion de la mémoire. Tant du point de vue de l'allocation que de celui de la désallocation. Cette opération s'effectue, bien sûr, sans *garbage collector* (ramasse-miettes) comme c'est le cas en C# ou en Java. On est significativement plus dans la situation du C++, dans laquelle la programmeuse ou le programmeur alloue et désalloue à sa guise.

Remarque

Un ramasse-miettes est un dispositif en code informatique en charge de la désallocation automatique de la mémoire, dès lors qu'elle n'est plus utilisée (ou supposée l'être). La personne en charge du développement n'a plus alors (en théorie) à se poser la question de la libération de la mémoire allouée.

Il existe cependant une différence notable entre C++ et Rust : en C++, une situation problématique n'apparaîtra qu'à l'exécution, que ce soit le fait d'accéder à un espace mémoire déjà désallouée, ou encore que l'on essaie de stocker de la donnée dans un espace non encore alloué. En Rust, ce type de problème est détecté dès la compilation et entraîne un échec de celle-ci. En cas d'erreur, il convient de corriger le code, afin que la compilation réussisse, donnant ainsi la garantie que l'exécution se passera bien, sans tentative d'accès à de l'espace corrompu.

Le langage Rust peut réaliser cela grâce à un principe qui le rend singulier : la **possession** (*ownership* en anglais). Ce principe est absolument central en Rust.

1.1 Le tas et la pile

Évidemment, la gestion de la mémoire implique d'avoir une vision assez claire du fonctionnement du tas et de la pile. Nous avons abordé la plupart des usages respectivement de la pile et du tas dans le chapitre précédent. Pour rappel, voici quelques points à garder à l'esprit :

– Quand on fait un appel de fonctions, les paramètres, c'est-à-dire les valeurs passées à la fonction, sont placés sur la pile. Ce sont des valeurs comme des entiers par exemple. Ce peut être des pointeurs vers quelque chose dans la mémoire. Dans le cas d'une fonction, dès lors que la fonction est terminée, les paramètres présents sur la pile sont retirés de la pile.
– Lorsqu'on alloue des valeurs dans le tas (grâce à *box*), une référence est maintenue sur la pile vers l'emplacement allouée.

1.2 Utilité de la possession

Avant d'expliquer le fonctionnement de la possession en Rust, voyons les quelques points relatifs à son intérêt quant à la gestion de la mémoire :

1. Premier intérêt : la minimisation des données en double (ou triple ou quadruple) dans le tas, et éventuellement dans la pile.

2. Second intérêt : libérer au plus tôt l'espace mémoire alloué dans le tas qui n'est plus utile.

3. L'intérêt principal sans doute : envisager la possession comme le garant d'une gestion optimisée de la mémoire qui, justement, permet de sécuriser l'ensemble du programme.

Sans plus attendre, voyons comment cela fonctionne.

2. Fonctionnement de la possession

2.1 Grands principes

Les principes de la **possession** sont extrêmement simples, c'est leur mise en œuvre concrète qui peut parfois s'avérer compliquée. Voici les deux principes en question :

1. Toute valeur en Rust a un propriétaire et un seul.

2. Quand le propriétaire sort de la portée, la valeur est supprimée.

Cela appelle trois remarques :

– Une valeur peut changer de propriétaire. Auquel cas, le premier propriétaire n'a plus aucun pouvoir sur la valeur après cession.
– Cela s'applique aussi bien à des valeurs stockées sur le tas qu'à celles stockées dans la pile.
– L'unique propriété serait limitative : c'est pour cela que l'on peut emprunter une valeur (**emprunt**, *borrowing*). L'utiliser sans pouvoir la modifier, avant de la rendre.

2.2 Exemples relatifs à la propriété

Dans cet exemple, nous allons tâcher de détailler le fonctionnement de la possession, pour deux valeurs :

– Une valeur entière stockée sur la pile.
– Une valeur relative à une chaîne de caractères, stockée dans le tas.

Remarque

L'exemple proposera différentes séquences dans lesquelles la compilation échoue car le code enfreint justement les grands principes de la possession.

2.2.1 Propriété dans le tas

On définit une chaîne de caractères allouée dans le tas, appartenant à la variable `chaine`. En termes de **possession**, la variable `chaine` possède la valeur `"ENI"` :

```
let chaine = String::from("ENI");
```

Puis il y a un **déplacement** à `chaine2` :

```
let chaine2 = chaine;
```

Si on essaie d'afficher `chaine`, on aura une erreur de compilation :

```
println!("{}", chaine);
```

En effet, on essaie d'accéder à une valeur que le propriétaire a prêtée à une autre variable (`chaine2`). On obtient donc ce message d'erreur :

```
error[E0382]: borrow of moved value: `chaine`
--> src/main.rs:9:20
  |
6 |     let chaine = String::from("ENI");
  |         ------ move occurs because `chaine` has type `String`,
which does not implement the `Copy` trait
7 |     let chaine2 = chaine;
  |                   ------ value moved here
8 |     println!("{}", chaine2);
9 |     println!("{}", chaine);
  |                     ^^^^^^ value borrowed here after move
```

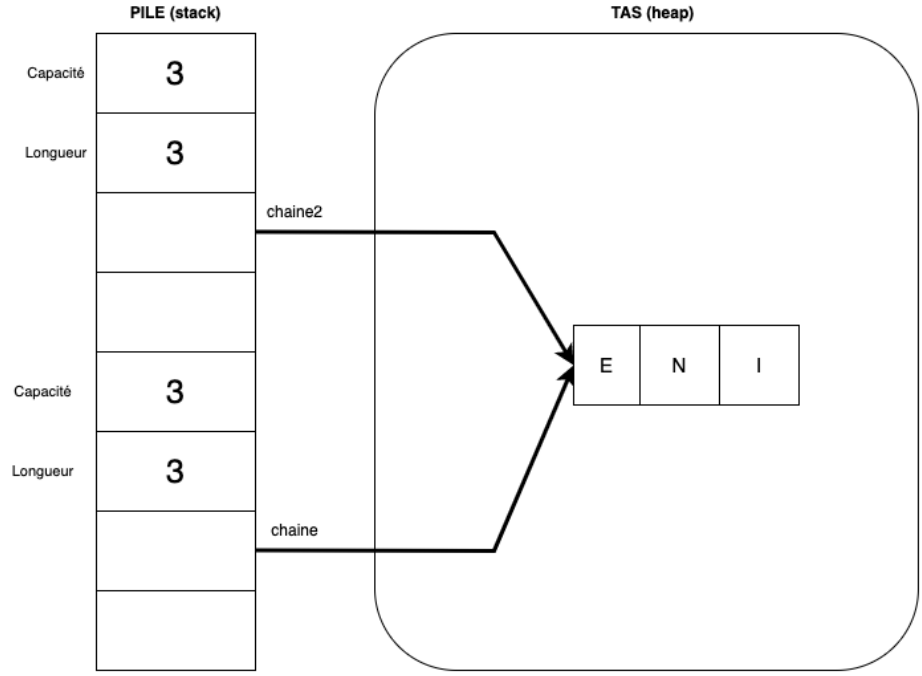
Le petit code associé à l'exemple est pour l'instant le suivant :

```
fn main() {
   possession_chaine()
}

fn possession_chaine() {
   let chaine = String::from("ENI");
   let chaine2 = chaine;

   println!("{}", chaine2);
   //println!("{}", chaine); // provoque une erreur car la valeur
                             // a été prêtée.
}
```

Représentant l'état de la mémoire à ce stade de l'exemple, le schéma ci-dessous montre que la chaîne de caractères n'a évidemment pas été copiée dans le tas. C'est bien une seconde référence depuis la pile vers le tas qui est créée selon une sorte de triplet dans la pile (adresse, longueur de la chaîne, capacité allouée).



L'état de la mémoire à ce stade de l'exemple

Rien n'interdit de faire une copie de la chaîne de caractères dans le tas. Auquel cas, on utiliserait la méthode `clone` disponible sur la classe `String` :

```
let chaine_clone = chaine2.clone();
println!("{:p}", &chaine2);
println!("{:p}", &chaine_clone);
```

Quand on affiche les deux adresses respectives, elles sont différentes. En effet, on a bien une copie dans le tas de la chaîne de caractères `ENI`. Chacun de ses emplacements mémoire dans le tas sont pointés par une référence propriétaire dans la pile, respectivement `chaine2` et `chaine_clone`.

2.2.2 Propriété dans la pile

On définit une variable de type `i64`, c'est-à-dire un entier signé codé sur 64 bits. C'est typiquement un cas où la valeur est stockée sur la pile elle-même. Dans l'exemple, la variable « `annee_hector` » est la propriétaire de la valeur :

```
let annee_hector : i64 = 2011;
```

Puis on effectue une copie. Cependant, ce n'est pas une référence vers la première valeur (pour raisonner de façon analogue à ce que l'on a vu avec le tas). Ici, il y a une copie sur la pile. « `annee_hector_2` » est la propriétaire de la valeur copiée :

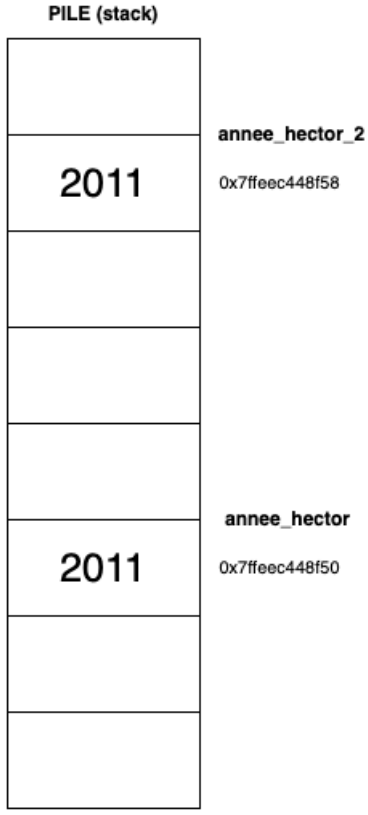
```
let annee_hector_2 : i64 = annee_hector;
```

On affiche pour chacune des variables sa valeur ainsi que son adresse (au sein de la pile :

```
println!("{}", annee_hector);
println!("{:p}", &annee_hector);
println!("{}", annee_hector_2);
println!("{:p}", &annee_hector_2);
```

En sortie, on obtient ceci (on remarque que les deux adresses sont différentes) :

```
> 2011
> 0x7ffeec448f50
> 2011
> 0x7ffeec448f58
```



Les deux valeurs entières dans la pile

À la fin de la portée courante (*scope*), les valeurs seront retirées de la pile car leurs propriétaires n'ont plus d'existence. Nous venons donc de voir au travers de deux exemples très simples les mécanismes de propriété en Rust.