

Ownership, lifetimes et références

Maintenant il faut s'attaquer à quelque chose de très important dans Rust : la manière dont laquelle la mémoire est gérée.

Les types de mémoire

Pour comprendre tout ceci, il faut d'abord que l'on s'attaque aux types de mémoire: le **stack** et le **heap**

stack explained with animation

- Le stack (ou la pile), peut être imaginé comme un pile d'assiettes, on peut mettre des assiettes sur la pile (push) ou en retirer (pop). C'est une mémoire plus rapide que le Heap, car il n'y a pas besoin de calculer un emplacement mémoire et de l'allouer, mais pour l'utiliser il faut connaître d'avance la taille des valeurs à stocker (donc une taille fixe). C'est ici que toutes les valeurs qui ont une taille fixe sont stockées.
- Le heap c'est là que vont tous les types dont la taille est variable et ne peut pas être connue d'avance. Ainsi on trouve un emplacement mémoire suffisamment grand et on y alloue des valeurs. La référence (l'adresse de l'emplacement) peut lui être stocké dans le stack sous forme de pointeur ou de référence (slice).

“ Note : Ce n'est pas parce qu'un type semble avoir une taille variable à première vue qu'il l'est vraiment. Par exemple, un Vecteur est un struct (donc une taille fixe) d'un `RawVec` qui est sur le Heap. On peut savoir si un type est directement sur le Heap si il a le trait `Allocator`

L'ownership

Chaque valeur en Rust a un *propriétaire* et ne peut en avoir qu'un seul à la fois. Et chaque valeur a une durée de vie après laquelle elle n'existe plus.

Par exemple

```
fn main() {  
    let a = String::from("Hello World"); // a est propriétaire du String  
    let b = a; // b devient le nouveau propriétaire de String  
  
    // Par conséquent a n'a plus accès au String et on ne peut plus accéder à a  
    println!("Ceci va générer une erreur : {}", a);  
}
```

Maintenant voyons en plus la durée de vie

```
fn ma_fonction(s: String) {  
    // Fait des choses  
} // A partir d'ici la valeur a n'existe plus (out of scope)  
  
fn main() {  
    let a = String::from("Hello World"); // a est propriétaire du String  
    ma_fonction(a); // ma_fonction devient le nouveau propriétaire  
}
```

En général les accolades {} représentent la durée de vie. Toute variable définie dans ceux ci, ne seront valable que pour cette durée. Donc il en va de même pour les `if`, `loop`, `for`, etc.

Copier la valeur

Maintenant si on peut copier la valeur en elle même on peut utiliser deux traits (on va voir dans un chapitre suivant ce que sont les traits) dépendant du type `Copy` ou `Clone`.

En pratique, tous les types primitifs (qui sont dans le stack) ont le trait `Copy`, ce qui signifie que leur valeurs sont automatiquement copiées.

```
fn ma_fonction(x: i32) {  
    // Faire des choses  
} // A partir d'ici x n'existe plus  
  
fn main() {  
    let a = 5; // a est propriétaire de 5  
    let b = a; // la valeur de a est copiée dans b. b est propriétaire de 5 également mais pas  
    dans le même emplacement mémoire  
    ma_fonction(a); // la valeur de a est copiée dans ma_fonction. ma_fonction est aussi  
    propriétaire de cette copie de valeur
```

```
println!("{}", a, b); // println n'est pas propriétaire de a et b car c'est une macro
qui prends la référence de ceux ci (voir plus tard)
} // a et b n'existe plus à partir d'ici
```

En revanche pour ce qui est des types qui sont dans le Heap, on doit utiliser le trait `Clone` manuellement :

```
fn ma_fonction(s: String) {
    // Faire des choses
} // la copie du string n'existe plus

fn main() {
    let a = String::from("Hello World"); // a est propriétaire du String
    let b = a.clone(); // On copie la valeur de a dans b. B est donc propriétaire de la copie de
a
    ma_fonction(b.clone()); // On copie la valeur de b dans ma_fonction. ma_fonction est
propriétaire de la copie de b

    // Par conséquent les deux valeurs restent accessibles
    println!("{}", a, b);
}
```

Les références (slices)

animation référence

Maintenant quand on ne veut pas copier la valeur mais quand même accéder à la mémoire, on peut utiliser `&` aussi appelé *slice* ou *référence*

```
fn ma_fonction(s: String) {
    // faire des choses
}

fn main() {
    let a = String::from("Hello World");
    let b = &a; // b est de type slice (type primitif stocké dans le stack) et est la référence
vers le String de a
    // n'est donc pas propriétaire du String mais seulement de la référence

    // Mais String et &String ne sont pas les même types, donc une fonction tel que ma_fonction
```

```
qui demande string ne peut pas accueillir &String
    ma_fonction(b);
}
```

Voici à quoi ressemble en mémoire le slice. `s` est une référence (slice) de `s1` qui est un String.



Les slices sont donc des types à part entière qui ont leur propres fonctions et méthodes. Vous pouvez trouver la doc [ici](#)

En savoir plus

- [Le langage de programmation Rust - Qu'est ce que la possession ?](#)
- [Le langage de programmation Rust - Les références et les emprunts](#)
- [Le langage de programmation Rust - Le type slice](#)
- Les animations de ce chapitre viennent de l'article [Rust Visualized: The Stack, the Heap, and Pointers](#) par [ender minyard](#)

Revision #1

Created 27 April 2023 06:10:29 by SnowCode

Updated 27 April 2023 06:12:49 by SnowCode