1_Ownership_fr

May 2, 2023

1 Ownership

Tous les programmes doivent gérer la façon dont ils utilisent la mémoire lors de l'exécution. Certains langages ont un système de ramasse-miettes qui recherche constamment la mémoire non utilisée pendant l'exécution du programme ; dans d'autres langages, le programmeur doit explicitement allouer et libérer la mémoire. Rust utilise une troisième approche : la mémoire est gérée par un système de propriété (ownership) avec un ensemble de règles que le compilateur vérifie au moment de la compilation.

1.1 Règles de ownership (propriété)

- Chaque valeur en Rust a une variable appelée owner (propriétaire).
- Il ne peut y avoir qu'un seul owner à la fois.
- Quand on sort de la portée du owner la valeur est supprimée.

1.2 La portée des variables (Variable scope)

[ ]:
```
{                      // s n'est pas valide ici, elle n'a pas encore été␣
 ↪déclarée
    let s = "hello";   // s est valide à partir d'ici

    // utiliser la variable s
}
```

L'utilisation des chaînes littérales n'est pas toujours la plus adaptée lorsqu'on souhaite manipuler du texte. Une des raisons est qu'elles sont immuables. Une deuxième raison est liée au fait qu'on ne connaît pas toujours leurs valeurs quand on écrit du code : par exemple, lorsqu'on demande à l'utilisateur de les rentrer et qu'on les stocke après. Dans ces cas là, Rust offre la possibilité d'utiliser un autre type : String. Ce type est alloué sur la mémoire de tas et peut ainsi stocker une quantité de texte inconnue au moment de la compilation. On peut créer un String à partir d'une chaîne littérale ainsi :

[2]:
```
let s = String::from("hello"); // ce type est alloué sur la mémoire de tas␣
 ↪(heap)
```

1.3 Mémoire et allocation de la mémoire

En Rust, la mémoire est automatiquement libérée lorsque la variable qui en garde la propriété sort du contexte (n'est plus valide).

[ ]:
```
{
    let s = String::from("hello"); // s est valide à partir d'ici

    // utiliser s

} // fin du contexte,
  // s n'est plus valide
```

Lorsqu'une variable sort du contexte, Rust appele une fonction spéciale : `drop`. Dans cet exemple, en implémentant la fonction drop pour le type String, on peut y introduire du code qu'on souhaite exécuter au moment du nettoyage de la mémoire. Rust appelera automatiquement la fonction `drop` sur le crochet fermant.

> **Note:** En C++, ce mode de désallocation de ressources à la fin de vie d'un item est parfois appelé *Resource Acquisition Is Initialization (RAII)*. Si vous avez déjà utilisé des patrons RAII, la fonction drop de Rust devrait vous être familière.

1.4 Interaction des variables et des données : Move

[ ]:
```
let s1 = String::from("hello");
let s2 = s1;
```

Un String est constitué de trois parties (à gauche de l'image) : un pointeur vers la zone de mémoire qui garde le contenu de la chaîne de caractères, une longeur (length), et une capacité. Ce groupe de données est stocké sur la pile. A droite de l'image est représentée la zone de mémoire de tas qui garde les contenus.

Lorsqu'on attribue s1 à s2, les données du String (les données sur la pile : pointeur, lonegeur, capacité) sont copiées. En revanche, les données de tas référencées par le pointeur ne sont pas copiées.

Imaginons ce qui se passerait si Rust copiait également les données de tas : l'opération s2 = s1 serait très coûteuse au moment de l'exécution si la taille des données était très importante. Lorsque les variables s2 et s1 seraient hors de contexte, elles essayeraient toutes les deux de libérer la même zone de mémoire. Ceci causerait une erreur de sécurité de la mémoire connue sous le nom de « double free error ». Essayer de libérer le même espace de mémoire deux fois peut engendrer une corruption de la mémoire et introduire des vulnérabilités dans le système.

Afin de préserver la sécurité de la mémoire, Rust ne copie pas la mémoire allouée mais tout simplement considère que s1 n'est plus valide. Par conséquent, lorsque s1 est hors de contexte, il n'est plus nécessaire de libérer l'espace alloué par cette variable. Vérifiez ce qui se passe si vous essayez d'utiliser s1 après qu'il a été attribué à s2. Ça ne marchera pas !

Le fait de copier le pointeur, la taille et la capacité sans copier le contenu des données vous rappele peut-être le mécanisme qui, dans d'autres langages, est connu sous le terme de *shallow copy* ou *deep copy*. Mais en réalité, puisque Rust invalide la première variable (qui est copiée), on appele cette opération **move**. Dans cet exemple, on dit que s1 a été déplacée (moved) dans s2.

1.5 Interaction des variables et des données :: Clone

Pour effectuer une copie de l'intégralité des données (pile + tas), on peut utiliser la fonction **clone**.

[ ]:
```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

L'image ci-dessus illustre ce qui se passe lors de l'utilisation de la fonction clone.

Les types de données ayant une taille connue au moment de la compilation, comme par exemple les entiers, sont entièrement stockés sur la pile, donc leur copie est rapide. Pour cette raison là, il n'est pas nécessaire d'invalider une variable après l'avoir copiée, comme montré dans l'exemple suivant :

[ ]:
```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

Rust a une annotation spéciale appelée le trait **Copy**, que l'on peut appliquer aux types stockés sur la pile (comme le type integer). Si le trait Copy est implémenté par un type, une variable plus ancienne peut toujours être utilisée après attribution.

> **trait** = un ensemble de méthodes que l'objet sur lequel il est appliqué doit implémenter

Quelques types *Copy*:

- Tous les types d'entiers, comme par exemple `u32`.
- Le type booléen, `bool`, avec les valeurs `true` et `false`.
- Tous les types floating point, comme le `f64`.
- Le type caractère, `char`.
- Les tuples, uniquement s'ils contiennent des types qui sont également Copy. Par exemple, `(i32, i32)` est Copy, mais pas `(i32, String)`.

1.6 Ownership et fonctions

Pour passer une valeur à une fonction, on utilise la même sémantique que dans le cas de l'attribution d'une valeur à une variable.

[ ]:
```
fn main() {
    let s = String::from("hello");  // s entre en contexte

    takes_ownership(s);             // la valeur de s est déplacée (move) dans␣
 ↪la fonction...
                                    // ... et donc à partir d'ici elle n'est␣
 ↪plus disponible

    let x = 5;                      // x entre en contexte

    makes_copy(x);                  // x serait aussi déplacée dans la fonction,
                                    // mais i32 étant Copy,
                                    // on peut toujours utiliser x par la suite
```

```
} // Ici, x sort du contexte, ensuite s sort du contexte. Comme la valeur de s␣
 ↪a été déplacée (move),
  // aucune action supplémentaire n'est nécessaire.

fn takes_ownership(some_string: String) { // some_string entre en contexte
    println!("{}", some_string);
} // Ici, some_string est hors contexte et `drop` est appélé. La mémoire est␣
 ↪libérée

fn makes_copy(some_integer: i32) { // some_integer entre en contexte
    println!("{}", some_integer);
} // Ici, some_integer est hors contexte. Aucune autre action n'est nécessaire.
main();
```

1.7 Valeurs retournées et leurs portées

Le transfert de propriété (ownership) peut également se faire par l'intermédiaire des valeurs retournées.

[ ]:
```
fn main() {
    let s1 = gives_ownership();         // gives_ownership retourne une valeur␣
 ↪qui est déplacée (move) dans s1

    let s2 = String::from("hello");     // s2 entre en contexte

    let s3 = takes_and_gives_back(s2);  // s2 est déplacée (move) dans␣
 ↪takes_and_gives_back,
                                        // la valeur retournée par␣
 ↪takes_and_gives_back est déplacée dans s3

} // Ici, s3 sort du contexte et est supprimée (dropped). s2 sort du contexte␣
 ↪mais comme elle a été déplacée,
  // rien d'autre n'arrive. s1 sort du contexte et est supprimée.

fn gives_ownership() -> String {         // gives_ownership déplace (move)
                                            // la valeur retournée dans la␣
 ↪fonction d'appel


    let some_string = String::from("hello"); // some_string entre en contexte

    some_string                             // some_string est retournée et
                                            // est déplacée dans la fonction␣
 ↪d'appel

}
```

```rust
// takes_and_gives_back prend un String en paramètre et retourne un autre String
fn takes_and_gives_back(a_string: String) -> String { // a_string entre en␣
 ↪contexte

    a_string  // a_string est retournée et est déplacée dans la fonction d'appel
}
```

2 Exercices

Exercice 1: Corriger le code suivant pour que cela fonctionne, sans toucher à la fonction `into_degrees`.

[ ]:
```rust
#[derive(Debug)]
struct GeoCoordinate {
    latitude: f64,
    longitude: f64
}

impl GeoCoordinate {
    fn into_degrees(self) -> Self {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees(),
        }
    }
}

fn main() {
    let radians = GeoCoordinate { latitude: 1.0, longitude: 1.5 };
    let degrees = radians.into_degrees();
    println!("radians = {:?}, degrees = {:?}", radians, degrees);
}

main();
```

[ ]:
```rust
#[derive(Debug, Copy, Clone)]
struct GeoCoordinate {
    latitude: f64,
    longitude: f64
}

impl GeoCoordinate {
    fn into_degrees(self) -> Self {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees(),
```

```
        }
    }
}

fn main() {
    let radians = GeoCoordinate { latitude: 1.0, longitude: 1.5 };
    let degrees = radians.into_degrees();
    println!("radians = {:?}, degrees = {:?}", radians, degrees);
}

main();
```