3_Variables Mutability_fr

May 2, 2023

1 Variables et mutabilité

2 Variables mutables et non-mutables

Par défaut, les variables sont **non-mutables** en Rust. C'est une des nombreuses caractéristiques de Rust qui poussent à écrire du code qui prend avantage des caractéristiques de sûreté du langage. Il est bien sûr possible cependant de rendre les variables mutables.

L'exemple ci-dessus montre comment le compilateur peut vous aider à trouver des erreurs. Bien que celles-ci puissent être frustrantes, elles indiquent seulement que votre programme ne réalise pas en toute sûreté ce que vous pensiez; elles ne veulent pas dire que vous n'êtes pas un bon programmeur ! Même les Rustacéens expérimentés y sont confrontés. Voici un exmple de code posant problème.

[ ]:
```
fn main() {
    let x = 5;
    println!("La valeur de x est: {}", x);
    x = 6;
    println!("La valeur de x est: {}", x);
}
main();
```

Il est important d'obtenir une erreur à la compilation lorsque l'on essaye de changer la valeur assignée à une variable alors que celle-ci a été déclarée non mutable, car sinon cela conduirait à des bugs. Si une partie de votre code repose sur l'hypothèse qu'une valeur de change pas alors qu'une autre partie effectue un changement, il est possible que la première partie du code ne fasse pas ce pour quoi elle a été écrite. En Rust, le compilateur garantie que lorsque vous spécifiez qu'une valeur ne change pas, elle ne change réellement pas. Cela implique que lorsque vous écrivez et lisez du code Rust, vous n'avez pas à essayer de trouver où une valeur est susceptible d'être modifiée si elle est déclarée non mutable. Le code est ainsi plus facile à lire et comprendre.

Vous pouvez toujours rendre une variable mutable avec l'utilisation du mot clé `mut` devant le nom de la variable. En plus de rendre la variable mutable, `mut` indique aux futurs lecteurs que l'intention originale qu'une partie du code va modifier cette valeur.

[ ]:
```
fn main() {
    let mut x = 5;
    println!("La valeur de x est: {}", x);
    x = 6;
    println!("La valeur de x est: {}", x);
```

```
}
main();
```

Il y a plusieurs compromis à considérer à l'utilisation de la mutabilité, en plus de la prévention des bugs. Par exemple, dans le cas de grandes structures de données, la mutation d'une instance sur place (*in-place*) peut être plus rapide que la copie et l'allocation de nouvelles instances. Pour de petites structures, la création de nouvelles instances et l'écriture de code dans un paradigme fonctionnel peut être plus facile à comprendre. Une éventuelle perte de performance peut être acceptable face au gain en terme de clareté du code.

3 Constantes

Les constantes en Rust ne sont pas juste non-mutables, elles sont toujours non-mutables.

Les constantes sont déclarées avec le mot clé `const` au lieu de `let`. Le type type de la constante *doit toujours* être annoté.

[ ]:
```
const MAX_POINTS: u32 = 100_000;
```

Les constantes peuvent être déclarées dans n'importe quel *scope*, y compris dans le *scope global*, ce qui est utile pour définir des valeurs que plusieurs parties de code doivent connaitre. La valeur assignée à une constante doit elle-même être une expression constante. En Rust, il est possible, dans une certaine mesure, d'utiliser des fonctions constantes dont l'évaluation se fait à la compilation.

4 Shadowing

Vous pouvez déclarer une nouvelle variable avec le même nom qu'une variable précédente. La nouvelle variable remplace la précédente (shadow); ce qui signifie que la précédente disparait et sa valeur n'est plus utilisable. Nous remplaçons une variable en utilisant le même nom et en répétant l'utilisation du mot clé `let`, comme suit:

[ ]:
```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("La valeur de x est: {}", x);
}
main();
```

En utilisant `let`, nous pouvons effectuer un certain nombre de transformation sur une valeur, mais celle-ci restera non-mutable par la suite. La différence avec `mut` est que nous créons réellement une nouvelle variable avec `let`, il est ainsi possible d'utiliser un nouveau type en réutilisant le même nom.

[ ]:
```
let spaces = "   ";
let spaces = spaces.len();
```

5 Exercices

Exercice 1: Corrigez le code suivant:

[ ]:
```
let pair = (5, 6);
let (mut x, y) = pair;
x += 5;
y = x;
println!("Coordinates: ({}, {})", x, y); // attendu (10, 10)

let x = x;
x = x + 5;
println!("x: {}", x); //attendu x: 15
```

solution:

[ ]:
```
let pair = (5, 6);
let (mut x, mut y) = pair;
x += 5;
y = x;
println!("Coordinates: ({}, {})", x, y);

let x = x;
let x = x + 5;
println!("x: {}", x);
```