Cours d'Initiation au Rust

Téo Bernier

Contents

Introduction

Rust est un langage de programmation bas niveau. Bien qu'il offre une multitude de fonctionnalités assez puissantes (Possibilité de `Typage dynamique`, `Itérateurs`, `Collections`, `Async`, `Multi-Threading`, ...), il reste tout de même un langage assez proche de la machine quand il s'agit de la mémoire. Son efficacité vient surtout du fait que les abstractions qu'il propose sont astucieusement gérées à la compilation et non pendant le runtime.

Par exemple, même si Rust propose du typage dynamique, la plupart des codes en Rust utilisera principalement du `typage statique` (potentiellement jusqu'à 10 fois plus rapide). De plus, il assure certaines sécurités statiquement (analyse pendant la compilation) comme les `data races`, les `dangling pointers`, et plus généralement, les `segmentation fault`.

Ressources

- Playlist des Cours
- GitHub

Références

- Rust Book
- Rust By Example
- Easy Rust

Partie I

L'utilisation de Cargo

Cargo est un outil très simple à utiliser, il vous suffira de connaître :

- Gestion de projet :
  - `cargo new project_name`
- Debug static du code :
  - `cargo check`
  - `cargo fmt`
  - `cargo clippy`
- Création d'exécutables :
  - `cargo build`
  - `cargo run`
  - `cargo clean`
- Release d'un exécutable optimisé :
  - `cargo build --release`
  - `cargo run --release`

Les Bases

`println!()`

La première chose que l'on peut voir est évidemment comment faire un `hello world`. On voit ici que la commande utilisée est `println!`.

Le `!` indique ici que ce n'est pas une fonction, mais une macro. Un peu comme `printf` en C.

```rust
fn main() {
    println!("Hello, world !");
    // Output -> "Hello, world !"
}
```

`println!` peut aussi permettre d'afficher des valeurs :

```rust
fn main() {
    println!("{} + {} = {}", 2, 2, 2 + 2);
    // Output -> "2 + 2 = 4"
}
```

On peut aussi afficher des structures plus complexes :

```rust
fn main() {
    println!("{:?}", [1, 2, 3, 4, 5]);
    // Output -> "[1, 2, 3, 4, 5]"
}
```

Le ':' indique qu'il y a des options d'affichage, et le # indique que c'est un affichage de Debug. En effet, les arrays n'ont pas d'affichage classique.

Les variables

Assignation

On peut utiliser des variables en Rust en utilisant le mot-clé `let`.

```rust
fn main() {
    let x = 4;
    println!("{}", x);
    // Output -> "4"
}
```

Le type des variables est souvent déterminé par inférence. Mais on peut tout de même le préciser :

```rust
fn main() {
    let x: u8 = 4;
    println!("{}", x);
    // Output -> "4"
}
```

u8 est un type représentant un entier non-signé sur 8 bits.

Types Primitifs

Il y a ainsi pas mal de types basiques :

- `u8, u16, u32, u64, u128, usize` (usize est un entier de la taille que préfère la machine (u32 si processeur 32 bits, u64 si processeur 64 bits))
- `i8, i16, i32, i64, i128, isize`
- `f32, f64` (floats)
- `char`
- `bool`
- `[type; N]` (Où N est le nombre d'éléments dans le tableau)
- `(type, type, ..., type)` (Un tuple avec possiblement différents types)
- `[type], str` (Les Slices, ils ont une taille inconnue à la compilation (ne sont pas utilisables comme tels))
- `()` (Représent un type vide (un peu comme None en Python))
- Référence / Pointeur et Fonction (On verra ça plus tard)

Immutabilité / Mutabilité

Une variable est de base immutable. C'est à dire qu'il est interdit de la modifier.

```rust
fn main() {
    let x = 4;
    x = 8; // Ne compilera pas
    println!("{}", x);
}
```

Il faut alors utiliser le mot-clé `mut`.

```rust
fn main() {
    let mut x = 4;
    x = 8;
    println!("{}", x);
    // Output -> "8"
}
```

Les fonctions

La définition de fonction n'est pas très compliquée :

En Rust, tout est expression, fonction comprise. (Ainsi, on peut ne pas mettre de ; à la fin d'une ligne. Elle sera alors considérée comme une valeur de retour).

```rust
fn return_2() -> usize {
    2
}

fn main() {
    let x = return_2();
    // C'est comme si return_2 était
    // évaluée comme un expression
    // étant égale à 2.

    println!("{}", x);
    // Output -> "2"
}
```

On peut aussi explicitement écrire :

```rust
fn return_2() -> usize {
    return 2;
}
```

Ainsi, une fonction sera toujours de la forme :

```
fn function_name(param1: Type1, param2: Type2, ...) -> ReturnType {
    ...
}
```

Les structures de contrôle

Comme pour les fonctions, les structures de contrôle sont évalués comme des expressions.

```
// Ce if-else sera alors une expression
// évaluée comme égale à 2
if 3 < 4 {
    1
} else if 4 < 5 {
    2
} else {
    3
}
```

Il y a aussi des boucles :

```
// Ce loop sera alors une expression
// évaluée comme égale à 4
loop {
    break 4;
}
```

Et du pattern matching (On notera qu'on peut aussi matcher une variable : ici _ est comme une variable dont la valeur n'a pas d'importance) :

```
// Ce match sera alors une expression
// évaluée comme égale à "5"
match 5 {
    0 => "0",
    1 => "1",
    2 => "2",
    3 => "3",
    4 => "4",
    5 => "5",
    _ => "Some numbers",
}
```

Il y a aussi des boucles plus habituelles :

```
while cond {
    body
}

// for i in (0..150) revient
// à for i in range(0, 150) en python
// ou à for(i = 0; i < 150; i++) en C

// On peut mettre n'importe quel
// itérateur à la place de (0..150)
for i in (0..150) {
    body
}
```

Les Structures / Enumérations

Définition

En Rust, comme en C, il est possible de définir des `struct`.

```
// On accède aux champs via
// variable.name, variable.age
struct Person {
    name: String,
    age: u8,
}

// On accède aux champs via
// variable.0, variable.1
struct Person(String, u8);
```

Pour les `enum`, c'est un peu pareil, sauf que l'on peut avoir plusieurs structures différentes :

```
enum Color {
    Red,
    Green,
    Blue,
    Black,
    // ...
}
```

```
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}

// On peut aussi avoir des champs nommés

enum ColorEncoding {
    RGBA {
        red: u8,
        green: u8,
        blue: u8,
        alpha: u8,
    },

    RGB {
        red: u8,
        green: u8,
        blue: u8,
    },

    GreyScale {
        grey: u8,
    },
    // ...
}
```

Construction

Pour les `struct` :

```
struct Person {
    name: String,
    age: u8,
}

let john = Person {
    name: "John Doe".to_string(),
    age: 32,
};

// On accède aux champs via
// variable.0, variable.1
struct Person(String, u8);

let john = Person("John Doe".to_string(), 32);
```

Pour les `enum` :

```
enum Color {
    Red,
    Green,
    Blue,
    Black,
    // ...
}

let c = Color::Red;

enum ColorEncoding {
    RGBA {
        red: u8,
        green: u8,
        blue: u8,
        alpha: u8,
    },

    GreyScale {
        grey: u8,
    },
    // ...
}
```

```rust
let c = ColorEncoding::GreyScale {
    grey : 255,
};

// Très inmportant pour après :
use ColorEncoding::*;
let c = GreyScale {
    grey : 255,
};
```

Déconstruction

Il est aussi possible de déconstruire des `struct` avec un `let` ou un `match` :

Pour les `struct` :

```rust
struct Person {
    name: String,
    age: u8,
}

let john = Person {
    name: "John Doe".to_string(),
    age: 32,
};

let Person {
    name: john_name,
    age: john_age
} = john;
```

Ici, on décompose alors la variable `john` pour créer les variables `john_name` et `john_age`. Il faut bien noter qu'il ne sera plus possible d'utiliser `john` après. A moins que les valeurs extraites soient copiables (elles seront donc juste copiées, et `john` existera toujours (C'est une des conséquences du système d'ownership que l'on verra plus en détail plus tard)).

On peut faire pareil avec les pattern matching, et encore même mieux avec les enums :

```
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}

use ColorEncoding::*;
let c = RGB(255,0, 0);

match c {
    RGB(r, g, b) =>
        println!("r: {}, g: {}, b: {}", r, g, b),

    RGBA(r, g, b, a) =>
        println!("r: {}, g: {}, b: {}, a: {}", r, g, b, a),

    GreyScale(g) => println!("Grey : {}", g),
}
```

Methode pour struct / enum

Il est possible, comme dans les langages objets (Rust n'est pas un langage objet), de définir des méthodes pour nos types personnalisés :

```
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Person {
        // Comme les params ont le même nom que les champs,
        // pas besoin d'écrire name: name
        Person {
            name,
            age,
        }
    }
}

let john = Person::new("John Doe".to_string(), 32);
```

Public / Privée / Use

Rust a un système de module assez simple. Ainsi, de base, chaque fichier est considéré comme un module. Ainsi, tous ce qui est dans un module peut-être utilisé sans problème, que ce soit public ou privé. Mais quand on importe des structures, il faut que ces structures soit publiques, et on ne pourra alors appeler que leurs fonctions publiques.

Tout est de base privé en Rust. Il faut alors utiliser le mot-clé `pub` afin de définir que quelque chose est public. Le constructeur de structure (`struct_name(expr1, expr2, ...)`) est d'ailleurs privé, d'où l'utilité de définir des méthodes `new` publiques.

```rust
// fichier person.rs
pub struct Person {
    name: String,
    age: u8,
}

impl Person {
    pub fn new(name: String, age: u8) -> Person {
        Person {
            name,
            age,
        }
    }
}

// autre fichier
use person::Person;
let john = Person::new("John Doe".to_string(), 32);
```

Structures intéressantes de la std

String

Une String est constitué d'un buffer contenant des charactères, et d'une capacité actuelle.

Si l'on rajoute plus de charactères qu'elle n'a de place, elle réallouera un buffer plus grand et recopiera tout son contenu.

```rust
fn main() {
    let mut s = String::new();
    s += "Hello";
    s += ", world!";
    println!("{}", s);

    let s = "Hello, world!".to_string();
    println!("{}", s);
}
```

Je vous conseille vraiment de prendre l'habitude d'aller regarder la doc : String.

Vector

Un Vecteur suit la même idée que les String, sauf qu'à la place, il peut contenir n'importe quel type.

```rust
fn main() {
    let mut v = Vec::new();
    v.push(4);
    v.push(5);
    println!("{:?}", v);
    // Output -> "[4, 5]"
}
```

Si on veut préciser le type d'un Vec :

```rust
fn main() {
    // C'est une macro pour créer
    // les vecteurs plus simplement.
    let mut v: Vec<u64> = vec![4, 5];
    println!("{:?}", v);
    // Output -> "[4, 5]"
}
```

Ici, la doc des Vec.

Premier Concept Important : Ownership

Rust n'utilise pas de Garbage Collector, c'est aussi une des raisons pour lesquelles il est assez efficace ! En effet, c'est le compilateur qui gérera les allocations et les désallocations. En contre-partie, on devra respecter certaines règles afin que le compilateur sache toujours quand allouer et quand désallouer.

Une de ces règles est le principe d'Ownership (possession en Français). Il nous dit que toute variable est possédée par une fonction / scope (`{ }`).

Ainsi, à la fin d'une fonction, toute variable qui n'est pas renvoyée, qui n'est plus utilisée, sera alors désallouée. Aussi, chaque fois qu'on passera une variable en paramètre, si cette variable n'est pas copiable, on ne pourra plus y accéder après.

```rust
fn print_vecteur(v: Vec<u64>) {
    println!("{:?}", v);
}

fn main() {
    let mut v: Vec<u64> = Vec::new();
    v.push(4);
    v.push(5);
    print_vecteur(v);
    // Output -> "[4, 5]"
    // print_vecteur(v);
    // ne compilera pas car v ne nous appartient plus...
}
```

Comme dit au-dessus, certaints type sont copiables, ainsi, ils seront copié avant d'être passé en paramètre, on pourra toujours réutiliser la version originale plus tard.

```rust
fn print_int(n : u64) {
    println!("{}", n);
}

fn main() {
    let n = 8;

    // On copie le n comme variable locale pour print_int.
    print_int(n);
    // Output -> "8"

    // On peut donc le réutiliser ici.
    print_int(n);
    // Output -> "8"
}
```

Macro et Propriété

Les types copiables ont la propriété `Copy` (mais on verra les propriétés plus tard).

Ici, je vais juste vous expliquer comment on demande au compilateur de donner certaines propriétés à nos structures.

Si vous vous rappelez, je parlais de `Debug` pour l'affichage. Cela correspond à la propriété `Debug`.

On a aussi une propriété `Clone`, qui permet de cloner une structure. La grande différence entre Copy et Clone, est que Copie doit toujours être en temps constant (le temps de copie ne doit pas varier selon ce que contient la structure), alors que le clone peut prendre autant de temps qu'il veut.

Ainsi, pour demander au compilateur de donner ces propriétés on utilise :

```
// On ne peut pas demander la copie,
// car copier une string ne se fait pas
// en temps constant, ça dépend de la
// taille de son buffer.

#[derive(Debug, Clone)]
struct Person {
    name: String,
    age: u8,
}

#[derive(Debug, Clone, Copy)]
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}
```