Brève introduction au language Rust

Vincent Espitalier, 26-05-2024

Cette brève introduction au language Rust a pour but de présenter les principales fonctionnalités et particularités de ce language. Une lecture rapide de ce document peut se faire en 30 minutes, et vous donner une première idée des possibilités de ce language. Vous pourrez ensuite approfondir les concepts à l'aide d'autres cours disponibles.

Accés rapide par mots-clefs

let if fn match mut scalaires entiers flottants bool char tuple slice vec struct enum option result

Syntaxe et concepts de base

1) Hello World

Voici le programme standard "Hello, world !" en Rust.

```rust
fn main() {
    println!("Hello, world !") ;
}
```

2) Déclaration des variables

Rust est un langage statiquement typé. Ainsi, lors de la compilation, il doit connaître le type de toutes les variables. Le type n'a pas besoin d'être explicitement s'il peut être déduit du contexte. Voilà quelques-uns des principaux types:

```rust
let i : i32 = 5 ; // type entier
let x : f64 = 3.14 ; // type virgule flottante
let b : bool = true ; // type booléen
let s : &str = "Hello" ; // type slice de chaine de caractères
let j = i ; // Rust déduit implicitement que j est de type i32, comme i.
```

3) Contrôle de flux

Les mots-clés de contrôle de flux de Rust comprennent, de façon assez standard: **if**, **else**, **while**, **loop**, **for** et **match**, cela sera abordé un peu plus tard.

```rust
if x < seuil {
    println!("x est inférieur au seuil") ;
} else if x > seuil {
    println!("x est plus grand que le seuil") ;
} else {
    println!("x est égal au seuil") ;
}
```

4) Fonctions

En Rust, les fonctions sont définies à l'aide du mot-clé **fn**.

```rust
fn saluer() {
    println !("Hello, world !") ;
}
```

5) Structures

Les structures sont utilisées pour créer des types de données complexes en Rust.

```rust
struct Rationnel {
    numerateur   : i32,
    denominateur : i32,
}
let r = Rationnel { numerateur : 4, denominateur : 3 } ; // instancie une structure Rationnel
```

6) Enums

Les Enums en Rust sont des types qui peuvent avoir plusieurs valeurs possibles, appelées 'variant'.

```
enum Direction {
    Haut,
    Bas,
    Gauche,
    Droite,
}
let d = Direction::Haut ; // utilise une variante de l'enum Direction
```

7) 'None' et Gestion des erreurs

Pour gérer les cas particuliers et les erreurs, Rust utilise 2 enums spéciaux **Option** et **Result**.

```
let option : Option<i32> = Some(42) ; // une valeur optionnelle
let result : Result<i32, &str> = Ok(42) ; // un résultat positif
```

Option: Enum qui admet 2 variants possibles: **Some(valeur)** ou **None**. Cela évite par exemple d'utiliser une valeur particulière (par exemple 0, -1) en retour d'une fonction pour signifier par exemple qu'elle n'a pas pu finir le calcul, ou que l'élément recherché n'a pas été trouvé.. **Result** peut prendre les valeurs **Ok(valeur)** et **Err(msg)**.

La gestion des erreurs sera approfondie plus loin.

7) match

switch case en Rust: A droite d'une 'flèche', on trouve un bloc d'instructions délimité par des accolades { }, ou simplement une seule instruction.

```
match d {
    Direction::Haut => println !("Haut"),
    Direction::Bas => { println !("Bas") },
    _ => println !("Ni Haut, ni bas. Peut-être Gauche ou Droite ?"),
}
```

Le pattern-matching est assez élaboré en Rust, comme on le verra par la suite.

8) mut

Variables et mutabilité: Une particularité de Rust est la notion de mutabilité. Les variables sont constantes par défaut en Rust. Pour rendre une variable mutable (i.e. pouvoir la modifier pendant l'exécution), il faut utiliser le mot-clé **mut**.

```
let x = 3.14 ; // variable constante
let mut y = 1.57 ; // variable mutable
y = 1.8 ; // valide
```

Voilà déjà un bref aperçu de la syntaxe et des concepts de Rust. La suite de ce cours approfondit un peu ces concepts.

Variables et types de données

Rust gère nativement 2 types de données:

- Types scalaires : Représentent une valeur unique. Exemples : entiers, nombres flottants, booléens et caractères.
- Types composés : regroupent plusieurs valeurs du même type ou de plusieurs types. Les tableaux (slices) et les tuples en sont des exemples.

Types scalaires

Entiers:

```
let a : i32 = 5 ; // i32 est le type d'un entier signé de 32 bits
```

Voilà tous les types d'entiers en Rust:

Taille	Signé	Non signé
8 bits	i8	u8

Taille	Signé	Non signé
16 bits	i16	u16
32 bits	i32	u32
64 bits	i64	u64
128 bits	i128	u128
archi	isize	usize

isize et usize ont une taille qui dépend de l'architecture de la machine. En général 32 ou 64 bits.

Voir: https://doc.rust-lang.org/book/ch03-02-data-types.html

Flottants:

```
let b : f64 = 3.14 ; // f64 est le type d'un nombre à virgule flottante de 64 bits
```

Rust accepte 2 types de flottants: f32 et f64 (norme IEEE 754-2008).

Plus d'informations:

https://doc.rust-lang.org/std/primitive.f32.html

https://doc.rust-lang.org/std/primitive.f64.html

bool: Type booléen:

```
let c : bool = true ; // bool est le type booléen classique, admettant les 2 valeurs: true, false.
```

Il est possible d'effectuer les opérations booléennes classiques (bit à bit) and, or, not sur booléens et entiers avec les caractères **'&'**, '|' et **'!'** :

```
let a : u8 = 10; // (10 = 2 + 8)
let b : u8 = 6;  // (6 = 2 + 4)
let c = a & b;   // c = 2
let d = a | b;   // d = 2 + 4 + 8 = 14
let e = !a;      // e = 255 - 10 = 245
```

char: En Rust, le type char est codé sur 4 octets et représente une valeur scalaire unicode (Contrairement au C, ou un char représente 256 valeurs, i.e. codé sur 1 octet).

```
let c : char = 'A' ; // char est le type d'un caractère, déclaré avec des guillemets simples.
```

Types composés

Tuples: Un tuple est une collection d'éléments de type potentiellement différents et se note avec des parenthèses.

```
let t : (u64, f32, bool) = (1234, -3.14, true) ;
```

Slice: Une slice est un tableau statique, d'éléments de même type.

```
let f : [i32 ; 5] = [10, 11, 12, 13, 14] ; // Une slice de i32 avec 5 éléments.
println!("{}", f[3]);  // Affiche '13'
```

Vec: Un Vec est un tableau dynamique: On peut ajouter/enlever des élements à la volée.

```
let mut v : Vec<u64> = Vec::new();
v.push(3);
v.push(14);
v.push(15);
println!("{}", v[1]);  // Affiche '14'
```

Voilà pour les types de données standards en Rust. Il est possible de créer des types de données plus complexes, comme on va le voir par la suite.

Types de données avancés

Struct: Les structs, ou structures, permettent de créer des types de données complexes, à partir de données plus simples tels que des scalaires.

Définition d'une structure, et création d'une instance:

```
struct Ligne
{
    x1: u32,
    y1: u32,
    x2: u32,
    y2: u32,
    couleur: String,
    epaisseur: u32
}
let ma_ligne = Ligne {x1: 100, y1: 200, x2: 150, y2:250, couleur: "bleu".to_string(), epaisseur:5} ;
println!("{}", ma_ligne);
```

Enum: Enum est un type de données qui admet différentes plages de valeurs nommées **variant**. Chaque **variant** de l'énumération ne correspond pas nécessairement à une valeur unique, contrairement au langage C.

Définition d'un enum :

```
pub enum TypeFichier {
    FichierRegulier,
    Dossier,
    LienSymbolique,
}
```

Création d'une instance d'une énumération :

```
let type_fichier = TypeFichier::FichierRegulier ;
```

Option: L'enum **Option** est un enum spécial fourni par Rust dans la bibliothèque standard. Sa valeur peut être quelque chose **Some()** ou rien **None**.

```
enum Option<T> {
    Some(T),
    None,
}
```

Elle est utilisée notamment lorsqu'une fonction peut ne pas avoir de valeur de retour.

```
let f : [i32 ; 5] = [10, 11, 12, 13, 14];
println!("{:?}", recherche_index(&f, 12));  // Some(2) - Le 3ème élément du tableau
println!("{:?}", recherche_index(&f, 27));  // None - '27' pas trouvé dans le tableau
```

Comme pour toutes les enums, on peut traiter le résultat avec une instruction match:

```
let result: Option<usize> = recherche_element(ma_liste, val_recherche);
match result {
    Some(index) => println !("L'index de l'élément est {}", index),
    None => println !("Element non trouvé"),
}
```

unwrap: On peut aussi utiliser le mot clef 'unwrap', si l'on est sûr que le résultat n'est pas None:

```
let result: Option<usize> = recherche_element(ma_liste, val_recherche);
println !("L'index de l'élément est {}", result.unwrap());
```

expect: Troisième façon de traiter: Avec 'expect':

```
let result: Option<usize> = recherche_element(ma_liste, val_recherche);
println !("L'index de l'élément est {}", result.expect("Element non trouvé"));
```

Ce qui permet d'afficher un message d'erreur plus précis si le result est 'None'.

On peut aussi utiliser les méthodes **.is_none()** et **.is_some()**.

Result: L'enum **Result** est un autre enum spécial de la bibliothèque standard, principalement utilisé pour la gestion des erreurs. Elle possède deux variants, **Ok** (pour le succès) et **Err** (pour l'erreur).

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Ces types de données en Rust permettent d'écrire des programmes plus robustes et plus flexibles, en gérant de façon explicite les cas particuliers liés aux valeurs absentes et les erreurs.

On peut gérer un Result avec un enum (cf ci-dessous), ou avec unwrap, expect, voire .is_ok() et .is_err().

```
rust resultat: Result<u64, String> = division(4, 2); match resultat {     Ok(result) => println !("Le
résultat est {}", result),     Err(msg) => println !("Erreur : {}", msg),rust
```