1_Data types_fr

May 2, 2023

1 Types de données

Toutes les valeurs en Rust ont un type de données, définissant pour Rust quelle sorte de donnée est spécifiée, de telle sorte que Rust sache l'utiliser. Nous utiliserons deux grands ensembles de types de données : les types **scalaires** et les types **composés**.

Rust est un langage *à typage statique*, ce qui veut dire que Rust doit connaitre le type de toutes les variables à la compilation. Le compilateur peut dans certains cas inférer le type à partir de la façon dont une variable est initialisée et/ou utilisée.

2 Types scalaires

Rust a quatre types scalaires primaires : les entiers, les nombre à virgule flottante, les booléens et les caractères.

2.1 Nombres entiers

Le nom des types pour les entiers signés commencent par **i**, alor que les non-signés commencent par **u**. Le nombre de bits pour le type est explicitement indiqué dans le nom du type, à l'exception des types **isize** et **usize** dont le taille dépend de l'architecture cible (64 bits sur une architecture 64 bits, 32 bits sur une architecture 32 bits).

Si vous n'êtes pas sûr de quel type d'entier utiliser, les choix par défaut de Rust sont en général bons. Le type d'entier par défaut est **i32**, qui est en général le plus rapide, y compris sur les systèmes 64 bits.

Longueur	Signé	Non-signé
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

[ ]:
```
let guess = 5; // i32 par défaut

let y: u32 = 5; // annotation de type pour déclarer le type
```

Une annotation de type est obligatoire lorsque plusieurs types sont possibles et que le compilateur de peut pas choisir.

[ ]:
```
fn main() {
    let guess: u32 = "42".parse().expect("Pas un nombre!");
    println!("Mon nombre est {}", guess);
}
main();
```

Exercice: essayer d'enlever l'annotation de type sur `let guess: u32` dans le code ci-dessus et essayer de le ré-executer.

[ ]:
```
fn main() {
    // Integer addition
    println!("1 + 2 = {}", 1u32 + 2);

    // Integer subtraction
    // TO DO:
    let x = 1i32 - 2; //short syntax: value followed by a type
    println!("1 - 2 = {}", x);

    // Syntax with underscore to improve readability
    println!("One million is written as {}", 1_000_000u32);
}
main();
```

Exercice : Essayez de changer `1i32` en `1u32` pour voir pourquoi le type est important

2.2 Nombres à virgule flottante

Les types de nombre à virgule flottante en Rust sont **f32** et **f64**, avec une taille respecctive de 32 et 64 bits. Le type par défaut est **f64** parce que sur les processeurs modernes il est quasiment aussi rapide que **f32**, mais avec plus de précision.

[ ]:
```
let x = 2.0; // f64

let y: f32 = 3.0; // f32
```

2.3 Opérateurs numériques

[ ]:
```
// addition
let sum = 5 + 10;

// soustraction
let difference = 95.5 - 4.3;

// multiplication
let product = 4 * 30;
```

```rust
// division
let quotient = 56.7 / 32.2;

// modulo
let remainder = 43 % 5;
```

2.4 Opérateurs logiques et binaires

[ ]:
```rust
// Opérateurs logiques
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

// Opérateurs binaires
    println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
    println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
    println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
    println!("1 << 5 is {}", 1u32 << 5);
    println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);
```

2.5 Type booléen

[ ]:
```rust
let t = true;

let f: bool = false; // avec une annotation de type explicite
```

2.6 Le type caractère

Le type le plus primitif pour le texte, **char** définie ses littéraux entre guillemets (quote) simple, par opposition aux littéraux de chaînes de caractères qui utilisent les doubles guillemets. Le type **char** en Rust a une taille de 4 octets et représente une valeur Unicode Scalar Value; ce qui signifie qu'il peut représenter bien plus que les caractères ASCII : lettres accentuées, Chinois, Japonais, Koréen, émojis, et autres espace zero-width sont tous des **char** valides en Rust.

[ ]:
```rust
let c = 'z';
let z = ' ';
let heart_eyed_cat = ' ';
```

2.7 String literals

Une chaîne de caractères est une séquence de n'importe quel caractère Unicode, commençant par et se terminant par des doubles guillemets ” (double quotes), à l'exception de ce caractère qui doit être échappé par un anti-slash (\ ).

> Notez que en interne, les chaînes de caractères Rust (type `String` et `str`) sont toujours représentées en mémoire comme une suite d'encodage UTF-8 valide. Il n'est **PAS**

possible en Rust de constuire une `String` contenant une séquence UTF-8 invalide.

Les caractères de fin de lignes sont autorisés dans les littéraux de chaînes de caractères. Ils sont intégrés tels que dans la chaînes de caractères, sauf dans le cas où ils sont d'abord échappés avec un anti-slash \. Dans ce cas, le caractère d'échappement, la fin de ligne et tous les espaces au début de la ligne suivante sont ignorés.

Ainsi, dans le code suivant, `a` et `b` sont égaux.

[ ]:
```
let a = "foobar";
let b = "foo\
         bar";
```

2.8 Conversions de types

Référence utile pour la converion de types : http://carols10cents.github.io/rust-conversion-reference/

- Utilisez `as` pour convertir des types primitifs.

[ ]:
```
let x = 6.5; // f64 implicitement
let y = x as u32; // conversion de f64 vers u32
println!("{}", y);
let z: String = x.to_string(); // conversion vers une chaîne de caractères
println!("{}", z);
```

2.9 Constantes

[ ]:
```
const N: i32 = 5;
```

A l'inverse d'une déclaration avec `let`, une déclaration avec de constante avec `const` doit utiliser une annotation de type. La librairie standard définie quelques constantes utiles :

[ ]:
```
const MAX: i32 = std::i32::MAX;
const MIN: i32 = std::i32::MIN;
```

[ ]:
```
println!("The i32 max value is: {}", std::i32::MAX);
println!("The u32 max value is: {}", std::u32::MAX);
```

3 Types composés

3.1 Le type tuple

Un tuple est un moyen général pour grouper ensemble des nombre, ou tout autres valeurs avec une variété de types, en un seul type composé. Les tuples ont une *taille fixe*, une fois déclarés, ils ne peuvent pas être aggrandis ou rétrécis.

On écrit un tuple comme une liste de valeurs séparées par des virgules et entourée de parenthèses. Chaque élément du tuple a un type, qui peut être différent d'un élément à l'autre.

[ ]:
```
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

Pour accéder à une valeur individuelle à l'intérieur d'un tuple, il est possible d'utiliser le *pattern matching* pour déstructurer le tuple. Un autre manière consiste à accéder directement à l'élément en utilisant le point (.), suivi de l'index de la valeur voulue dans le tuple.

[ ]:
```
let tup = (500, 6.4, 1); // L'inférence de type peut être utilisée

let (x, y, z) = tup; // déstructuration du tuple

println!("The value of y is: {}", y);
println!("The tuple is: {:?}", tup); // l'option :? sera expliquée plus tard
```

[ ]:
```
let x: (i32, f64, u8) = (500, 6.4, 1);

let five_hundred = x.0; // accès à un élément du tuple

let six_point_four = x.1;

let one = x.2;
```

3.2 Les tableaux

A l'inverse d'un tuple, tous les éléments d'un tableu doivent avoir le même type. Les tableaux en Rust sont différent d'autres langages en cela que la taille de ceux-ci est fixe, à l'instar des tuples. Les tableaux sont utiles lorsque l'on souhaite allouer de la mémoire sur le pile (stack), au lieu du tas (heap).

[ ]:
```
let a: [i32; 5] = [1, 2, 3, 4, 5];
let first = a[0];
let second = a[1];
println!("Premier élément dans le tableau: {}", first);
println!("Second élément dans le tableau: {}", second);
```

Index invalide à l'accès à un tableau

[ ]:
```
let a = [1, 2, 3, 4, 5];
let index = (|| 10)(); // on apprendra la signification de cette syntaxe plus␣
 ↪tard
// TODO: essayer avec `let index = 10;` que constate-t-on ?

let element = a[index];

println!("La valeur de l'élément est: {}", element);
```

La compilation de ce bout de code ne produit pas d'erreur, mais le programme produit une erreur à l'exécution. Lorsque l'on essaye d'accéder à un élément du tableau à un index donné, Rust va

vérifier que l'index est bien dans les bornes du tableau. Lorsque l'index de correspond pas aux bornes du tableau, Rust va lever une erreur (panic).

Ceci est un premier exemple des principes de sûreté de Rust. Dans beaucoup de langages bas niveau ces vérifications ne sont pas faites et il est alors possible d'accéder à une zone invalide de la mémoire. Rust vous protège contre ce type d'erreurs par arrêtant immédiatement l'exécution, au lieu d'autoriser l'accès à une zone invalide de la mémoire.

4 Exercises

Exercise 1: Use the correct data structure.

Sample test data:

Matrix	
1.1	1.2
2.1	2.2

[ ]:
```
fn main() {
    // TO DO (1) Définir une structure qui liste les mois de l'année ou les␣
 ↪jours de la semaine

    // TO DO (2) Définir une constante qui garde la valeur standard epsilon =␣
 ↪1e-11

    // TO DO (3) Définir une matrice de 2 X 2 contenant des nombres réels et␣
 ↪affichez la matrice transposée

}
main();
```

Solution:

[ ]:
```
fn main() {
    // TO DO (1) définir une structure qui liste les mois de l'année ou les␣
 ↪jours de la semaine
    let months: [&str; 12] = ["January", "February", "March", "April", "May",␣
 ↪"June", "July",
                "August", "September", "October", "November", "December"];
    let days = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi",␣
 ↪"Dimanche"];
    // TO DO (2) Définir une structure qui garde la valeur standard epsilon =␣
 ↪1e-11
    const EPSILON: f64 = 1e-11_f64;
    // TO DO (3) définir une matrice de 2 lignes et 2 colonnes contenant des␣
 ↪nombres réels et affichez la matrice transposée
    let matrix: ((f32,f32),(f32,f32)) = ((1.1, 1.2), (2.1, 2.2));
```

```
    println!("{} {}", (matrix.0).0, (matrix.1).0);
    println!("{} {}", (matrix.0).1, (matrix.1).1);

}
main();
```