TELECOM Paris

IP PARIS

Rust express

pour les élèves de 1A

Samuel Tardieu

Juin 2022

Institut Mines-Télécom

TELECOM Paris

IP PARIS

Peut-on vraiment faire un cours express de Rust ?

Oui : bien que puissant et un peu original, Rust reste un langage classique.

- Rust est un langage classique, ressemblant à C ou Java
- Rust est un langage puissant
- Rust est un langage original

Nous allons insister sur les différences. Certains concepts sont volontairement simplifiés.

Posez un maximum de questions sur chaque page pour être sûr d'avoir bien tout compris.

Exemple de fonction

Cette fonction ajoute 10 à la valeur reçue en paramètre :

```
1 fn add_10(x: u32) -> u32 {
2   x + 10
3 }
```

- Toute expression retourne une valeur, nul besoin d'un `return` (il ne sert que pour sortir de manière prématurée).
- Les types entiers non signés sont `u8`, `u16`, `u32`, `u64` et `u128`, ainsi que `usize` (taille d'un pointeur).
- On dispose des mêmes types en version signée (`i32`, `isize`, etc.).
- Les nombres à virgule flottante utilisent `f32` et `f64`.

Notre propre type

```
1  pub struct MyType {
2    pub some_field: i32,
3    n: u64,
4  }
5
6  fn foobar() {
7    let s: MyType = MyType { some_field: -5, n: 42 };
8    let t = MyType { n: 12, some_field: 1 };
9    …
10 }
```

- `pub` permet de référencer ce type en dehors de la hiérarchie du module courant ou d'avoir un accès direct au champ (ici `some_field`).
- `let` permet de créer une nouvelle variable locale, le type est facultatif quand il peut être déduit.

Les pointeurs

Entrons dans le vif du sujet avec les pointeurs :

- En Rust, chaque objet a un unique propriétaire à un moment donné (par exemple une variable). Le propriétaire stocke l'objet et sera chargé de le détruire quand il l'abandonnera.
- Donner un objet à une fonction lui en transmet la propriété exclusive.
- Quand on veut garder la propriété d'un objet, on en passe une référence à un tiers plutôt que l'objet lui-même.
- Une référence est comme un pointeur mais est assorti d'une durée de vie qui ne peut pas dépasser celle de l'objet.

```
1 fn main() {
2   let s = MyType { n: 12, some_field: 1 };
3   foo(&s);   // &s is a immutable reference to s (cannot modify s)
4   bar(s);  // s no longer exists after this call
5 }
```

Immutable and mutable

Par défaut, les variables créées avec `let` sont immuables et l'objet ne peut pas être modifié. Le mot clé `mut` permet de créer des variables et références mutables :

```
1  fn main() {
2    let mut s = MyType { n: 12, some_field: 1 };
3    s.some_field = 17;
4    foo(&mut s);    // &mut s is a mutable reference to s (can modify s)
5    bar(&mut s.n);  // &mut s.n is a mutable reference to the n field of s
6  }
7
8  fn foo(v: &mut MyType) {
9    v.some_field = 30;
10 }
```

Un objet ne peut pas être modifié si quelqu'un possède une référence sur lui. Une référence mutable ne peut exister que s'il n'existe aucune autre référence sur le même objet.

Dit autrement, personne ne peut voir un objet changer *sous ses pieds*.

Durée de vie

On ne peut pas stocker une référence plus longtemps que la durée de vie associée :

```
fn main() {
  let a = 42;
  let mut r: &i32 = &a;  // r is mut so it can be updated
  {
    let b = 10;
    r = &b; // ERROR: b doesn't live long enough for r, won't compile
  } // b lifetime ends here (as well as &b lifetime)
  println!("r contains {}", *r);  // r must live until there
}
```

En C, cela aurait fonctionné et aurait affiché n'importe quoi :

```
int main() {
  int a = 42;
  int *r = &a;
  { int b = 10; r = &b; }
  printf("r contains %d\n", *r); // Incorrect
}
```

Nommer une durée de vie

Cette fonction renvoie une référence sur le champ `some_field` de son paramètre :

```
fn access_to_field<'a>(s: &'a MyType) -> &'a i32 {
  &s.some_field
}
```

- La fonction prend une durée de vie (`'a`) en paramètre générique entre <>.
- Les références ont une durée de vie `'a`. L'accès au champ `some_field` ne peut pas vivre plus longtemps que l'accès à la structure `s` elle-même : aucun risque de pointer sur une zone désallouée.

Avec une seule référence en entrée, on aurait pu omettre la durée de vie, celle-ci aurait été implicitement copiée du paramètre :

```
fn access_to_field(s: &MyType) -> &i32 {
  &s.some_field
}
```

Autre exemple de vérification des durées de vie

Ces deux versions ne compilent pas :

```
fn add_10_v1(a: i32) -> &i32 {  // ERROR: unable to guess lifetime for return value
  let b = a + 10;
  &b  // ERROR: b does not live long enough
}

fn add_10_v2(a: &i32) -> &i32 {  // equiv. to fn add_10_v2<'t>(a: &'t i32) -> &'t i32
  let b = *a + 10;
  &b  // ERROR: b does not live long enough
}
```

`'static` indique une durée de vie qui est valable jusqu'à la fin du programme :

```
fn ref_to_10() -> &'static i32 {
  &10
}
```

Fonctions associées et méthodes

```rust
pub struct MyType {
  pub some_field: i32,
  n: u64,
}

impl MyType {
  fn new(sf: i32, n: u64) -> Self {  // Self is MyType in this impl block
    MyType { some_field: sf, n }
  }

  fn increase_n(&mut self, by: u64) {
    self.n += by;   // Equiv. to (*self).n += by
  }
}

fn main() {
  let mut s = MyType::new(42, 10); // :: for associated function call
  s.increase_n(4);  // . for method call
  println!("s = MyType {{} some_field: {}, n: {} }}", s.some_field, s.n);
}
```

Et donc il y a des classes ? Et de l'héritage ?

Non ! En Rust, on utilise des `Trait` (de personnalité) et on indique qui les implémente et comment :

```rust
pub trait CanBeLarge {
  fn is_large(&self) -> bool;  // bool is false or true
}

impl CanBeLarge for u32 {
  fn is_large(&self) -> bool { *self > 10_000 }
}

impl CanBeLarge for u8 {
  fn is_large(&self) -> bool { false }
}

fn check_if_large(v: &dyn CanBeLarge) {  // Ref. to any type implementing CanBeLarge
  // Note that if argument doesn't require parentheses
  if v.is_large() { println!("It is large;") } else { println!("It is small"); }
}
```

Quelques traits prédéfinis

- `Display` : type ayant un affichage canonique (chaînes de caractères, nombres, etc.), utilisable avec `{}` dans les macros de formattage comme `println!()`.
- `Debug` : idem mais pour un affichage non-canonique utilisé pour le développement (vecteurs, tableaux, structures, etc.).
- `PartialEq`, `Eq`, `PartialOrd`, `Ord` : égalités partielle et totale, ordres partiel et total.
- `Default` : type avec une valeur par défaut.
- `Clone`, `Copy` : type clonable avec `.clone()` ou implicitement copiable (`i32`).

```rust
#[derive(Default)]  // Automatic derivation
pub struct MyType {
  pub some_field: i32,
  n: u64,
}
```

vs.

```rust
impl Default for MyType {   // Explicit trait implementation
  fn default() -> Self { MyType { some_field: 0, n: 0 }}
}
```

Pourquoi ne pas avoir parlé des chaînes de caractères ?

C'est un sujet délicat car bien pensé :

- Un caractère (`char`) occupe 32 bits et peut contenir n'importe quelle valeur Unicode (langues terrestres, smileys, etc.).
- Le codage `UTF-8` permet de représenter ces symboles avec une taille variable, de 1 octet (alphabet latin, chiffres, symboles de ponctuation) à 4 octets (symboles rares).
- Les chaînes de caractères en Rust sont des successions d'octets (`u8`) représentant nécessairement une succession de séquences UTF-8 valides.
- Le type `String` contient une chaîne de caractères et la stocke en mémoire, puis la libère quand l'instance est détruite.
- Le type `&str` représente une référence sur une suite d'octets interprétée comme une chaîne de caractères mais ne possède pas la chaîne elle-même.
- `String` se déréférence automatiquement en `&str` quand le contexte le demande.

Et en pratique, ça donne quoi ?

- On peut utiliser `String::from()` ou `.to_owned()` ou `.to_string()` pour créer une nouvelle `String` (avec sa propre zone mémoire).
- Les chaînes littérales (présentes dans le code source) sont des `&str` avec une durée de vie statique (valable jusqu'à la fin du programme).

```
1  fn main() {
2    let str1: &'static str = "this is a preexisting string";
3    let str2: String = String::from(str1);  // Makes a copy
4    let str3: &str = &str2;  // Takes a reference to str2
5    println!("String is {}", str3); // &str implements Display
6    let str4: String = str1.to_owned();  // Makes a copy
7    let str5: String = str1.to_string(); // Makes a copy
8  }
```

Mais pourquoi c'est mieux que C ou Java ?

- En Rust, une `&str` est exactement comme une suite d'octets en mémoire (`&[u8]`, une *slice* (tranche) de `u8`, *i.e.* une adresse de début et une longueur) avec la garantie que cette suite d'octets représente une chaîne UTF-8 valide. On ne peut pas agrandir ou diminuer une `&str`. On n'a jamais à se soucier de la désallocation d'un `&str` car le `&str` n'est pas propriétaire de la chaîne sous-jacente.
- En Rust, une `String` gère une zone mémoire qu'elle a allouée et qu'elle peut réallouer si la chaîne grandit (elle doit être mutable pour être modifiée) ou la désallouer si la chaîne est détruite. On peut à tout moment obtenir un `&str` depuis une `String` car cette zone mémoire contient nécessairement une séquence UTF-8 valide.
- En Java, une `String` est toujours modifiable. Quand on la passe à une fonction (par référence), celle-ci peut la modifier dans notre dos.
- En C, un `char *` n'indique pas clairement qui est propriétaire de la chaîne et qui doit la désallouer.

Autre exemple de version propriétaire/non-propriétaire

- `Vec<T>` représente un vecteur d'objets de type `T` (qu'il possède) qui peut croître, diminuer, etc.
- `[T; N]` représente un tableau de `N` objets de type `T` (qu'il possède). Il ne peut pas changer de taille.
- `&[T]` représente une *slice* (tranche) d'objets de type `T`, conceptuellement une adresse de début et une longueur. On peut obtenir un `&[T]` depuis un `Vec<T>` ou un `[T; N]`.

```
1  fn max(data: &[u32]) -> u32 {
2    let mut max = 0;
3    for d in data {
4      if *d > max { max = *d; }
5    }
6    max
7  }
8
9  fn main() {
10   let v: Vec<u32> = vec![10, 20, 30];
11   println!("max of {:?} is {}", v, max(&v)); // Makes slice
12 }
```

Types énumérés

Les énumérations en Rust sont plus puissantes que les `enum` et les `union` en C :

- Les variantes peuvent contenir des valeurs (pas comme les `enum`).
- Lorsqu'on a une variable contenant une énumération on peut savoir quelle est la variante utilisée (pas comme les `union`).

Le type prédéfini `Option<T>` représente soit une valeur de type `T` (`Some(T)`) soit l'absence de valeur (`None`).

```
1  pub enum Option<T> {
2    None,
3    Some(T),
4  }
5
6  fn check(opt: Option<i32>) {
7    match opt {   // Use pattern matching to identify the variant
8      None => println!("Doesn't contain anything"),
9      Some(x) => println!("Contains {x}"),
10   }
11 }
```

Exemple d'utilisation de Option

Attardons nous sur `Option<T>` :

```
1  fn maybe_add_one(opt: Option<i32>) -> Option<i32> {
2    match opt {
3      None => None,
4      Some(x) => Some(x+1),
5    }
6  }
7
8  fn maybe_add_one_v2(opt: Option<i32>) -> Option<i32> {
9    opt.map(|x| x+1)  // |x| x+1 is an anonymous function (lambda)
10 }
11
12 fn display_or_panic(opt: Option<i32>) {
13   println!("opt contains {}", opt.unwrap());  // Panic if None
14 }
```

`panic!()` est un moyen de crasher le programme courant et est appelé par `unwrap()` si on ne trouve pas un `Some()`.

Toujours Option

- `Option<T>` implémente `Default` (renvoie `None`).
- Si `T` implémente `Debug`, `PartialEq` `PartialOrd`, `Eq`, `Ord`, `Option<T>` l'implémente également.

Le *pattern-matching* n'est pas limité à la construction *match*, elle peut aussi être utilisé dans un `if let` par exemple :

```
1  fn display<T: Display>(opt: Option<T>) {  // T must implement Display
2    if let Some(data) = opt {
3      // data exists only in this branch
4      println!("The content is {data}");
5    } else {
6      println!("Nothing to see here");
7    }
8  }
```

Et tout ça on le met où ?

Un programme Rust :

- Est organisé en modules.
- Un module est déclaré avec le mot clé `mod` et son contenu est soit donné dans le fichier source où il est déclaré, soit dans un fichier avec le nom du module.

```
1  pub mod my_module {
2    pub struct S { a: i32 }
3  }
4
5  pub mod my_other_module;  // In my_other_module.rs, or my_other_module/mod.rs
```

L'éditeur Visual Studio Code dispose d'un excellent support pour Rust avec son extension `Rust analyzer`.

Et les dépendances ?

- Un programme Rust dépend de bibliothèques appelées *crates* (coffres).
- Le programme `cargo` est le couteau suisse de Rust pour récupérer les dépendances, compiler, générer la documentation, lancer le programme, etc.
- Le fichier de configuration `Cargo.toml` permet de configurer `cargo`.
- Des outils additionnels peuvent enrichir `cargo` en ajoutant des sous-commandes.
- `rustup` est l'utilitaire d'installation de Rust qui permet de maintenir son installation de Rust à jour ainsi que `cargo`.

Que faut-il connaître d'autre ?

Plein de choses, dont :

- le type `Result<T, E>`,
- la bibliothèque standard,
- le parallélisme,
- la pratique, la pratique, la pratique.

```rust
pub enum Result<T, E> {
  Ok(T),
  Error(E),
}
```

Pour continuer

- La documentation de Rust.
- Le site crates.io pour chercher des crates.
- Le site doc.rs pour la documentation des crates.
- Le site rustup.rs pour l'installation de Rust.
- La vidéo space invaders, excellent tutorial.