Carte d'identité de Rust

- Origine : Projet personnel de Graydon Hoare @ Mozilla
- Site officiel : https://www.rust-lang.org/
- Créneau : Abstractions haute performance
  - Faire un maximum de travail à la compilation
  - Donner pleinement accès aux fonctionnalités du matériel
  - Éviter tout coût caché, imprévisible ou inévitable (ex : GC)
  - Programmation structurée, fonctionnelle, et générique
- But : Remplacer C++ par une alternative plus ergonomique, mieux conçue, et au moins aussi efficace et productive
- Un des moyens : **Prouver l'absence d'erreurs mémoire**

La propriété (ownership)

- Cousin de la RAII (C++) et des context managers (Python)
- Idée : Chaque objet est “possédé” par une région du code. Quand ce code se termine, les ressources sont libérées.
- Comment peut-on partager une ressource ?
  - En déplaçant l’objet associé vers un autre scope
  - En “prêtant” des références vers ledit objet
- Et les copies dans tout ça ?
  - Pas toujours possible ou pertinent (ex : copier un mutex ?)
  - Souvent coûteux d’un point de vue de performances
  - Donc explicite sauf rares exceptions : `x.clone()`

Déplacement

```rust
// Fonction qui prend un vecteur en entrée, par valeur
fn process_vec(v: Vec<i32>) {
  // Ici, la fonction est propriétaire de l'argument "v"
}

fn main() {
  let v = vec![1, 2, 3];  // Création d'un vecteur v
  process_vec(v);  // Transmission à process_vec

  // Utiliser "v" ici serait une erreur de compilation
}
```

Limites du déplacement

- Tout faire avec des déplacements ? Mauvaise idée !
- C'est souvent malpratique, et parfois inefficace

```
fn modify(mut s: BigStruct) -> BigStruct {
  s.some_member += 1;
  return s;
}

fn main() {
  // ...initialisation de my_struct...
  my_struct = modify(my_struct);
}
```

L’emprunt (borrowing)

- On a souvent envie de “prêter” des références vers un objet
- C’est très pratique… mais potentiellement très dangereux
  - Code moins clair (ex : pointeurs C, objets Java/Python…)
  - Problèmes de durée de vie (ex : “dangling reference”)
  - Problèmes de programmation concurrente (ex : “data races”)
- L’approche Rust : on l’autorise… sous certaines conditions
  - Une référence ne peut pas sortir du scope de l’objet parent
  - Un seul morceau de code à la fois peut modifier l’objet
  - On ne peut lire un objet que si personne ne peut y écrire

Quelques emprunts valides

```
fn modify(s: &mut BigStruct) {
  s.some_member += 1;
}

fn readout(s: &BigStruct) -> i32 {
  s.some_member
}

fn main() {
  // ...initialisation de my_struct...
  modify(&mut my_struct);
  println!("New value is: {}", readout(&my_struct));
}
```

Quelques emprunts invalides

```
let r: &i32;
{
  let x = 42;
  r = &x;
}
println!("Valeur pointée par r: {}", *r);

let mut x = 42;
let t1 = thread::spawn(|| { x += 1; });
let t2 = thread::spawn(|| { x += 1; });

let mut x = 42;
let t = thread::spawn(|| { x += 1; });
println!("Valeur de x: {}", x);
```

Et plus encore

- Tout ce que je viens de montrer est vérifié à la compilation
  - Aucun coût et aucun crash à l'exécution
  - Des variantes "dynamiques" sont disponibles si besoin
- Quelques autres bonnes idées de gestion mémoire :
  - Une variable doit être initialisée avant utilisation
  - Les structures contenant des références (ex : itérateurs) sont traitées avec l'attention particulière qu'elles méritent
  - Toute donnée est immutable ("const") par défaut
  - L'utilisation de variables globales est fortement restreinte
  - Thread qui crashe en tenant un mutex ≠ deadlock