TP1 - Plongée dans le monde de Rust

Qualité de Développement - R4.02

Hello Rust world!

Si vous travaillez sur un poste de l'IUT, Rust est déjà installé.

Si vous souhaitez travailler sur un ordi personnel, utilisez la méthode d'installation utilisant `rustup` pour installer tous les outils utiles au développement Rust.

Hands on "Hello World"

Pour ce premier contact avec le langage Rust, lancez les commandes suivantes pour créer un nouveau paquet Rust (*Rust package*) et vous placer dedans :

```
cargo new hello_rust
cd hello_rust
```

Explorer le répertoire du paquet avec votre IDE ou avec :

```
tree
```

Une fois que vous avez observé la hiérarchie de fichiers et répertoires du paquet, lancez votre programme avec :

```
cargo run
```

Votre terminal devrait saluer le monde.

Plusieurs choses sont à noter :

- Comme vu en cours, `cargo` est l'outil à tout faire en Rust.
- Un projet Rust suit des conventions :
  - les fichiers sources du projet sont dans un répertoire `src`,
  - le code source du programme est dans le fichier `main.rs`,
  - les produits de la compilation sont dans un répertoire `target`,

    - l'exécutable porte le nom du projet, `hello_rust` ici.
    - Les dépendances du projets sont stockées dans le fichier `Cargo.toml`
- Si vous faites un `git status`, vous constaterez que `cargo` crée un dépôt git par défaut pour votre projet.

Éditeur

Pour éditer votre code, vous avez 3 options sur les postes IUT:

- Utiliser votre éditeur de texte favori pour modifier le fichier `main.rs` qui se trouve dans le répertoire `src` de votre projet et les commandes fournies par `cargo` pour la compilation et l'exécution.
- Utiliser RustRover, l'IDE fournie par JetBrains pour le Rust.
- Utiliser VSCode avec l'extension "rust-analyzer" qui fournit l'intégration de Rust.

Dans tous les cas, vous pouvez continuer à utiliser les commandes fournies par `cargo` pour la compilation, l'exécution, les tests ou le formatage du code (`cargo fmt`) depuis le terminal.

Première variable

Ajoutez une variable `i` de type entier puis affichez là :

```
fn main() {
    let i : i32 = 12;
    println!("Ma variable : {i}");
}
```

Ici `i32` est le type de la variable `i` un entier sur 32 bits. C'est le type par défaut pour les valeurs entières.

Lancez le programme

- via votre IDE si vous en utilisez un ou
- en utilisant `cargo`

```
cargo run
```

À noter :

- Le mot clef `let` introduit une nouvelle variable.
- Rust est explicite sur la taille en mémoire des types de base.
- On a les types de base : `i8, i16, i32, i64, bool, f32, f64, char, u8, u16, u32, u64`

Première bonne surprise : déduction de type

Supprimez le `: i32` de la déclaration de la variable : tout fonctionne.

Dans 99% des cas, le compilateur sait déduire le type de vos variables.

Si vous utilisez un IDE digne de ce nom, il devrait vous indiquer le type que le compilateur à déduis à côté de la variable.

```
let i: i32 = 12;
```

Première variable non constante, "*mutable*"

Que dit le compilateur si vous lui donnez le code suivant ?

```
fn main() {
    let i = 12;
    i = 42;
    println!("Ma variable : {}", i);
}
```

Les variables sont **constantes** *par défaut en Rust*

Pour que le code compile, il faut ajouter le mot clef "`mut`" avant le nom de la variable :

```
fn main() {
    let mut i = 12;
    i = 42;
    println!("Ma variable : {}", i);
}
```

Première fonction

Ajoutez une fonction à votre programme pour afficher une salutation au monde plus personnelle :

```
fn main() {
    let repetition = 5;
    greetings(repetition);
}

fn greetings(rep: usize) {
    let mut greet = String::from("Hello");
    greet.push_str("o".repeat(rep).as_str());
    greet.push_str(" world!");
    println!("{}", greet);
}
```

À noter :

- Il est possible d'utiliser une fonction avant de l'avoir déclarée.
- Tous les types ont des méthodes associées : ici on utilise la méthode `repeat` de la chaîne `"o"`.

- Examinez le type de la variable `repetition` dans la fonction `main`. Le compilateur a déduit qu'elle était de type `usize` car la fonction `greetings` prend un paramètre de ce type.

Première fonction qui retourne un résultat

Modifiez votre programme pour que la fonction `greetings` ne réalise plus l'affichage mais retourne la chaîne désirée :

```
fn main() {
    let repetition = 5;
    println!("{}", greetings(repetition));
}

fn greetings(rep: usize) -> String {
    let mut greet = String::from("Hello");
    greet.push_str("o".repeat(rep).as_str());
    greet.push_str(" world!");
    greet
}
```

À noter :

- La syntaxe pour préciser le type de retour d'une fonction est `->`
- La dernière expression d'une fonction (ou d'un bloc de code) est le résultat de la fonction s'il n'y a **pas de point-virgule**. Ici le dernier `greet` indique que la fonction retourne cette variable comme résultat.

Première documentation

Ajoutez quelques informations sur votre code :

```
/// My first Rust program that display a personalized greeting
fn main() {
    let repetition = 5;
    println!("{}", greetings(repetition));
}

/// My first function, returns a friendly greeting
/// # Arguments
/// * `rep` - The number of 'o' to add to the greeting
///
fn greetings(rep: usize) -> String {
    let mut greet = String::from("Hello");
    greet.push_str("o".repeat(rep).as_str());
    greet.push_str(" world!");
    greet
}
```

Lancez ensuite la commande :

```
cargo doc --open
```

Cela lance la génération de la documentation de votre projet et ouvre cette toute nouvelle documentation dans un navigateur. Vous pouvez aussi ouvrir le fichier `index.html` qui se trouve dans le répertoire `target/doc/hello_rust/`

Premier test

Afin de valider que votre fonction donne bien le résultat attendu, ajoutez un test à votre programme, dans le `main.rs`.

```
#[cfg(test)]
mod tests {
    use crate::greetings;

    #[test]
    fn greetings_works() {
        let g = greetings(5);
        assert_eq!(g, "Hellooooo world!");
    }
}
```

Vous pouvez tester la fonction avec l'IDE qui devrait avoir ajouter un moyen d'exécuter les tests au niveau de la ligne comportant `'mod tests` ou alors en ligne de commande :

```
cargo test
```

Ajoutez un 'o' à la première ligne de la fonction `greetings` puis vérifiez que les tests ne passent plus.

Ajoutez un nouveau test qui vérifie que la fonction fonctionne toujours même avec un argument à 0.

Ajoutez un test pour un nombre de répétition négatif. Que constatez-vous ?

Premier tableau

Il existe deux principaux types de tableau en Rust : statiques et dynamiques.

Les tableaux statiques, ou (`fixed-size`) `array`, consistent en une liste d'éléments du même type stockés de manière contiguë en mémoire.

Créez un tableau d'entier puis affichez-le.

Création d'un tableau de 100 entiers avec initialisation :

```
let a = [42;100];
```

Une première version pour l'affichage du tableau consiste à afficher chaque élément dans une boucle :

```
for e in a {
    print!("{e},");
}
```

On peut également utiliser la version fonctionnelle :

```
a.iter().for_each(|e| print!("{e},"));
```

Que se passe-t-il si vous essayez une version plus intuitive comme suit ?

```
println!("a : {}", a);
```

Lisez la suggestion faites par le compilateur et faites afficher le tableau.

Premier vecteur

Les tableaux statiques que vous venez de voir sont alloués sur la pile. Il n'est pas judicieux de les utiliser pour des grandes quantités de données.

Pour des tableaux de plus grande taille et également pour bénéficier de fonctionnalités plus avancées, il est pertinent d'utiliser les vecteurs.

Création

Il existe deux façons de créer un vecteur : l'appel à un fonction associée et une macro. Les fonctions associées sont, comme leur nom l'indique, des fonctions associées à des types. Le type `Vec` possède une méthode associée `new` qui permet de créer un vecteur :

```
let v = Vec::new();
```

Compilez et observez ce que vous dit le compilateur.

`Vec` est un type générique

C'est à dire qu'il est paramétrable en fonction du type des éléments qu'il va contenir.

Ici le type des éléments n'étant pas précisé, le compilateur renvoie une erreur. Et nous suggère de préciser le type au moment de la déclaration :

```
`let v: Vec<i32> = Vec::new();`
```

C'est une option mais on peut aussi tirer partie de la déduction de type du compilateur en ajoutant un élément d'un type bien défini dans le vecteur :

```
let v = Vec::new();
v.push(12);
```

Tel quel, le code ne compile pas. En prenant en compte le message d'erreur de compilation, corrigez le code puis appréciez l'intérêt de la déduction de type.

L'autre façon de créer un nouveau vecteur utilise une **macro** :

```
let v1 = vec![12;100];
println!("v1 : {:?}", v1);
```

Comparez avec la sortie de :

```
let v2 = vec![12,100];
println!("v2 : {:?}", v2);
```

Un petit caractère peut faire une grande différence.

Accès

L'accès au éléments d'un vecteur se fait de manière classique avec l'opérateur `[ ]` :

```
let mut v = Vec::new();
v.push(12);
println!("first element of v : {}", v[0]);
```

Rust étant un langage moderne, il existe des outils efficaces pour spécifier des intervalles :

```
v.extend(1..20);
println!("first element of v : {}", v[0]);
println!("Some elements of v : {:?}", &v[2..7]);
println!("Some other elements of v : {:?}", &v[5..=10]);
println!("Yet other elements of v : {:?}", &v[..4]);
println!("Yet again other elements of v : {:?}", &v[16..]);
```

Itération - Parcours

Pour parcourir les éléments d'un vecteur sans les modifier, il y a le classique :

```
println!("v with for : ");
for e in v.iter() {
    print!("{e},");
}
```

Ou sa forme fonctionnelle :

```
println!("v with functional : ");
v.iter().for_each(|e| print!("{e},"));
```

Pour modifier les éléments de `v`, vous devez obtenir une référence mutable sur ses éléments à l'aide de la fonction `iter_mut` :

```
for e in v.iter_mut() {
    *e += 100;
}
println!("v after modification : {:?}", v);
```

Pour accéder à l'élément à travers sa référence, on utilise, comme en C, l'opérateur '*'.

Premier tuple

Les tableaux et les vecteurs contiennent des éléments du même type. Pour construire une liste d'éléments de types différents Rust offre le type `tuple` :

```
let t1 = (12, 'a', true);
println!("v2 : {:?}", t1);
```

L'accès aux éléments d'un tuple se fait de deux façons :

- En utilisant un index : `let b : bool = t1.2;`
- En utilisant un **pattern** :

```
let t1 = (12, 'a', true);
let (un_entier, un_char, un_booleen) = t1;
println!("le booleen : {}", un_booleen);
```

À la deuxième ligne le tuple 't1' est "déstructuré" pour initialiser les trois variables. On peut ensuite accéder à chacune des variables indépendamment.

Vecteur de tuples

On peut combiner les deux et travailler avec un vecteur de tuples :

```
let vt = vec![(42,43);10];
vt.iter().for_each(|(a,b)| print!("{}, ", a+b));
```

Remplacez l'initialisation du vecteur par :

```
let vt = vec![(5,"abc ");3];
```

Modifiez ensuite la fonction passée au `for_each` pour qu'elle affiche le deuxième élément du tuple le nombre de fois correspondant au premier élément du tuple. Vous devez obtenir l'affichage : `abc abc abc abc abc , abc abc abc abc abc , abc abc abc abc abc ,`

Observations

À retenir de ces premières manipulations :

- `cargo` est l'ami du développeur Rust pour :
  - la **génération** d'un projet,
  - la **compilation** et l'**exécution** d'un programme (fini les `Makefile`),
  - la **génération** de la documentation.
  - le **passage des tests**.
- Un projet Rust par défaut obéit à des **conventions** (`src`, `target`, etc.).
- Rust est un langage fortement typé mais le compilateur déduit les types la plupart du temps.

Plus loin

Une fois que vous avez terminé, explorez :

- Le livre sur Rust ou
- La visite guidée

Réalisez ensuite le tutoriel du livre.

Un lien vers le TP suivant : Application TDD sur palindrome