Devoir maison: Programmation systèmes et Réseaux en Rust, contrôle de connaissances

Axel Viala <axel.viala@darnuria.eu>

21 janvier 2019

Nom et Prénom :_____

Classe :_____

Rendu : Vous devez rendre ce devoir, avant le dimanche 31 janvier 18h par courriel avec le sujet «`[rust-esgi] QCM "votre promotion" "nom" "prenom"`» a mon adresse *axel.viala@darnuria.eu*. Vous pouvez soit répondre directement sur le PDF, soit imprimer et scanner/(scanner avec ordiphone) et m'envoyer le scan. Le QCM est à faire seul, la clarté sera un plus, les mauvaises réponses font perdre des points. **Objectifs :** Le but du contrôle de connaissances en début de cours est pour vous de vérifier où vous en êtes par rapport au cours précédent.
Il s'agit pour moi un moyen de vérifier que la pédagogie est adaptée à la classe.
Notation : Les points sont indiqués à titre d'information, la notation peut changer pour des raisons d'harmonisation. Les réponses fausses en QCM font perdre des points.

1 Culture générale autour de Rust

1. En Rust, avez-vous à écrire les types par vous-même :
   - ◯ Oui comme en C++ sans `auto`
   - √ **Non le compilateur infére les types et parfois me demande du secours**
   - ◯ Comme en JavaScript ou Python les types sont gérées à l'exécution
2. En Rust, quelles propositions sont vraies :
   - ◯ Rust gère la mémoire avec un gabarge collector
   - √ **En Rust les allocations sont gérées par scope, le compilateur place les appels aux destructeurs**
   - ◯ En Rust par défaut `rustc` compile pour une machine virtuelle
   - √ **`rustc` compile en assembleur natif pour mon ordinateur**
   - ◯ La mémoire allouée est dans le segment de tas (HEAP) sans que je le demande explicitement
   - ◯ C'est moi qui appelle `free` pour libérer les allocations sur le tas/heap
   - ◯ La gestion par défaut de la mémoire est gérée par un compteur de référence comme en Swift

2 Syntaxe de Rust

3. En Rust, le mot clef `let` sert à :
   - ◯ Faire un branchement conditionnel *Il existe le* `if-let` *mais c'est pas vraiment un branchement classique.*
   - √ **Lier une expression à un nom (déclarer une variable).**
   - ◯ Par défaut la variable sera dans le tas/heap
   - √ **Par défaut la variable sera allouée dans la pile/STACK**
   - √ **Il peut servir à destructurer une expression ex :** `let (a, b) = (1, 42);`
4. En Rust, par défaut les variables sont :
   - √ ***imutables* / constantes**
   - ◯ mutables
5. Que dois-je faire si je souhaite échanger entre plusieurs threads en Rust ?
   - √ **Par mémoire partagée : avec un compteur atomique de références `std::sync::Arc` doc et une `std::sync::Mutex` doc**
   - √ **Par passage de messages : avec des channels : voir `std::sync::mpsc` doc**
   - ◯ Une simple référence mutable suffit
   - ◯ Je ne peux pas
6. A quoi sert `std::sync::Mutex` ?
   - ◯ Compter des références
   - √ **Protéger un accès concurent à une ressource**
   - ◯ Déterminer quand libérer la mémoire
7. A quoi sert `std::sync::Arc` ?
   - √ **Compter des références de façon atomique pour liberer si le compteur est a 0**
   - ◯ Protéger un accès concurent à une ressource
   - √ **Déterminer quand libérer la mémoire (J'ai été généreux et pas compter l'oubli de cette réponse).**
8. Quelles propositions pour le code Rust suivant sont vraies.

```rust
// Rappel: Vec n'implemente pas copy.
let v = vec![2, 42, 1];
let s = v;
println!("{}", v[0]); // v[0] fait un borrow.
```

   - ◯ Ce code compile
   - √ **`s` est un *move* (capture), de `v` donc `v` n'est plus utilisable**
   - ◯ `s` est un *borrow* (emprunt), de `v`.
9. Expliquer avec vos mots le concept de *move semantics* (capture ou déplacement en Français)/.

> Un *move* ou *déplacement* est l'action lorsque une valeur change de propriétaire (*owner*), par exemple lors d'un appel de fonction, une liaison avec `let` ou lors d'une capture dans une closure.
> Cette variable n'est alors plus utilisable ayant été déplacée. Dans le cas ou on désire pas de déplacement ni de copie on peut faire un emprunt. Certains types implementent *Copy et Clone* et donc le problème ne se pose pas (ex : `i32`).

10. En Rust, implémenter le `trait Copy` fait que votre type sera :
    - √ **Dupliqué et passé par copie bit à bit.**
    - √ **Après un passage de paramètre j'ai donc 2 occurences de mon types dans la mémoire**
    - ◯ Modifier la copie modifie l'original
    - ◯ C'est gratuit en terme de performance pour les grosses structures
11. Le type `std::boxed::Box` sert a quoi ? Dans quels cas en avez vous absolument besoin ?

> `std::boxed::Box` permet de faire des allocations sur le tas *Heap* et manage sa desallocation grace à l'ownership. Il s'agit d'un pointeur dit intelligent comparable à un `std::unique_ptr<T>` de *C++*.

12. A quoi sert le mot clef `match` ?
    - √ **Faire du filtrage par motif *pattern matching***
    - ◯ Faire des allocations sur la Heap
    - √ **Déconstruire des `enum` selon leur contenu**
    - ◯ Sur une `enum` je peux filtrer seulement certains variants d'une `enum`
    - √ **Sur une `enum` je dois être exhaustif et filtrer tout les variants**
    - √ **Je peux matcher autre chose que des `enum` par exemle entiers et chaines**
13. En Rust je peux emprunter *borrow* une référence sur une valeur avec `&` et *borrow* mutablement avec `mut &`. Quelles sont les règles ?
    - √ **Une valeur immutable peut être partagée imutablement plusieurs fois en lecture**
    - √ **Une valeur mutable peut être partagée mutablement par un lecteur/écrivain**
    - ◯ Une valeur mutable peut être partagée mutablement par plusieurs lecteur/écrivain
    - ◯ Une valeur immutable peut être partagée mutablement quand je veux

    Soit le code suivant : code sur le playpen

```rust
fn main() {
    // Indice: les &str n'implémentent pas `Copy`, ni `Clone`.
    // Indice2: Si vous n'êtes pas Copy vous êtes Move! ;)
    let mut s = vec![42, 21, 0];
    let a = &s[1];
    s.remove(1);
    println!("{}", a);
}
```

14. Expliquer succinctement pourquoi ce code ne compile t-il pas ? (indice borrow mutable/immutable)

> Ici *Rust* interdit

2.1 struct et question de mémoire

15. Soit le code suivant : (lien playpen)

```rust
use std::boxed::Box;

struct Point {
    x: i32,
```

```
    y: i32
}

fn main() {
    let p = Point { x: 2, y: 4 };
    let h = Box::new(Point { x: 6, y: 42 });
}
```

16. Quelle place occupe la structure `Point` en mémoire ? Indice : doc size_of
    - √ **64bits**
    - ◯ 128bits
    - ◯ 42bits
    - ◯ 32bits
17. Où est située la valeur de `p` ?
    - √ **Stack/Pile**
    - ◯ Heap/Tas
18. Où est située la valeur de `h` ?
    - ◯ Stack/Pile
    - √ **Heap/Tas**
19. Où est située le pointeur `h` ?
    - √ **Stack/Pile**
    - ◯ Heap/Tas
20. Dessiner comment est organisée la mémoire pile et tas à la fin de la fonction `main()` avant les dessallocations. Préciser ce qui est le Tas/Heap, la Pile/Stack et travers les éventuels pointeurs. Indice : les questions précédentes devraient être des indices.

    ☐

21. Le type `std::boxed::Box` sert a quoi ? Dans quels cas en avez vous absolument besoin ?

    ☐

2.2 Enum, mémoire et developpement

Soit un type `Peano` qui implémente des nombres de *Peano*. lien playppen

```
use std::boxed::Box;

enum Peano {
    Zero,
    S(Box<Peano>)
}

fn main() {
    let z = Peano::Zero;
    let h = Peano::S(Box::new(Peano::Zero));
}
```

22. Où est située la valeur de `z` ?
    - √ **Stack/Pile**
    - ◯ Heap/Tas
23. Où est située la valeur de `h` ?

- √ **La valeur du variant `Peano::S` est dans la Stack/Pile**
- √ **`Zero` dans le Heap/Tas**

24. Dessiner comment est organisée la mémoire pile et tas à la fin de la fonction `main()` avant les dessallocations. Préciser ce qui est le Tas/Heap, la Pile/Stack et travers les eventuels pointeurs. Indice : les questions précédentes devraient être des indices.

On veut pouvoir écrire des fonctions pour manipuler nos nombres de *Peano*. Vous devrez dans les questions suivantes, remplacer les *???*, écrire le code des fonctions dans un **fichier** nommée `nom_prenom_promo.rs`.
Indice : vous aurez besoin du mot clef **`match`** et pourquoi pas de récursivité pour vous simplifier l'écritude du code. lien playpen rust

```rust
impl Peano {
    /// Crée un nombre de peano depuis un nombre natif sur 32bits.
    /// ```rust
    /// assert_eq(new_from_i32(2), Peano::S(Box::new(Peano::S(Box::new(Peano::Zero)))));
    /// ```
    fn new_from_i32(n: u32) -> Peano {

    }

    /// Transforme un Peano en nombre
    /// ```rust
    /// assert_eq(Peano::S(Box::new(Peano::Zero)).to_i32(), 1);
    /// ```
    fn to_i32(???) -> u32 {

    }

    /// Additionne deux Peano entre eux.
    /// Exemple
    /// ```rust
    /// assert_eq!(p, p.add(Peano::Zero));
    fn add(???, ???) -> ??? {

    }
}
```

25. Écrivez le code de la fonction `new_from_i32`.

26. Écrivez le code de la fonction `to_i32`.

27. Écrivez le code de la fonction `add`.